
YASH
Technologies
SM
More than what you think.
Python Generator
© 2017 YASH Technologies | www.yash.com | Confidential

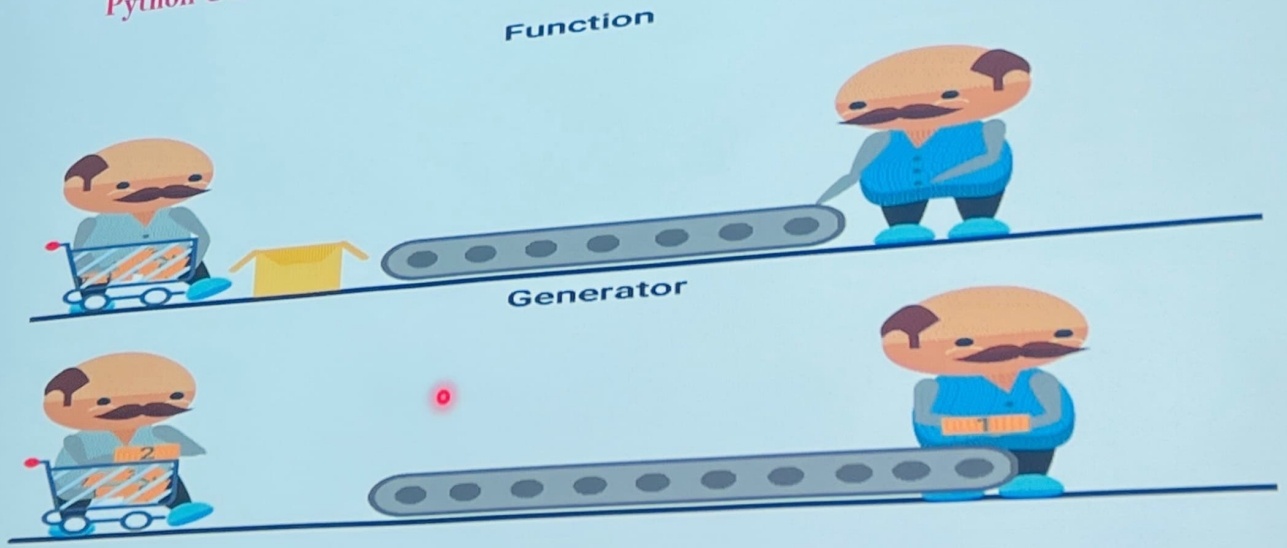
Agenda

- What is a Generator?
- Python function vs generator.
- Challenges Solved by Generators
- Python lazy evaluation.
- Python processing large data sets.
- Python creating custom iterators.

Python Generators

```
def generator_name( ) :
    #statements
    yield something
```

Python Generators: Understanding the Power and Different Use Cases



Problem:

Imagine you're processing a file with 1 crore lines. You want to print each line, or calculate something from it. What will happen if you load the entire file into memory at once?
Then it may be-

- "It'll get crash"
- "Memory error"
- "Too slow"

So, traditional functions store all data in memory.
But what if we could process one item at a time, like streaming —without holding everything?

That's where **Python Generators** come in.

What is a Generator in Python?

