Day 49

Generators in Python:

What Is a Generator?

A generator is a special kind of function that:

- Produces values one at a time
- Remembers its state between values
- Uses yield instead of return

Think of a generator as a lazy factory: it creates values *only when asked*.

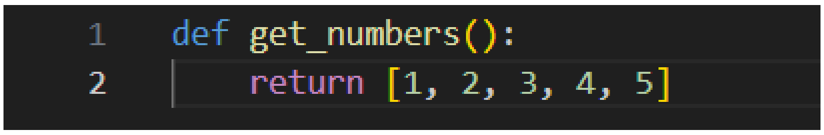
Problem with normal functions:

A normal function returns everything at once.

This:

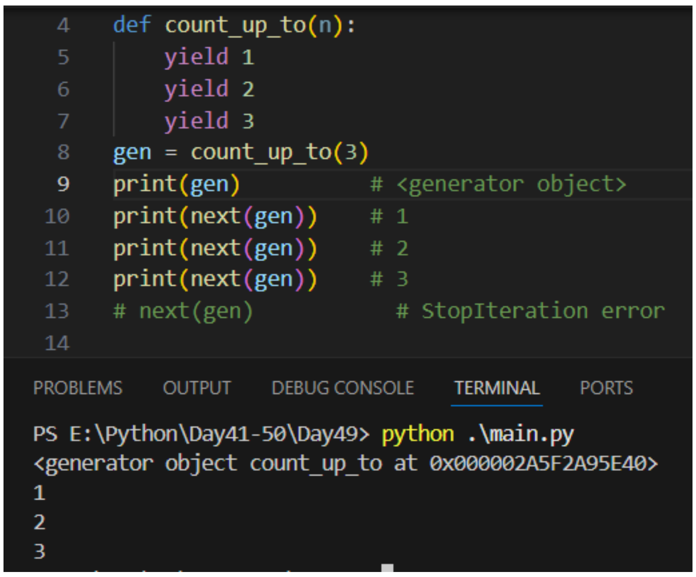
- Uses more memory
- Computes all values immediately

Example:



```python
def get_numbers():
    return [1, 2, 3, 4, 5]
```

Example: using generator.



```python
def count_up_to(n):
    yield 1
    yield 2
    yield 3
gen = count_up_to(3)
print(gen)              # <generator object>
print(next(gen))        # 1
print(next(gen))        # 2
print(next(gen))        # 3
# next(gen)               # StopIteration error
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
<generator object count_up_to at 0x000002A5F2A95E40>
1
2
3
```

How yield Works

- yield pauses the function
- Saves all local variables
- Resumes from where it stopped

yield vs return

Feature	**yield**	**return**
Returns value	Yes	Yes
Pauses function	Yes	No
Remembers state	Yes	No
Multiple values	Yes	No

Summary:

- Generators use yield
- Produce values lazily
- Save memory
- Very Pythonic and powerful

Function Caching in Python:

What Is Function Caching?

Remembering the result of a function call so the next time the same inputs are used, Python can return the saved result instead of recomputing it.

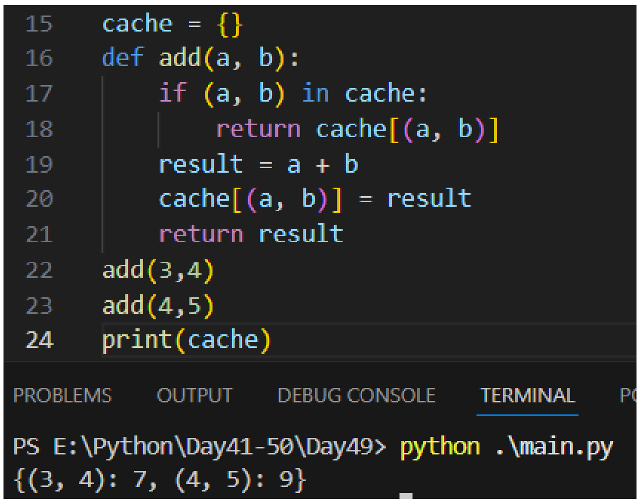
This makes programs faster and more efficient.

