IT CAREER HUB

Урок 29.
Генераторы

Генератор 2
Функция с yield 3
Ключевое слово yield 4
Генераторные функции с параметрами 6
Использование генератора в for 7
Задания для закрепления 1 8
Бесконечные генераторы 9
Метод close 11
Конструкция yield from 12
Задания для закрепления 2 14
Ответы на задания 15
Практическая работа 16

IT CAREER HUB

Генератор

 **Генератор** – это специальный вид итератора, который **создаёт элементы по запросу, вместо того чтобы загружать всю последовательность в память.**

Все **генераторы** являются **итераторами**, но не все **итераторы** являются **генераторами**.

Генераторы создаются с помощью:

- **Генераторных функций** – функций, использующих ключевое слово `yield`.
- **Генераторных выражений** – компактного способа создания генераторов в круглых скобках (...).

Основные особенности генераторов

- Ленивые вычисления – создают элементы только при запросе.
- Не занимают много памяти – не хранят всю последовательность в памяти.
- Сохраняют состояние – продолжают выполнение **с того места, где остановились**.
- Используют `yield` вместо `return` – это делает их похожими на обычные функции, но с возможностью приостановки выполнения.

Генераторы удобны для **обработки больших файлов**, **создания бесконечных последовательностей** и **работы с потоками данных**.

IT CAREER HUB

Функция с yield

Функция с yield — это функция, которая возвращает объект генератора.

При вызове `next()` у такого генератора выполнение функции **приостанавливается на** `yield`, сохраняя текущее состояние.
При следующем вызове выполнение **продолжается с того же места**, а не начинается заново, как в обычных функциях.

IT CAREER HUB

Ключевое слово yield

yield – это ключевое слово, которое используется для приостановки выполнения функции и возврата значения без завершения её работы (в отличие от return).

Как работает yield?

1. **При первом вызове** `next()` выполнение функции начинается **с самого начала** и продолжается **до первого** yield. Значение после `yield` **возвращается**.
2. **При следующем вызове** `next()` выполнение **продолжается с того места, где оно было остановлено** (а не начинается заново).
3. **Функция работает, пока не завершит выполнение**, после чего генератор вызывает `StopIteration`.

Синтаксис

```python
def generator():
    yield value
```

- `def generator():` – объявление функции, которая вернет генератор.
- `yield` – ключевое слово, приостанавливающее выполнение функции и возвращающее значение.
- `value` – результат, который будет передан при вызове `next()`.

Пример генератора с yield

```python
def generate_values():
    print("Начало работы")
    yield 1  # Приостанавливаем выполнение и возвращаем 1
    print("Продолжение работы")
```

```
    yield 2  # Приостанавливаем выполнение и возвращаем 2
    print("Завершение работы")

gen = generate_values()  # Создаём генератор, но код внутри функции пока не выполняется

print(next(gen))  # Начало работы → 1
print(next(gen))  # Продолжение работы → 2
print(next(gen))  # Завершение работы → StopIteration, так как нет третьего yield
```

IT CAREER HUB

Генераторные функции с параметрами

Функция-генератор может принимать аргументы, которые передаются однократно при создании генератора и используются для генерации значений.

Синтаксис

```python
def generator(param1, param2):
    yield value
```

- `param1`, `param2` – аргументы, передаваемые при создании генератора.

Пример: генератор чисел "n" раз

```python
def count_up_to(n):
    count = 1
    while count <= n:
        yield count  # Возвращаем текущее значение и "замораживаем" выполнение
        count += 1   # После следующего вызова next() продолжится отсюда

gen = count_up_to(5)

print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
# print(next(gen))  # StopIteration
```

IT CAREER HUB

Использование генератора в for

Генератор можно **перебирать в цикле** `for`, как и любой итератор. В этом случае `for` **автоматически вызывает** `next()`, пока генератор **не исчерпает все значения**.

Пример: генератор и for

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

gen = count_up_to(5)

for number in gen:
    print(number)
```

IT CAREER HUB

Задания для закрепления 1

1. Какой результат выдаст следующий код?

```python
def countdown(n):
    while n > 0:
        yield n
        n -= 1

gen = countdown(3)

print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

a. 3, 2, 1, 1
b. 3, 2, 1, 0
c. 3, 2, 1, 3
d. 3, 2, 1, StopIteration

Посмотреть ответы

2. Какие утверждения о yield верны?

a. yield приостанавливает выполнение функции и запоминает состояние
b. После yield выполнение начинается заново
c. Генератор можно использовать только один раз
d. yield можно использовать только внутри функции

Посмотреть ответы

IT CAREER HUB

Бесконечные генераторы

Бесконечный генератор — это генератор, который никогда не завершает выполнение и может неограниченно выдавать новые значения.