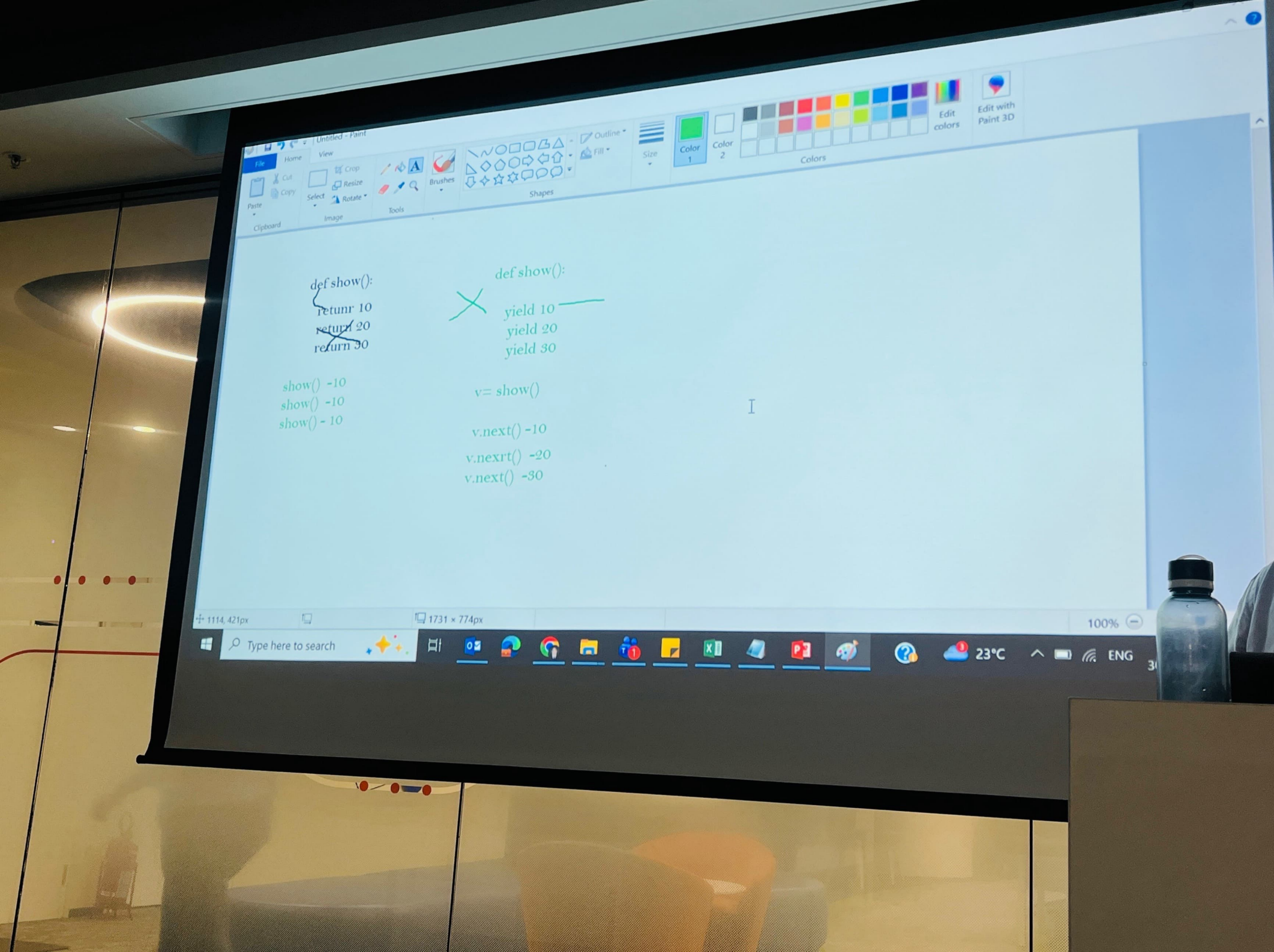
A **generator** is a special type of **iterator** in Python that allows you to iterate over data **one item at a time, on demand**, using the yield keyword.

Difference from regular functions:

- Regular functions use return and exit immediately.
- Generator functions use yield, which **pauses** the function and **resumes** from where it left off.

Basic Syntax:

```
def generator_function():
    yield value1
    yield value2
    yield value3
```


Untitled - Paint
File
Home
View
Paste
Cut
Copy
Clipboard
Crop
Resize
Rotate
Select
Image
Brushes
Tools
Shapes
Outline
Fill
Size
Color 1
Color 2
Colors
Edit colors
Edit with Paint 3D
def show():
returnr 10
return 20
return 30
show() -10
show() -10
show() - 10
def show():
yield 10
yield 20
yield 30
v= show()
v.next() -10
v.nexrt() -20
v.next() -30
1114, 421px
1731 × 774px
100%
Type here to search
23°C
ENG

Normal Function v/s Generator

•**Normal Function:** Executes from start to finish in one go. It uses the return statement to send a single value back to the caller and then terminates.