Why Do We Need Caching?

Some functions:

- Are slow
- Do the same calculation repeatedly
- Return the same output for the same input

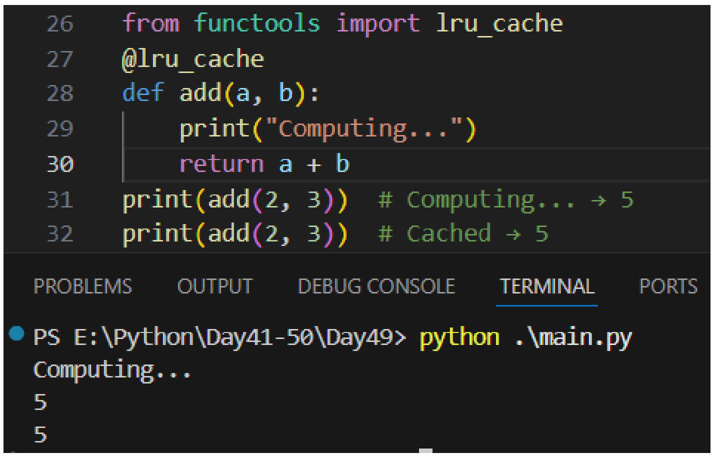
Example: manual caching.



```python
cache = {}
def add(a, b):
    if (a, b) in cache:
        return cache[(a, b)]
    result = a + b
    cache[(a, b)] = result
    return result
add(3,4)
add(4,5)
print(cache)
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
{(3, 4): 7, (4, 5): 9}
```

Example: python built-in solution for caching.



```python
from functools import lru_cache
@lru_cache
def add(a, b):
    print("Computing...")
    return a + b
print(add(2, 3))  # Computing... → 5
print(add(2, 3))  # Cached → 5
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
Computing...
5
5
```

What Is lru_cache?

LRU = Least Recently Used

- Stores function results in memory
- Removes old values when cache is full
- Uses function arguments as cache keys

Caching vs Memoization

- **Memoization**: caching function results
- **Caching**: general term

In Python, they’re often used interchangeably.

Summary

- Function caching improves performance
- lru_cache is easy and powerful
- Use for deterministic, expensive functions
- Monitor cache size and memory

Regular Expressions in Python:

What Are Regular Expressions?

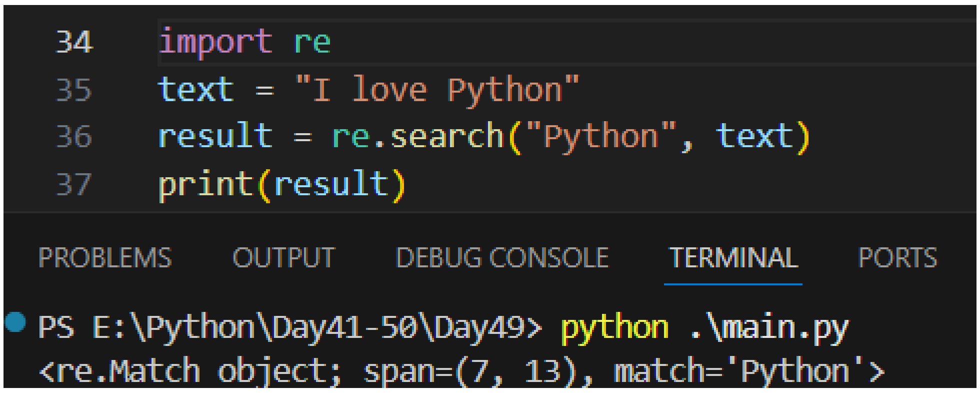
Regular Expressions (regex) are patterns used to search, match, and manipulate text.