Такие генераторы **особенно полезны**, когда нужно **создавать поток данных** без заранее известного количества элементов.

Пример 1: генератор бесконечной последовательности

```python
def infinite_counter():
    count = 1
    while True:  # Бесконечный цикл
        yield count
        count += 1


gen = infinite_counter()

for number in gen:
    if number > 5:  # Условие выхода
        break  # Принудительная остановка генератора
    print(number)

print()

# Вызов next 10 раз
for _ in range(10):
    print(next(gen))
```

Особенности

- Генератор **не завершится сам по себе**, так как while True работает бесконечно.
- Нужно **вручную остановить выполнение** (например break в for) или вызвать next() вручную.

Бесконечные генераторы могут применяться **для циклического распределения ресурсов**, например, автоматического назначения задач на сотрудников в команде.

Пример 2: генератор распределения задач

```python
def task_assigner(employees):
    while True:
        for employee in employees:
            yield employee


# Список сотрудников
team = ["Alice", "Bob", "Charlie"]


# Создаём генератор распределения задач
assigner = task_assigner(team)


# Назначаем 7 задач
for i in range(7):
    print(f"Task {i + 1} assigned to: {next(assigner)}")
```

Метод close

Метод `close()` используется для принудительного завершения работы генератора.
Когда вызывается `close()`, генератор **завершается** и при следующем вызове `next()` вызывает `StopIteration`.

Синтаксис

```python
generator.close()
```

- `generator` – объект генератора.
- `close()` останавливает выполнение генератора, не дожидаясь его естественного завершения.

Метод `close()` может использоваться для **прерывания работы генератора в определённой ситуации**, например, когда получено нужное значение.

Пример: закрытие генератора при достижении условия

```python
def sensor_data(data):
    for value in data:
        yield value

numbers = [10, 20, 30, 40, 50]
gen = sensor_data(numbers)

for element in gen:
    print("Получено значение:", element)

    # Завершаем генератор, когда получено нужное значение
    if element >= 30:
        print("Значение найдено, закрываем генератор.")
        gen.close()

# next(gen)  # Вызовет ошибку
```

IT CAREER HUB

Конструкция yield from

yield from – это конструкция в Python, которая делегирует управление другому генератору или итерируемому объекту.
Она упрощает вложенные генераторы, позволяя не вызывать yield вручную в цикле.

Синтаксис

```python
yield from iterable
```

- `iterable` – любой итерируемый объект (список, кортеж, строка, другой генератор).
- `yield from` **поочерёдно передаёт элементы** от указанного источника.

Пример 1: итерируемый объект

```python
def letters():
    yield from "ABC"

gen = letters()

print(next(gen))  # A
print(next(gen))  # B
print(next(gen))  # C
```

Пример 2: Вспомогательный генератор

Python

```python
# Умеет работать с одним объектом
def process_values(data):
    for value in data:
        yield value * 2

# Собирает несколько объектов
def main_generator(*sequences):
    # Делегирует обработку каждого объекта вспомогательному генератору
    for seq in sequences:
        yield from process_values(seq)

# Используем несколько источников данных
data1 = [1, 2, 3]
data2 = [10, 15]

for result in main_generator(data1, data2):
    print(result)
```

IT CAREER HUB

Задания для закрепления 2

Какие утверждения о yield from верны?

a. Позволяет делегировать управление другому генератору
b. Используется только с бесконечными генераторами
c. Может передавать элементы из списка
d. Упрощает работу с вложенными генераторами

Посмотреть ответы

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Результат выполнения кода	Ответ: d
2. Утверждения о `yield`	Ответ: a, c, d
Задания на закрепление 2	Вернуться к заданиям
1. Утверждения о `yield from`	Ответ: a, c, d

IT CAREER HUB

Практическая работа

1. Фильтр чисел

Создайте генератор, который принимает **список чисел** и выдаёт **только числа, кратные 5**.

Данные:

```python
numbers = [12, 15, 33, 40, 55, 62, 75, 83, 90]
```

Пример вывода:

```python
15
40
55
75
90
```

Решение:

Python

```python
def filter_by_five(numbers):
    for num in numbers:
        if num % 5 == 0:
            yield num

numbers = [12, 15, 33, 40, 55, 62, 75, 83, 90]
gen = filter_by_five(numbers)

for num in gen:
    print(num)
```

IT CAREER HUB

2. Квадраты чисел

Создайте **генератор**, который принимает число **n** и **генерирует квадраты чисел** от **1 до n** включительно.

Данные:

```python
n = 10
```

Пример вывода:

```python
1
4
9
16
25
36
49
64
81
100
```

Решение:

```python
def square_numbers(n):
    for num in range(1, n + 1):
        yield num ** 2

# Используем генератор
gen = square_numbers(10)

for square in gen:

    print(square)
```