

Python

Full stack Skills Bootcamp

Introducing Python Iterators

■ What are Iterators?

- Iterators are objects that allow you to traverse through all elements of a collection (like lists, strings, etc.) one at a time.
- They implement the `__iter__()` and `__next__()` methods.

■ Difference between iterables and iterators:

- Iterable: An object capable of returning its elements one at a time.
- Iterator: The object that produces the next element from the iterable when `next()` is called.



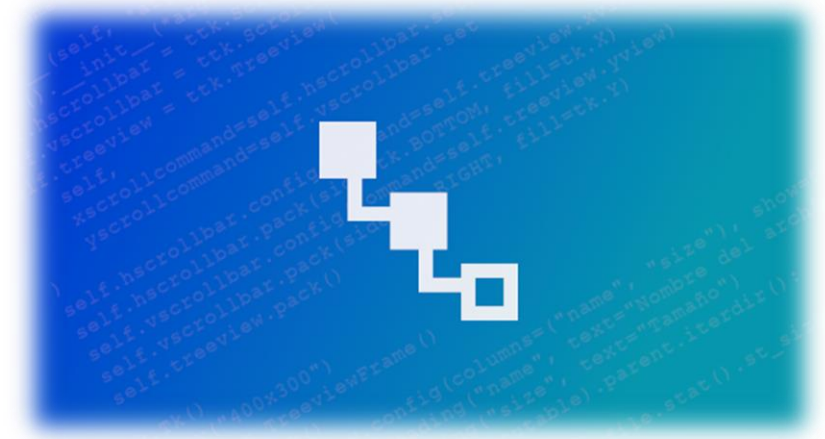
Basic Iterator Example

■ Basic Iterator Example with Lists:

python

```
my_list = [10, 20, 30, 40, 50]
iterator = iter(my_list)
print(next(iterator)) # Output: 10
print(next(iterator)) # Output: 20
print(next(iterator)) # Output: 30
```

- Lists are iterables, meaning we can convert them into an iterator using the `iter()` function.
- Using `next()`, we retrieve the next element in the sequence.
- The iterator internally maintains its state, meaning it “remembers” the current position in the sequence.
- Important: When the iterator has no more elements, it raises a `StopIteration` exception.




StopIteration Exception

■ Handling the End of an Iterator:

```
python  
  
# print(next(iterator)) # Raises StopIteration
```

- When the iterator is exhausted (i.e., all elements are iterated over), calling next() again raises a StopIteration exception.
- This is how Python signals that there are no more elements left to iterate.
- In large datasets, this behavior can help efficiently manage resources and stop when necessary.



```
KeyboardInterrupt Traceback  
/var/folders/xh/rcvz9t116mgc1sdw9wmrj3m00000gn/T/  
...:1: KeyboardInterrupt:  
3 except KeyboardInterrupt:  
4 print('User pressed key to stop run')  
  
KeyboardInterrupt Traceback  
/var/folders/xh/rcvz9t116mgc1sdw9wmrj3m00000gn/T/  
3 except KeyboardInterrupt:  
4 print('User pressed key to stop run')
```

StopIteration

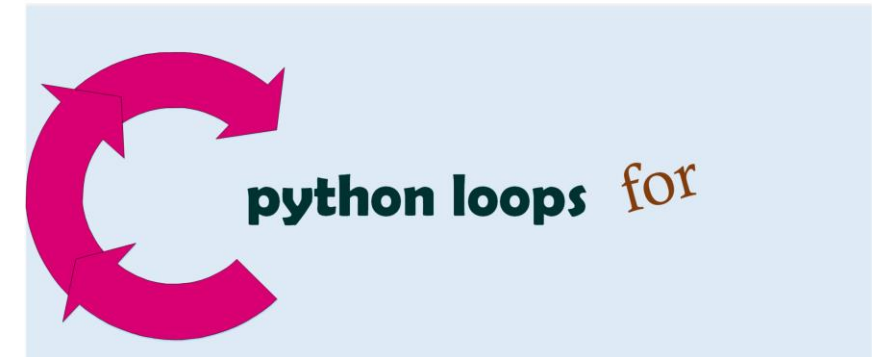
During handling of the above exception, another exception occurred

