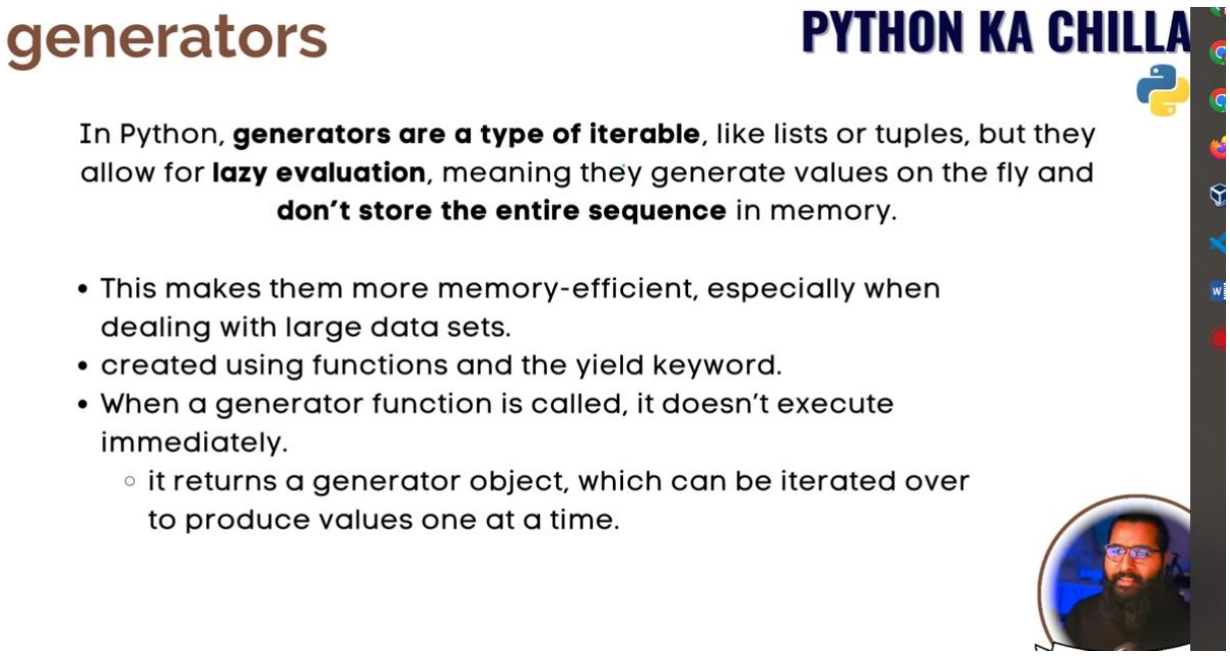
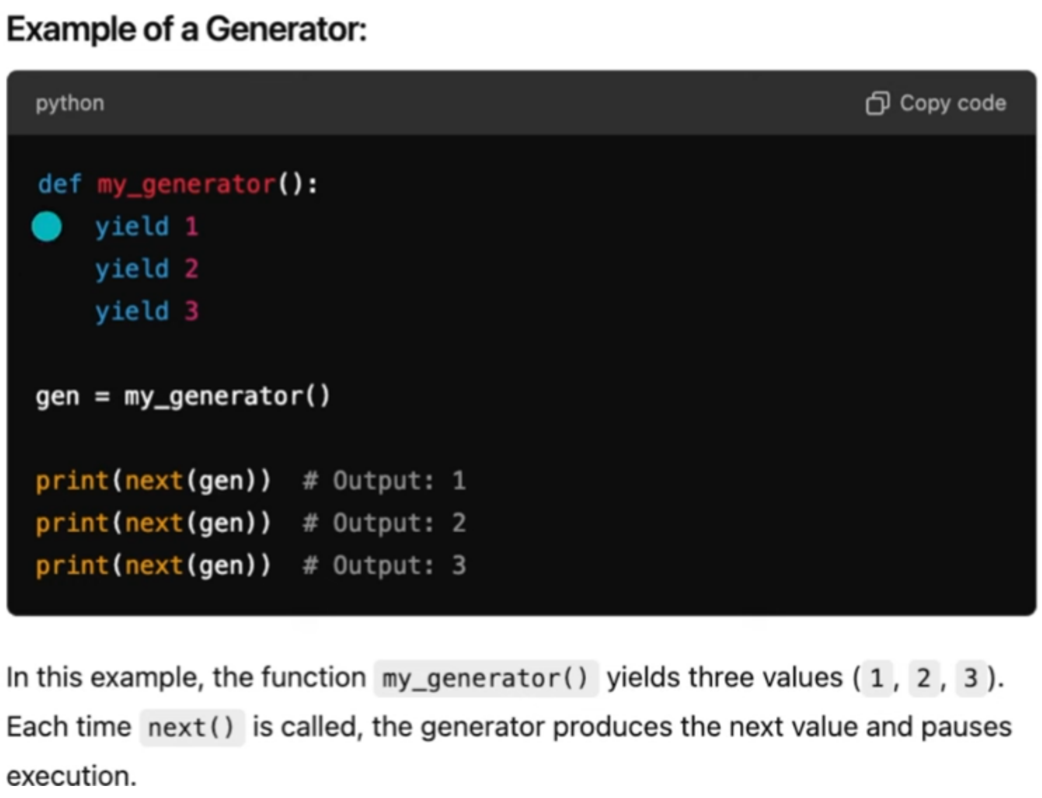