Normal Function:

```
def normal():
    return [1, 2, 3]
```

•**Generator Function:** Uses the yield statement instead of return. When yield is encountered, the function pauses its execution, sends a value back to the caller, and remembers its state. When called again (e.g., in a loop or with next()), it resumes execution from where it left off.

Generator Function:

```
def generator():
    yield 1
    yield 2
    yield 3
```

Here, "A normal function returns all values immediately. A generator *yields* values one by one — it pauses and resumes."

What is yield in Python?

The yield keyword is used **inside a function** to make it a **generator function**. When Python sees a yield inside a function, it doesn't treat it like a normal return. Instead, it makes the function **pause** and **return a generator object**.

Difference between return and yield:

return → gives the final result and stops the function.
yield → gives one result at a time and pauses, allowing continuation later.

Eexecution Flow

return Flow:

Function Start → Execute Code → return → Function End

yield Flow:

Function Start → yield (pause) → resume → yield (pause) → ... → Function End

Why Should We Use Generators?

Generators are useful when:

- You want to **save memory**.
- You're working with **large datasets** or **infinite streams**.
- You want **lazy evaluation** (compute values only when needed).

Problems We Faced Before Generators

Problem 1: Memory Issues with Large Lists

```
def create_large_list(n):
    result = []
    for i in range(n):
        result.append(i * 2)
    return result

# This creates a list with 1 million elements in memory!
large_list = create_large_list(1000000)
for item in large_list:
    if item > 100:  # We only need first few items
        break
```

Problem:

- Entire list created in memory
- Wasted computation for unused elements
- Memory overflow for very large n

Solution 1: Memory-Efficient Large Sequences

With Generator

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

# Only one value in memory at a time!
gen = large_sequence_generator(1000000)
for item in gen:
    if item > 100:
        break  # Rest of values never computed!
```

State Preservation

State preservation is a core feature of Python generators that allows them to pause their execution and then resume it exactly where they left off. When a generator function uses the yield keyword, it returns a value but does not terminate. Instead, it saves its entire internal state, including the values of all local variables and the precise point of execution.

Example of state preservation:

```
def count_up_to(n):
    num = 1
    while num <= n:
        yield num
        num += 1

# Create a generator object
counter = count_up_to(3)
# First call: Execution starts
print(next(counter)) # Output: 1
# Second call: Execution resumes from where it left off
print(next(counter)) # Output: 2
# Third call: Execution resumes again
print(next(counter)) # Output: 3

# Fourth call: The generator is exhausted
print(next(counter)) # Raises StopIteration
```

Python Generator use Case

Use Case 1: Reading Large Files

```
def read_large_file(file_path):
    """Generator to read large files line by line"""
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()

# Process 10GB file without loading it all in memory
for line in read_large_file('huge_file.txt'):
    if 'error' in line:
        print(f"Found error: {line}")
```

Use Case 2: Data Processing Pipeline

```
def data_processing_pipeline(data):
    """Multi-step data processing using generators"""

    # Step 1: Filter valid data
    def filter_valid(records):
        for record in records:
            if record.get('valid', False):
                yield record

    # Step 2: Transform data
    def transform_data(records):
        for record in records:
            record['processed'] = True
            yield record

    # Step 3: Add timestamps
    def add_timestamps(records):
        import time
        for record in records:
            record['timestamp'] = time.time()
            yield record

    # Chain generators
    pipeline =
        add_timestamps(transform_data(filter_valid(data)))
    return pipeline
```

Data Processing Pipeline