Think of regex as: *a powerful "find and replace" language for strings*

Python's Regex Module: re

import re

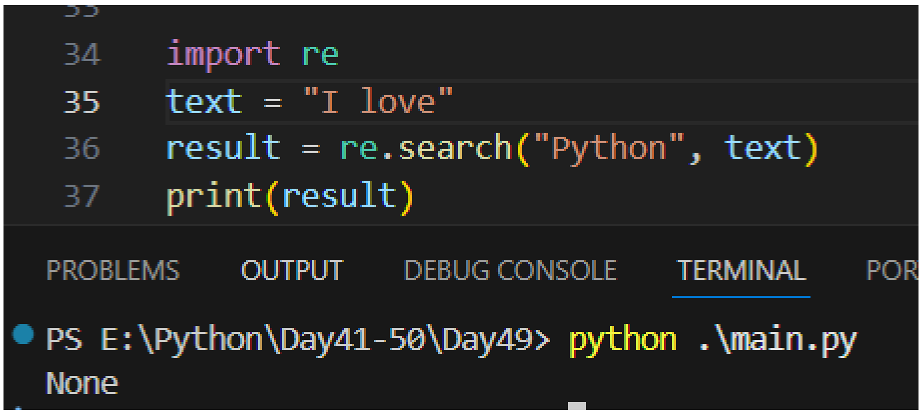
Example: regex search.



```python
import re
text = "I love Python"
result = re.search("Python", text)
print(result)
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
<re.Match object; span=(7, 13), match='Python'>
```

Example: returns None if not found.



```python
import re
text = "I love"
result = re.search("Python", text)
print(result)
```

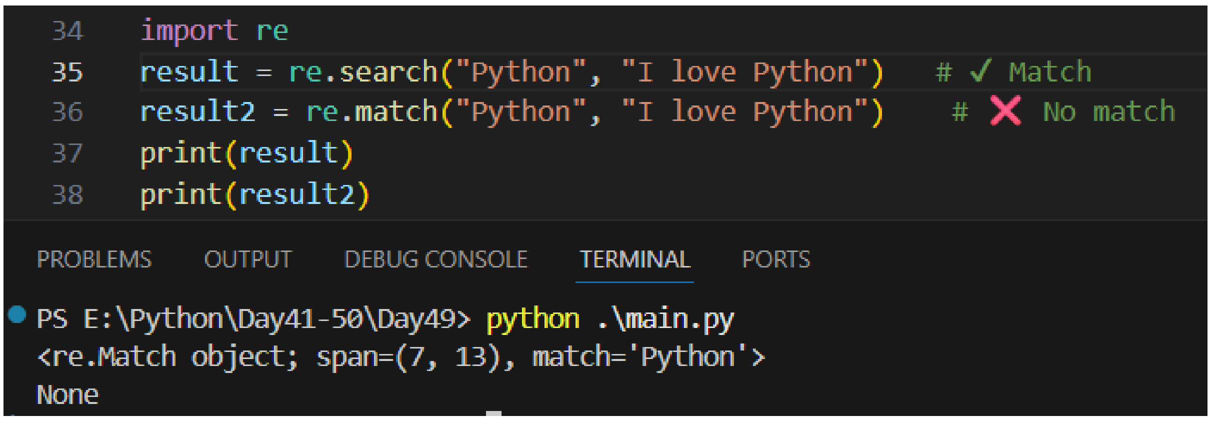
```
PS E:\Python\Day41-50\Day49> python .\main.py
None
```

Common Regex Functions

Function	Purpose
re.search()	Find first match
re.match()	Match from start only
re.findall()	Find all matches
re.finditer()	Iterator of matches
re.sub()	Replace matches

Function	Purpose
re.split()	Split string

Example: re.search() vs re.match()

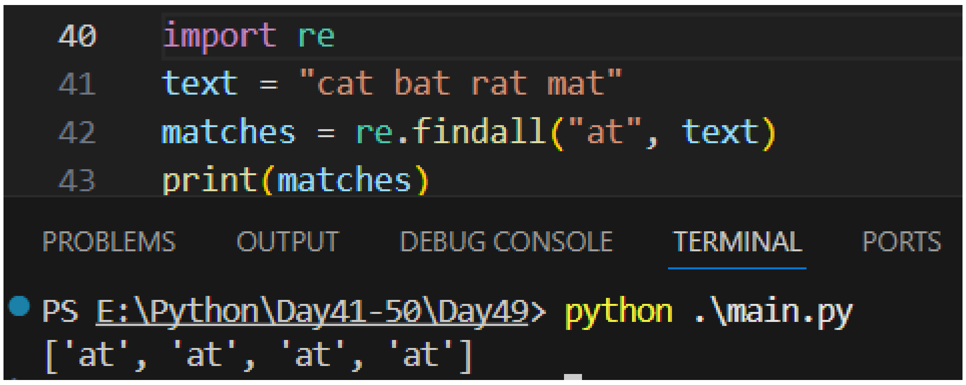


```
import re
result = re.search("Python", "I love Python")   # ✓ Match
result2 = re.match("Python", "I love Python")    # ❌ No match
print(result)
print(result2)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\Python\Day41-50\Day49> python .\main.py
<re.Match object; span=(7, 13), match='Python'>
None
```