Using for loop with an Iterator

■ For Loop Simplifies Iteration:

```
python  
  
my_list = [10, 20, 30, 40, 50]  
for item in my_list:  
    print(item)
```

- Python's for loop abstracts the use of `iter()` and `next()` functions.
- The for loop automatically handles the iteration and stops when `StopIteration` is raised.
- Why use for loops: They provide a cleaner and more readable way to iterate over collections.
- This pattern is highly optimized and preferred in Python.



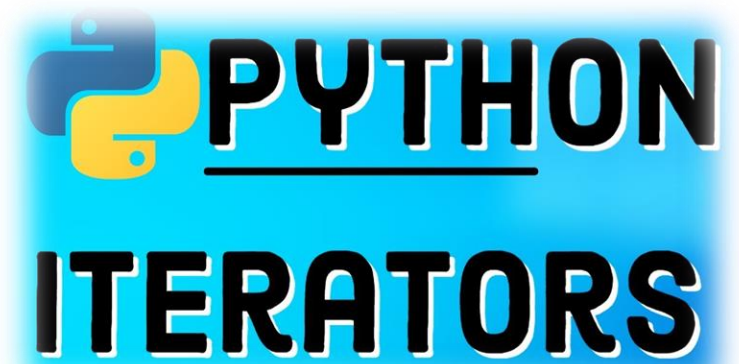
Iterating over Strings

■ Strings as Iterables

```
python

my_string = "Step8up"
string_iterator = iter(my_string)
print(next(string_iterator)) # Output: S
print(next(string_iterator)) # Output: t
print(next(string_iterator)) # Output: e
```

- Just like lists, strings are also iterables.
- We can create an iterator from a string using `iter()`.
- Using `next()`, characters in the string are retrieved one at a time.
- This concept applies to all iterables, such as tuples, sets, and dictionaries.



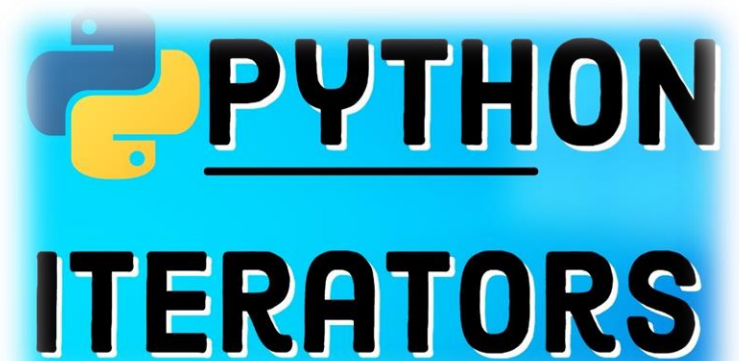
Built-in Functions with Iterators

■ Working with Built-in Functions

python

```
fruits = ['apple', 'banana', 'cherry']  
for index, fruit in enumerate(fruits):  
    print(f"{index}: {fruit}")
```

- Python offers various built-in functions to enhance working with iterators.
- `enumerate()` is a common function that returns both the index and the element during iteration.
- This is particularly useful when both the element and its position are needed.

