IT CAREER HUB

Урок 28.
Итераторы и генераторные выражения

IT CAREER HUB

Итератор

Итерируемый объект — это объект, который может состоять из множества элементов и предоставляет их по одному.

Итератор – это объект, который позволяет поочерёдно получать элементы коллекции без необходимости загружать их все в память сразу.

Итератор содержит:

- Ссылку на итерируемый объект, из которого он извлекает данные.
- Текущую позицию – он запоминает, где остановился при последнем вызове.
- Логику получения следующего элемента – сообщает как извлечь следующий элемент из итерируемого объекта.

Итератор не изменяет коллекцию, на которую ссылается, а просто управляет процессом последовательного извлечения данных.

Итераторы используются при работе с циклами `for`, ленивыми вычислениями, чтением файлов построчно и в других ситуациях, когда обработка данных должна выполняться эффективно и без лишнего расхода памяти.

IT CAREER HUB

Методы iter и next

Итерация в Python основана на магических методах (или dunder-методах - сокращение от double underscore) – это специальные методы, которые вызываются автоматически при использовании встроенных механизмов языка, например цикла for.

Их названия начинаются и заканчиваются двойным подчёркиванием.

Для работы с итерацией используются два магических метода:

1. `__iter__()` – создаёт объект итератора из итерируемого объекта.
2. `__next__()` – выдаёт следующий элемент итерируемого объекта.

Пример

```python
numbers = [10, 20, 30]  # Обычный список

iterator = numbers.__iter__()  # Получаем итератор


print(iterator)  # Итератор для списка

print(list(iterator))  # Преобразование в список


print(list(iterator))  # Второй раз список будет пустой
```

Особенности

- `numbers.__iter__()` создаёт итератор, связанный со списком.
- Второй раз список будет пустой, так как итератор "потрачен"

IT CAREER HUB

Пример: последовательное получение элементов

```python
numbers = [10, 20, 30]

iterator = numbers.__iter__()  # Создаём итератор


print(iterator.__next__())  # Получаем нулевой элемент

print(iterator.__next__())  # Получаем первый элемент

print("-" * 30)  # Можно прерваться на другие действия

print(iterator.__next__())  # Получаем второй элемент
```

Особенности

- `iterator.__next__()` возвращает следующий элемент.
- При каждом вызове итератор запоминает текущую позицию.

IT CAREER HUB

Аналоги магических методов

Python предоставляет одноименные встроенные функции, которые вызывают магические методы "под капотом":

1. `iter(obj)` – вызывает `obj.__iter__()`, создавая итератор из итерируемого объекта.
2. `next(iterator)` – вызывает `iterator.__next__()`, возвращая следующий элемент.

Пример использования встроенных функций

```python
numbers = [1, 2, 3]  # Итерируемый объект

iterator = iter(numbers)  # Вызывает numbers.__iter__()


print(next(iterator))  # Вызывает iterator.__next__()

print(next(iterator))  # Вызывает iterator.__next__()

print(next(iterator))  # Вызывает iterator.__next__()
```

IT CAREER HUB

Оценка потребления памяти

Функция `sys.getsizeof()` позволяет измерять, сколько памяти занимает объект в байтах. Это полезно для сравнения объектов.

Пример: сравнение списка и итератора

```python
import sys


# Список из 1 000 000 чисел
numbers_list = [x for x in range(1_000_000)]
print("Размер списка:", sys.getsizeof(numbers_list), "байт")


# Итератор, из списка
numbers_iterator = numbers_list.__iter__()
print("Размер итератора:", sys.getsizeof(numbers_iterator), "байт")
```

IT CAREER HUB

Ошибка StopIteration

Когда итератор исчерпал все доступные элементы, и метод `__next__()` не может вернуть следующий, возникает ошибка `StopIteration`.

Пример

```python
numbers = [1, 2, 3]

iterator = iter(numbers)  # Создаём итератор



print(next(iterator))

print(next(iterator))

print(next(iterator))  # Последний элемент

print(next(iterator))  # Ошибка StopIteration
```

Как избежать StopIteration?

Функция `next()` поддерживает второй аргумент, который возвращается вместо ошибки `StopIteration`, если элементы в итераторе закончились.

IT CAREER HUB

Пример

Python

```python
numbers = [1, 2, 3]

iterator = iter(numbers)



print(next(iterator))

print(next(iterator))

print(next(iterator))

print(next(iterator, None))  # None вместо StopIteration

print(next(iterator, 0))  # 0 вместо StopIteration
```

Как работает цикл for

Цикл `for` в Python используется для перебора итерируемых объектов (списки, строки, итераторы и т.д.).

Внутри себя `for` автоматически использует итератор, вызывая методы `__iter__()` и `__next__()`.

Как for обрабатывает итерацию шаг за шагом

При выполнении цикла `for element in iterable:` Python выполняет следующие действия:

1. Вызывает `iter(iterable)`, чтобы получить итератор.
2. Вызывает `next(iterator)`, чтобы получить следующий элемент.
3. Передаёт полученное значение в переменную цикла (`element`).
4. Повторяет шаги 2–3, пока `next()` не вызовет `StopIteration`.
5. Когда `StopIteration` возникает, цикл автоматически завершает работу без ошибки.