```
# Usage
sample_data = [
    {'id': 1, 'valid': True, 'data': 'A'},
    {'id': 2, 'valid': False, 'data': 'B'},
    {'id': 3, 'valid': True, 'data': 'C'}
]

for processed_record in
data_processing_pipeline(sample_data):
    print(processed_record)
```

Assignment

Q1: Create a generator that yields even numbers up to n.
Q2: Write a generator to yield squares of numbers from 1 to n.
Q3: Write a generator that yields Fibonacci numbers up to n terms.
Q4: Create a generator that yields lines from a text file one by one.

Q5: Write a generator that yields only words starting with a vowel from a given sentence

Example:
Input → "Apples are awesome and bananas are boring"
Output → ['Apples', 'are', 'awesome', 'and', 'are']

YAS
Technolog
More than w

Assignment

Q1: Create a generator that yields even numbers up to n.
Q2: Write a generator to yield squares of numbers from 1 to n.
Q3: Write a generator that yields Fibonacci numbers up to n terms.
Q4: Create a generator that yields lines from a text file one by one.

Q5: Write a generator that yields only words starting with a vowel from a given sentence

Example:
Input → "Apples are awesome and bananas are boring"
Output → ['Apples', 'are', 'awesome', 'and', 'are']

UseCase

1- File Processing Generator

Create a generator that reads a file and yields lines that contain a specific keyword.