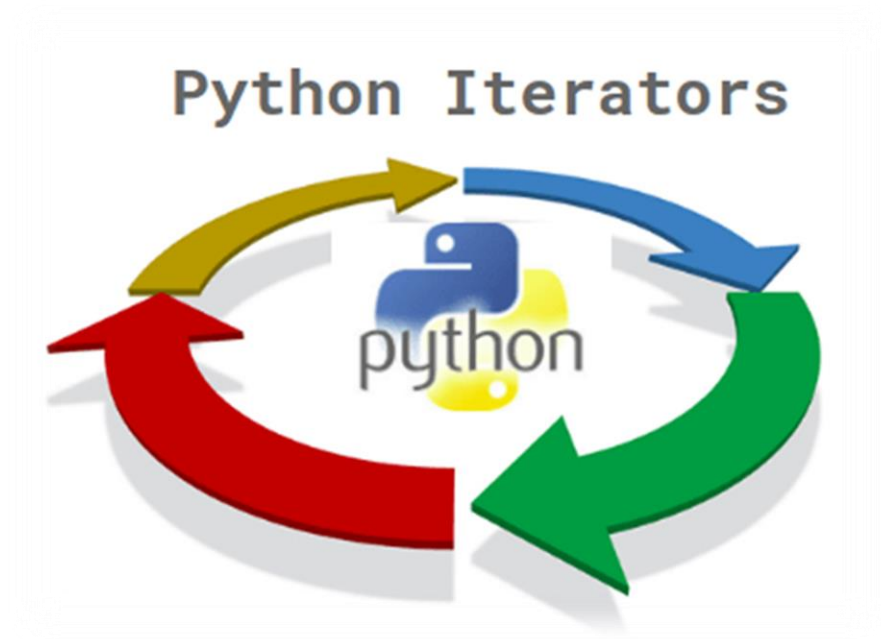


Concluding Iterators

■ Real-World Use Cases of Iterators

Iterators are used extensively in Python, especially when working with:

- Large datasets: Iterators avoid loading all elements into memory at once.
- Generators: They are a type of iterator and provide a memory-efficient way of handling data.
- File reading: Iterators are used when processing files line by line.
- Data streaming: Iterators are ideal for streaming and processing data chunks in real-time.



Introducing Python Comprehensions

■ What are Comprehensions?

- List comprehensions are a concise way to create lists in Python. They allow you to define the content of the list using a single line of code, which can enhance readability and efficiency.
- Syntax:
`new_list = [expression for item in iterable if condition]`

python

```
squares = [x**2 for x in range(1, 6)]  
print("List of squares:", squares)
```

Output: [1, 4, 9, 16, 25]



List
Comprehensions

```
[ x for x in range(10) ]
```

Filtering with List Comprehensions

■ Filtering

python

```
even_numbers = [x for x in range(1, 11) if x % 2 == 0]  
print("List of even numbers:", even_numbers)
```

- List comprehensions are not just for constructing new lists.
- They can also filter elements based on conditions.
- By adding an if clause at the end, you can specify which items to include in the new list.

LIST COMPREHENSION
iteration
[**a x 2** for a in input_list]
expression

Set Comprehensions

■ What are set comprehensions:

python

```
unique_squares = {x**2 for x in range(1, 6)}  
print("Set of unique squares:", unique_squares)
```

- Set comprehensions are like list comprehensions, but they create sets, which are unordered collections of unique elements.
- This means that any duplicate values are automatically removed when using a set comprehension.

Python Set Comprehension



Dictionary Comprehensions

■ What are set comprehensions:

python

```
square_dict = {x: x**2 for x in range(1, 6)}  
print("Dictionary of numbers and their squares:", square_dict)
```

- Dictionary comprehensions allow you to create dictionaries in a clean and efficient manner.
- Just like with list and set comprehensions, you can build dictionaries using a concise syntax that includes both keys and values.