Пример работы цикла for

```python
numbers = [10, 20, 30]  # Итерируемый объект

# Цикл for по итерируемому объекту

for num in numbers:

    print(num)

print()

iterator = iter(numbers)  # Итератор

# Цикл for по итератору

for num in iterator:

print(num)
```

Как Python выполняет этот цикл внутри:

```python
numbers = [10, 20, 30]  # Итерируемый объект

iterator = iter(numbers)  # Создание итератора


while True:

    try:

        num = next(iterator)  # Получаем следующий элемент

        print(num)

    except StopIteration:

        break  # Завершаем цикл при окончании элементов
```

IT CAREER HUB

Итератор и итерируемый объект

Все итераторы являются итерируемыми объектами, но не все итерируемые объекты являются итераторами.

Сравнение итератора и итерируемого объекта

Характер истика	**Итерируемый объект (iterable)**	**Итератор (iterator)**
Методы	Реализует `__iter__()`, который возвращает итератор	Реализует `__iter__()` и `__next__()`
Создание	Списки, кортежи, множества, строки, словари	Создаётся с помощью `iter(iterable)`
Работа с next()	`next()` вызывать нельзя, вызовет ошибку	`next()` выдаёт следующий элемент, запоминая позицию
Хранение данных	Хранит **все элементы** сразу	Не хранит все элементы, а выдаёт их **по одному**
Перебор	Можно перебирать **много раз**	Можно перебирать **только один раз**
Конец данных	Не зависит от `StopIteration`	При завершении элементов вызывает `StopIteration`

Особенности итераторов

- Экономия памяти: итератор не хранит всю коллекцию в памяти, а запоминает ссылку на источник данных и текущую позицию. Это позволяет работать с большими данными без перегрузки памяти.
- Последовательный доступ: доступ к элементам происходит по порядку, перескочить к произвольному элементу нельзя.

- Неизменяемость: итератор не поддерживает изменение данных, он просто возвращает элементы.
- Однократное использование: Итератор можно использовать только один раз – после полного прохода по элементам он становится "пустым", и для нового перебора нужно создать новый итератор.

IT CAREER HUB

Задания для закрепления 1

1. **Какие утверждения о итераторах верны?**
   a. Итератор можно использовать многократно
   b. Итератор хранит все элементы в памяти
   c. Итератор использует методы `__iter__()` и `__next__()`
   d. Итератор можно создать с помощью `iter(iterable)`

2. Какое из утверждений о StopIteration верное?

a. `StopIteration` вызывается, когда итератор не может вернуть новый элемент
b. `StopIteration` вызывается при каждом вызове `next()`
c. `StopIteration` можно перехватить с помощью `try-except`
d. `StopIteration` никогда не вызывается автоматически

3. Какой результат будет у следующего кода?

```python
numbers = [5, 10, 15]

iterator = iter(numbers)

for num in iterator:

    print(num)

for num in iterator:

    print(num)
```

a. `5, 10, 15, 5, 10, 15`
b. `5, 10, 15`
c. `5, 10`
d. `5, 10, 15, Ошибка StopIteration`

Посмотреть ответы

IT CAREER HUB

Модуль itertools

Модуль itertools предоставляет набор инструментов для работы с итераторами, позволяя создавать эффективные итерируемые последовательности, объединять, фильтровать и изменять потоки данных без лишнего расхода памяти.

Примеры использования itertools

1. Бесконечная последовательность чисел: count()

```python
import itertools


counter = itertools.count(start=1, step=10)


print(next(counter))
print(next(counter))
print(next(counter))
print(next(counter))
print(next(counter))
```

2. Бесконечное повторение элементов: cycle()

```python
import itertools


cycler = itertools.cycle(["A", "B", "C"])


print(next(cycler))
print(next(cycler))
print(next(cycler))
print(next(cycler))  # A (повторяется)
print(next(cycler))  # B (повторяется)
```

3. Объединение итерируемых объектов: chain()

```python
import itertools


merged = itertools.chain([1, 2, 3], [100, 200, 300])


print(list(merged))
```

4. Декартово произведение: product()

```python
import itertools


pairs = itertools.product([1, 2, 3], ["A", "B", "C"])

print(pairs)

print(list(pairs))
```

5. Перестановки элементов: permutations()

Функция itertools.permutations() создаёт все возможные упорядоченные перестановки элементов. Можно указать длину перестановки (r), или использовать всю последовательность по умолчанию.

```python
import itertools


letters = ["A", "B", "C"]


# Все возможные перестановки

perms = itertools.permutations(letters)

print(list(perms))


# Все возможные перестановки длины 2

perms = itertools.permutations(letters, 2)

print(list(perms))
```

IT CAREER HUB

Генераторное выражение

Генераторное выражение – это способ создания последовательности значений без предварительного вычисления всех элементов. Вместо хранения данных в памяти оно вычисляет каждый следующий элемент только при необходимости.