Example: finding all matches.



```
import re
text = "cat bat rat mat"
matches = re.findall("at", text)
print(matches)
```

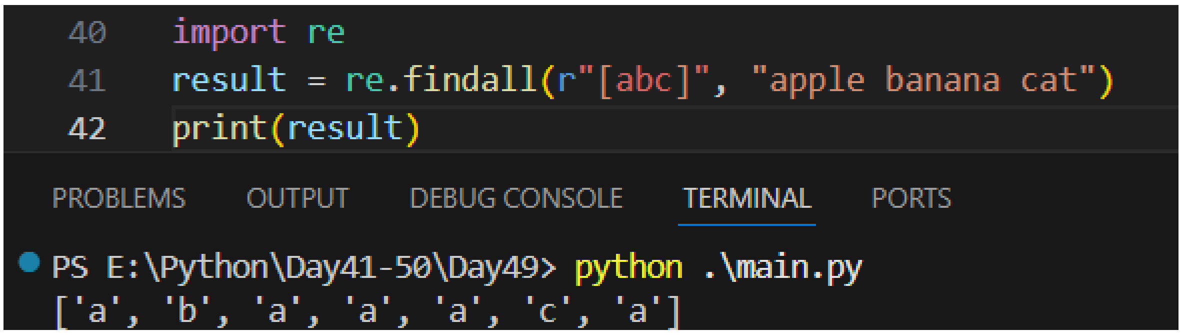
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\Python\Day41-50\Day49> python .\main.py
['at', 'at', 'at', 'at']
```

Character Classes

Pattern	Meaning
.	Any character
\d	Digit (0–9)
\w	Word (letters, digits, _)
\s	Whitespace
\D, \W, \S	Negations

Example:



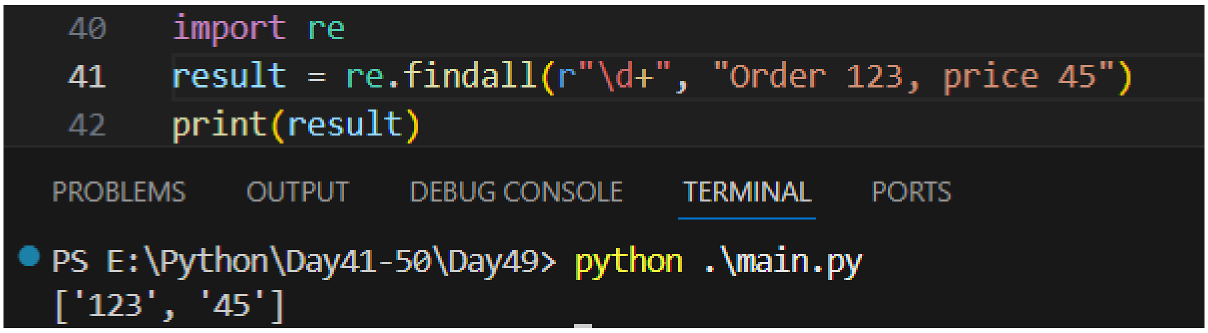
```
import re
result = re.findall(r"[abc]", "apple banana cat")
print(result)
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
['a', 'b', 'a', 'a', 'a', 'c', 'a']
```

Quantifiers (How Many?)

Symbol	Meaning
*	0 or more
+	1 or more
?	0 or 1
{n}	Exactly n
{n,}	At least n
{n,m}	Between n and m

Example:



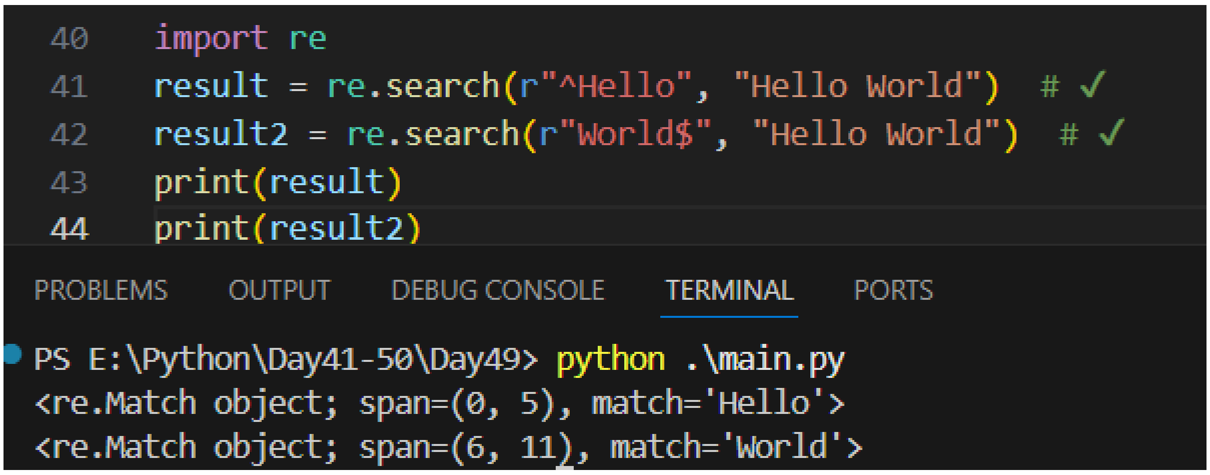
```
import re
result = re.findall(r"\d+", "Order 123, price 45")