Dictionaries

Dictionary Comprehension



Advantages of Comprehensions

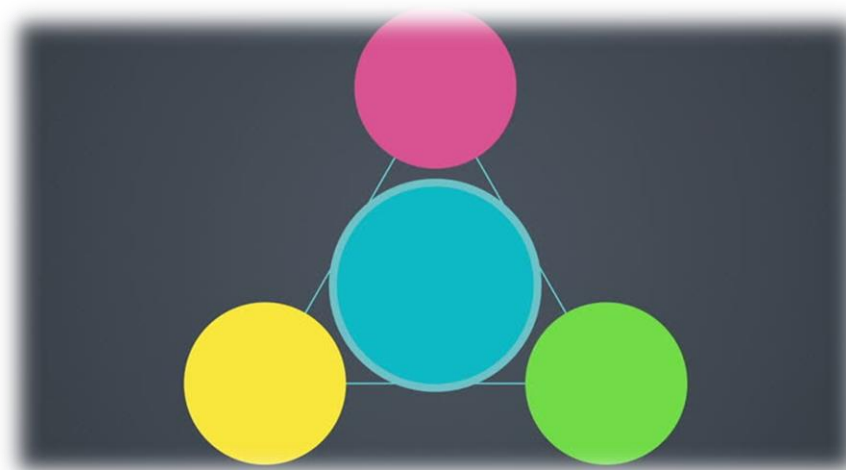
- **Conciseness:** Comprehensions provide a way to write less code compared to traditional loops, making it easier to understand the purpose of the code briefly.
- **Efficiency:** They can be more efficient in terms of both time and space, as comprehensions can often eliminate the need for intermediate storage of data that loops might require.
- **Versatility:** Comprehensions can be used to create lists, sets, or dictionaries, allowing you to choose the best data structure for your specific use case while maintaining a similar syntax.



Concluding Comprehensions

■ So,

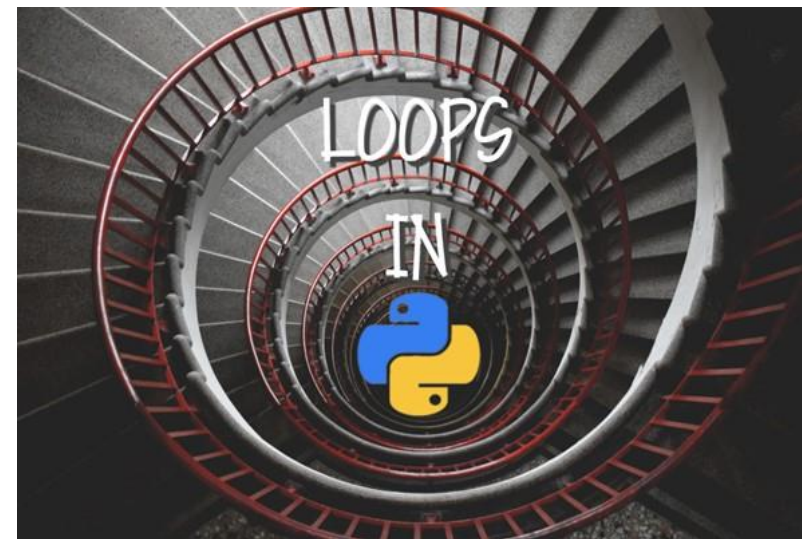
Python comprehensions are a powerful feature that enables the creation of lists, sets, and dictionaries in a concise and efficient manner. They help streamline your code and make it more readable.



Introducing Python Loops

■ What are Loops?

- Loops are constructs that allow you to repeat a block of code multiple times based on a condition or the elements of a sequence.
- Types of Loops:
 - For Loop: Iterates over a sequence (like a list or tuple).
 - While Loop: Repeats if a condition is true.



For Loop with Lists

python

```
fruits = ["apple", "banana", "cherry"]  
print("Fruits list:")  
for fruit in fruits:  
    print(fruit)
```

- This loop iterates over a list of fruits, printing each fruit in the list.

for Loop with Lists

• List

```
cars = ['Audi', 'BMW', 'Toyota']
```

```
for i in range(len(cars)):  
    print(cars[i])
```

range() Function

90 / Python

For Loop with Tuples & Ranges

python

```
numbers = (1, 2, 3, 4, 5)
print("\nNumbers tuple:")
for number in numbers:
    print(number)
```

- Like lists, this loop iterates over a tuple of numbers, displaying each number.

python

```
print("\nRange from 0 to 4:")
for i in range(5):
    print(i)
```

- This loop uses the range() function to iterate over a sequence of numbers from 0 to 4.