generators

In Python, **generators are a type of iterable**, like lists or tuples, but they allow for **lazy evaluation**, meaning they generate values on the fly and **don't store the entire sequence** in memory.

- This makes them more memory-efficient, especially when dealing with large data sets.
- created using functions and the yield keyword.
- When a generator function is called, it doesn't execute immediately.
  - it returns a generator object, which can be iterated over to produce values one at a time.



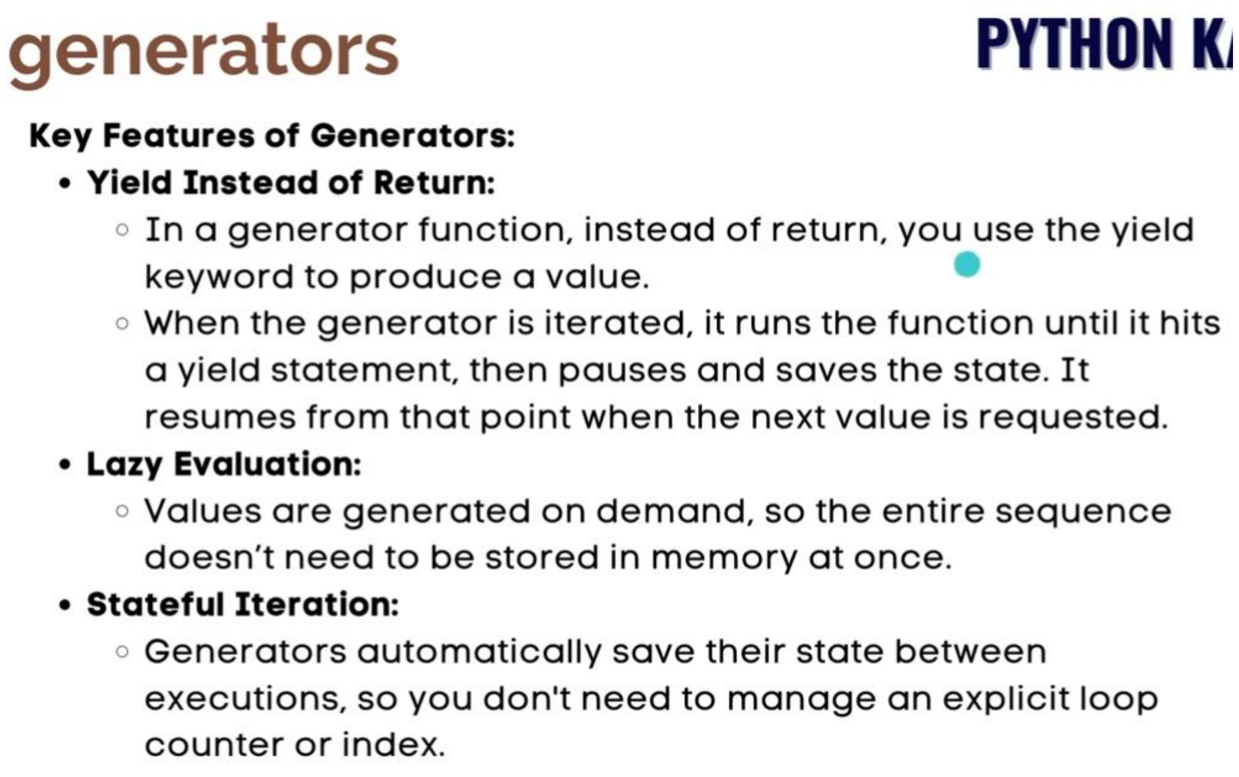
Example of a Generator:

```python
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

In this example, the function `my_generator()` yields three values (`1`, `2`, `3`). Each time `next()` is called, the generator produces the next value and pauses execution.

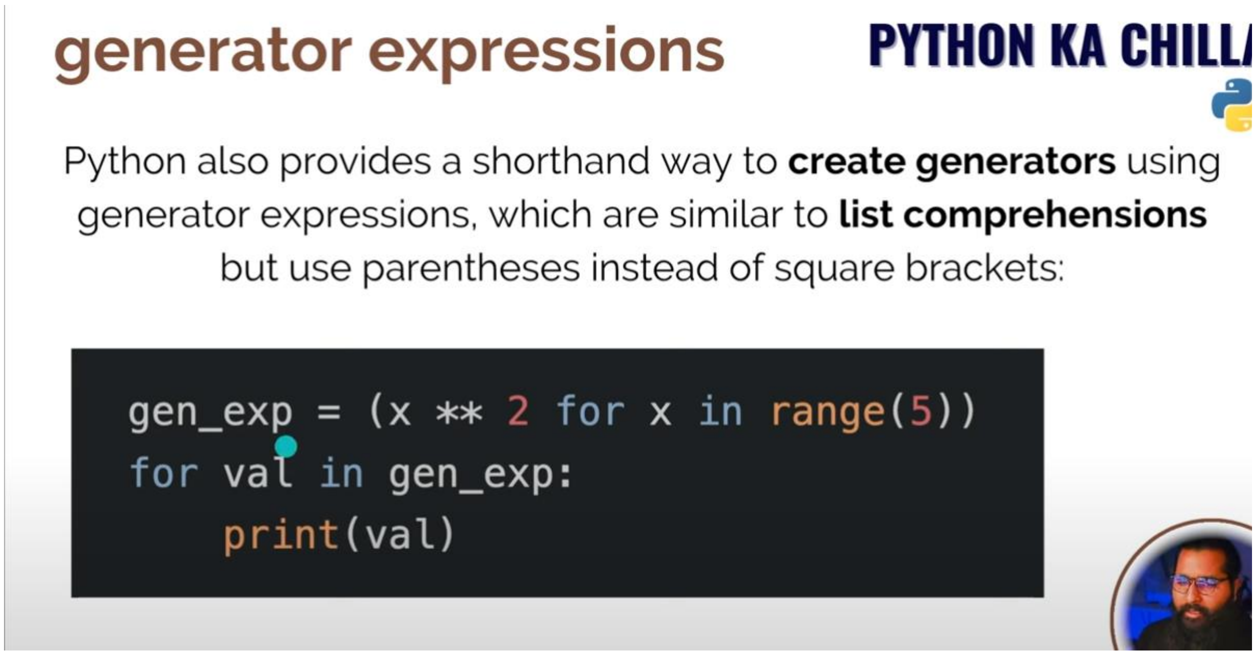


generators

PYTHON K

Key Features of Generators:

- **Yield Instead of Return:**
  - In a generator function, instead of return, you use the yield keyword to produce a value.
  - When the generator is iterated, it runs the function until it hits a yield statement, then pauses and saves the state. It resumes from that point when the next value is requested.
- **Lazy Evaluation:**
  - Values are generated on demand, so the entire sequence doesn't need to be stored in memory at once.
- **Stateful Iteration:**
  - Generators automatically save their state between executions, so you don't need to manage an explicit loop counter or index.



generator expressions

PYTHON KA CHILL

Python also provides a shorthand way to **create generators** using generator expressions, which are similar to **list comprehensions** but use parentheses instead of square brackets:

```python
gen_exp = (x ** 2 for x in range(5))
for val in gen_exp:
    print(val)
```