Генераторное выражение возвращает объект генератора, который является итератором и поддерживает метод `__next__()`, позволяя получать элементы по одному.

Оно похоже на списковое включение, но использует круглые скобки `( ... )` вместо квадратных `[ ... ]`.

Генераторное выражение и списковое включение

Характеристика	**Списковое включение [...]**	**Генераторное выражение (...)**
Создаваемый объект	Список (`list`)	Генератор (`generator`)
Создание элементов	Все элементы вычисляются сразу и хранятся в памяти	Каждый элемент создаётся только в момент запроса
Использование памяти	Требует хранения всей последовательности	Экономит память, так как не хранит данные
Доступ к элементам	Можно обращаться по индексу (`list[0]`)	Доступ возможен только через `next()` или в for
Подходит для	Малых и средних последовательностей	Больших последовательностей и потоковой обработки данных

IT CAREER HUB

Синтаксис

```python
generator = (expression for item in iterable)
```

Пример 1: Создание генераторного выражения

```python
# Генерирует квадраты чисел от 0 до 4

squares = (x ** 2 for x in range(5))


print(squares)  # Объект генератора


print(next(squares))

print(next(squares))

print(next(squares))

print(next(squares))

print(next(squares))
```

Особенности

- Возвращает генератор, а не сразу список значений.
- `next()` вычисляет следующий элемент только при запросе.

Пример 2: Генераторное выражение в for

```python
squares = (x ** 2 for x in range(5))


for num in squares:

    print(num)
```

Особенности

- Генератор автоматически перебирается в `for`, без явного вызова `next()`.
- После прохода по всем элементам он исчерпывается и повторно использовать его нельзя.

Пример 3: Потребление памяти

```python
import sys


list_comp = [x**2 for x in range(10**6)]  # Списковое включение

gen_expr = (x**2 for x in range(10**6))  # Генераторное выражение


print("Размер списка:", sys.getsizeof(list_comp), "байт")

print("Размер генератора:", sys.getsizeof(gen_expr), "байт")
```

Особенности

- Список занимает много памяти, так как хранит все элементы сразу.
- Генератор почти не занимает памяти, так как создаёт элементы по мере запроса.

Использование генераторных выражений

Генераторные выражения можно передавать в аргументы функций, где ожидается итерируемый объект. Это особенно полезно при обработке больших последовательностей.

Пример: Генераторное выражение в any() и all()

```python
words = ["apple", "Banana", "cherry", "Apricot"]


print(any(word[0].isupper() for word in words))  # Есть слово с заглавной буквы

print(all(len(word) > 3 for word in words))  # Все слова длиннее 3 букв
```

IT CAREER HUB

Задания для закрепления 2

1. **Какой результат выдаст следующий код?**

```python
import itertools


letters = ["A", "B", "C"]

cycled = itertools.cycle(letters)


print(next(cycled))

print(next(cycled))

print(next(cycled))

print(next(cycled))
```

a. A, B, C, A
b. A, B, C, C
c. A, B, C, Ошибка StopIteration

2. Какие утверждения о генераторных выражениях верны?

a. Генераторное выражение использует круглые скобки
b. Генератор хранит все элементы в памяти
c. Генератор можно использовать только один раз
d. Генераторное выражение сразу создаёт список

Посмотреть ответы

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Утверждения об итераторах	Ответ: c, d
2. Утверждения о `StopIteration`	Ответ: a, c
3. Результат выполнения кода	Ответ: b
Задания на закрепление 2	Вернуться к заданиям
1. Результат выполнения кода	Ответ: a
2. Утверждения о генераторных выражениях	Ответ: a, c

IT CAREER HUB

Практическая работа

Генерация безопасных паролей

Программа должна сгенерировать все возможные пароли длиной 4 символа, соблюдая следующие условия:

Пароль должен содержать хотя бы одну заглавную букву, одну строчную букву и одну цифру.

Символы не должны повторяться.

Соседние символы не могут быть расположены подряд в таблице символов.

Все подходящие пароли записываются в файл `valid_passwords.txt`.

Данные:

```python
from string import ascii_lowercase, ascii_uppercase, digits
```

Пример данных в файле:

acA0

acA1

acA2

acA3

acA4

acA5

acA6

acA7

acA8

...

Решение:

```python
from itertools import permutations
from string import ascii_lowercase, ascii_uppercase, digits
all_chars = list(ascii_lowercase + ascii_uppercase + digits)

# Фильтр для проверки условий пароля
def is_valid(password):
    has_lower = any(c in ascii_lowercase for c in password)
    has_upper = any(c in ascii_uppercase for c in password)
    has_digit = any(c in digits for c in password)

    if not (has_lower and has_upper and has_digit):
        return False

    for i in range(len(password) - 1):
        if abs(ord(password[i]) - ord(password[i + 1])) == 1:
            return False

    return True
# Генерация всех перестановок
all_variants = permutations(all_chars, 4)
valid_passwords = ("".join(p) for p in all_variants if is_valid(p))

with open("valid_passwords.txt", "w") as file:
    file.write("\n".join(valid_passwords))
```