2- Pipeline Generator

Create a data processing pipeline using multiple generators.

1. Simulate reading data
2. Filter data where value > 50
3. Transform data by adding new field

Lazy Evaluation

Generators can also be used for lazy evaluation. Lazy evaluation is a technique where values are only computed when they're needed, rather than being computed all at once. This can be very useful when working with large data sets or expensive computations. For example, here's a generator that generates the squares of numbers:

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

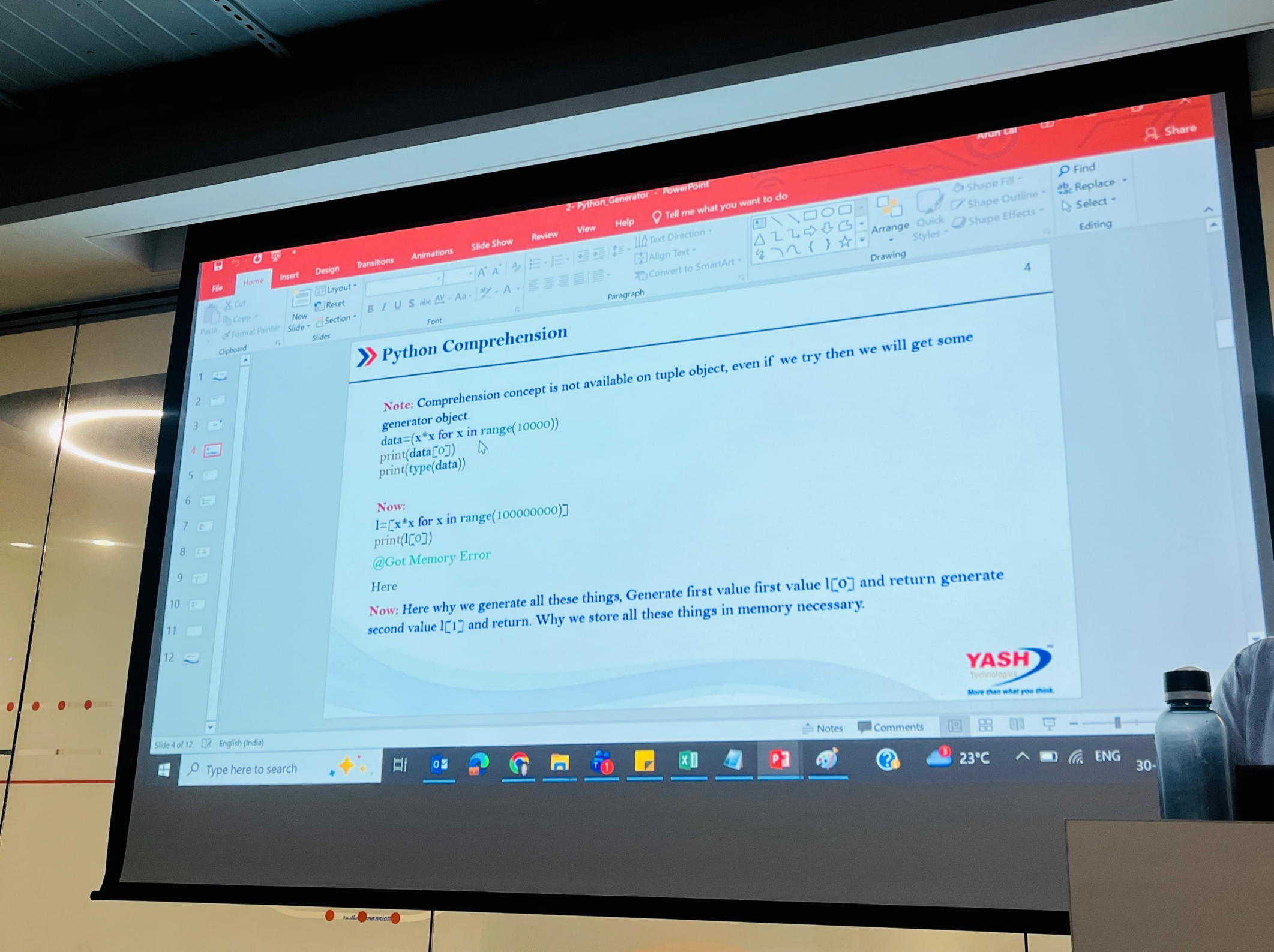
You can use this generator to generate the squares of the numbers from 0 to 9999, without computing all the squares at once:

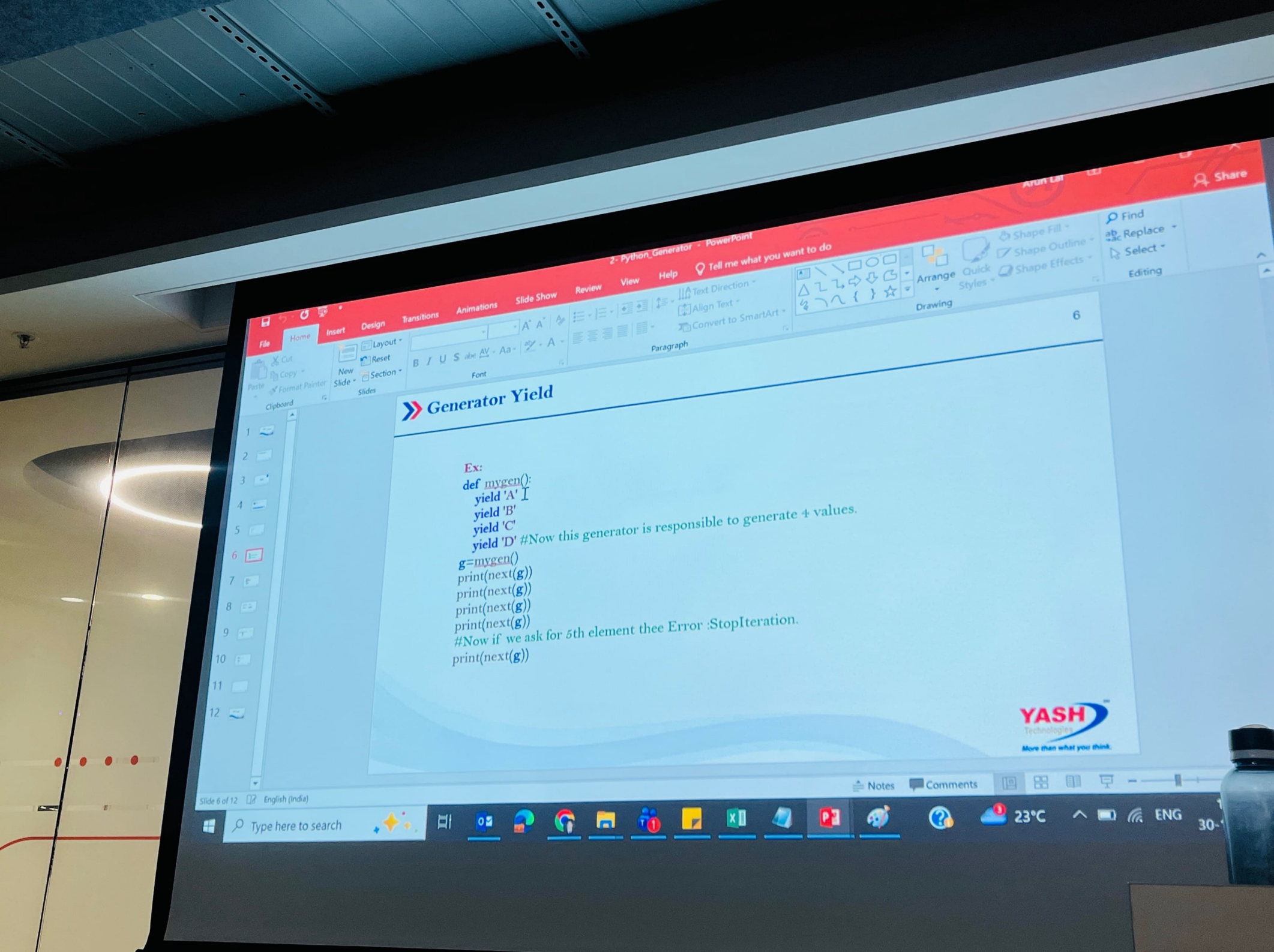
```
>>> for i in squares(10000):
...     print(i)
...
0 1 4 9
```

Yield

The **yield** keyword is a special keyword in Python that is used in generator functions to produce a sequence of values. When a generator function encounters a **yield** statement, it temporarily suspends its execution and returns the value to the caller. The generator function can then be resumed from where it left off when the next value is requested.
For example, consider the following generator function:

```
def count_up_to(n):
  i = 1
  while i <= n:
    yield i
    i += 1
```


2- Python_Generator - PowerPoint
Python Comprehension
Note: Comprehension concept is not available on tuple object, even if we try then we will get some generator object.
data=(x*x for x in range(10000))
print(data[0])
print(type(data))
Now:
l=[x*x for x in range(100000000)]
print(l[0])
@Got Memory Error
Here
Now: Here why we generate all these things, Generate first value first value l[0] and return generate second value l[1] and return. Why we store all these things in memory necessary.
YASH
Slide 4 of 12
English (India)
Type here to search
23°C
ENG


2- Python_Generator - PowerPoint
Tell me what you want to do
Share
Generator Yield
6
Ex:
def mygen():
yield 'A'
yield 'B'
yield 'C'
yield 'D' #Now this generator is responsible to generate 4 values.
g=mygen()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
#Now if we ask for 5th element thee Error :StopIteration.
print(next(g))
YASH
Technologies
Move than what you think.
Notes
Comments
Slide 6 of 12
English (India)
Type here to search
23°C
ENG

Generator Yield

Now: Use of yield

Qu : Generator number by using Genrator:

Ex: Without Genrator and (yield)

```
def first(num):
    n=1
    while n<=num:
        n=n+1
values=first(1000000000)
for x in values:
    print(x)
```

YASH Technologies
More than what you think.

Fibonacci Number

Note: Without storing if you want to generate a sequence of value then we should go for generator.

Now: Generate Fibonacci Number- 0 1 1 2 3 5 8 13.. Next number is the sum of previous 2 number.

Adv: We are not going to store anywhere, just generate and use.

Ex:

```
def fib():
    a,b=0,1
    while True:
        yield a
        a,b=b,a+b
for f in fib():
    if f<100:
        print(f)
    else:
        break;
```

YASH Technologies
Move than what you think

Fibonacci Number

Note: Without storing if you want to generate a sequence of value then we should go for generator.

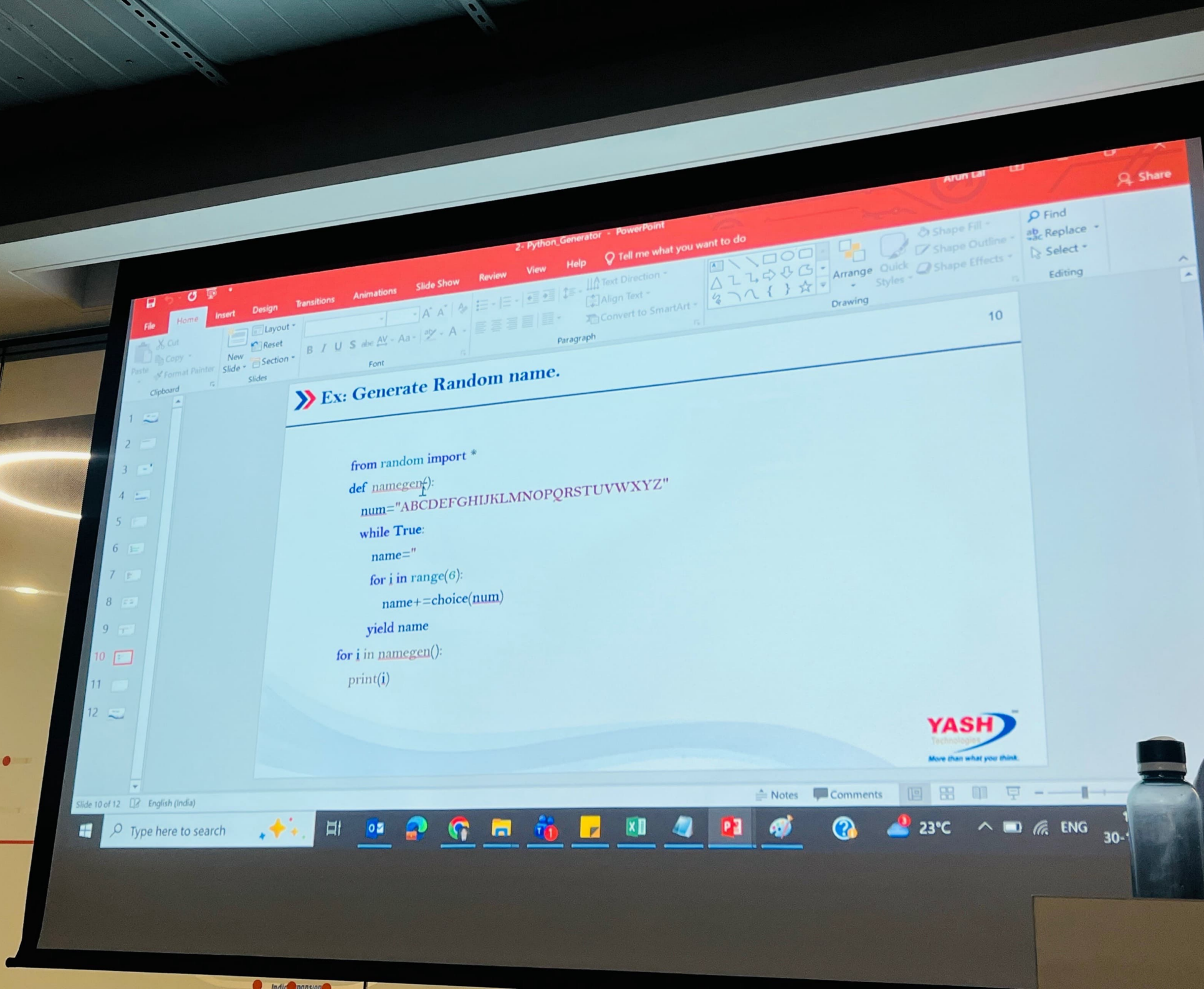
Now: Generate Fibonacci Number- 0 1 1 2 3 5 8 13.. Next number is the sum of previous 2 number.

Adv: We are not going to store anywhere, just generate and use.

Ex:

```
def fib():
    a,b=0,1
    while True:
        yield a
        a,b=b,a+b
for f in fib():
    if f<100:
        print(f)
    else:
        break;
```

YASH Technologies
More than what you think.


2- Python_Generator - PowerPoint
Tell me what you want to do
Share
10
Ex: Generate Random name.
from random import *
def namegen():
num="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
while True:
name=''
for i in range(6):
name+=choice(num)
yield name
for i in namegen():
print(i)
YASH
Technologies
Slide 10 of 12
English (India)
Notes
Comments
Type here to search
23°C
ENG