print(result)
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
['123', '45']
```

Anchors (Position Matters)

Symbol	Meaning
^	Start of string
$	End of string

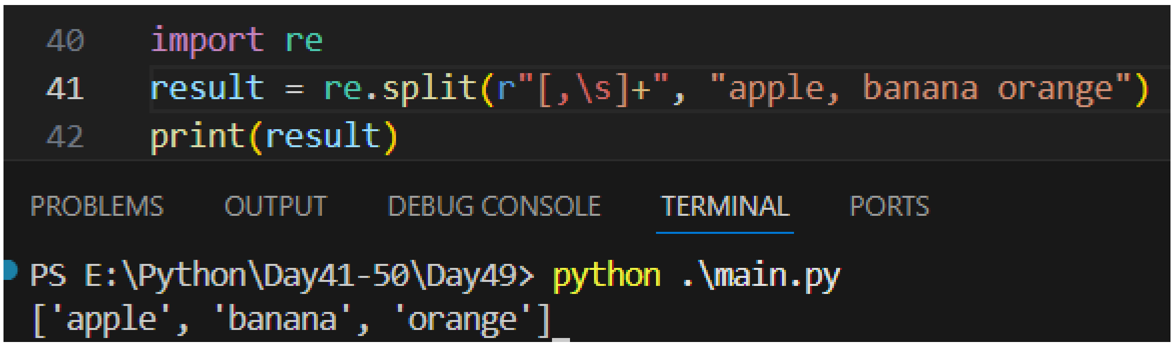
Example:



```python
import re
result = re.search(r"^Hello", "Hello World")  # ✓
result2 = re.search(r"World$", "Hello World")  # ✓
print(result)
print(result2)
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
<re.Match object; span=(0, 5), match='Hello'>
<re.Match object; span=(6, 11), match='World'>
```

Example: splitting text



```python
import re
result = re.split(r"[,\s]+", "apple, banana orange")
print(result)
```

```
PS E:\Python\Day41-50\Day49> python .\main.py
['apple', 'banana', 'orange']
```

Summary

- Regex is powerful but needs practice
- Python’s re module is feature-rich
- Start small, test often
- Readability matters more than cleverness

--The End--