PYTHON
LOOP TUPLES




While Loop

python

```
count = 1
print("\nCounting with while loop:")
while count <= 3:
    print(count)
    count += 1
```

- This loop continues to execute as long as the count is less than or equal to 3, incrementing count each iteration



**WHILE LOOPS
ARE AWESOME**

```
while 1 == 1:
    print("I'M STUCK IN A LOOP!")
```

I'M STUCK IN A LOOP!
I'M STUCK IN A LOOP!
I'M STUCK IN A LOOP!

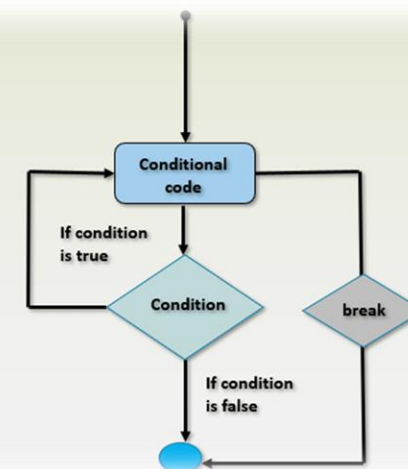
Break Statement

python

```
print("\nLoop with break statement:")  
for i in range(5):  
    if i == 3:  
        break  
    print(i)
```

- The loop exits early when i reaches 3 due to the break statement.

Break Statement in Python

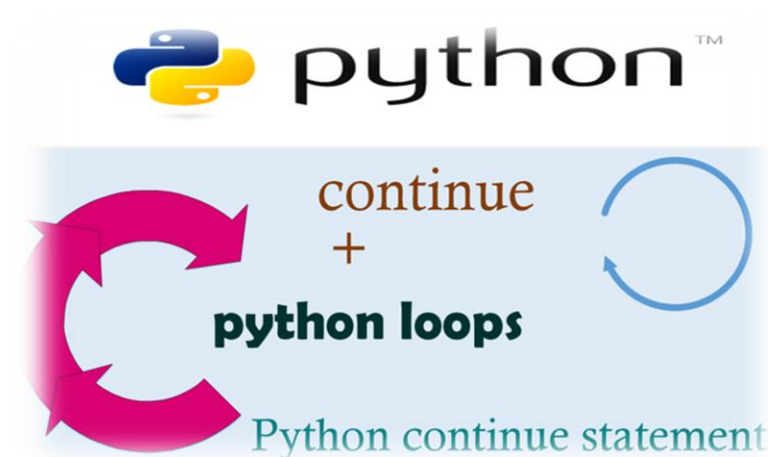


Continue Statement

python

```
print("\nLoop with continue statement:")  
for i in range(5):  
    if i % 2 == 0:  
        continue  
    print(i)
```

- This loop skips printing even numbers, only displaying the odd ones due to the continue statement.

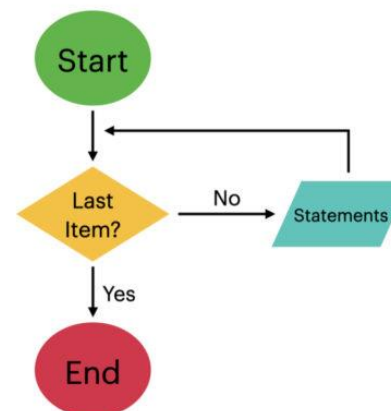


Concluding Loops

■ Key Points

- Loops allow repetitive execution of code.
- For Loop: Best for iterating over sequences.
- While Loop: Useful when the number of iterations is not known.
- Control flow statements (break and continue) enhance loop functionality.

For Loop



While Loop

