Методические указания

Урок 19.1. Генераторы и итераторы

Задачи урока:

- Генераторы и итераторы

0. Подготовка к уроку

До начала урока преподавателю необходимо:

1) Просмотреть, как ученики справились с домашним заданием
2) Прочитать методичку

1. Выражения-генераторы

Учитель: Сегодня мы продолжим изучение генераторов и познакомимся с выражениями-генераторами.
Выражение-генератор — это объект, выполняющий то же вычисление, что и списковое включение, но выдающий результат в итеративной форме. Синтаксис схож с синтаксисом спискового включения, но вместо квадратных скобок используются круглые.

```
numbers = [1, 2, 3, 4]
result = (x * x for x in numbers)
print(result)
```

Результат:

<generator object <genexpr> at 0x00000244AF762960>

Для того,чтобы перебрать все числа, мы можем воспользоваться циклом

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

или функцией next().

```
print(next(result))
print(next(result))
```

Результат:

1
4

Функция next() - возвращает следующий элемент итератора, вызвав его метод __next__().
Синтаксис:
next(iterator, default)

- iterator - объект итератора
- default - значение по умолчанию, которое будет возвращено вместо исключения StopIteration.

Если все значение перебраны:

- возбуждается исключение StopIteration, если значение по умолчанию default не задано
- возвращается значение default, если оно задано

Выражение-генератор можно использовать только один раз:

```
numbers = [1, 2, 3, 4]
result = (x * x for x in numbers)
print(next(result))
print(next(result))
print(next(result))
print(next(result))

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

Результатом примера выше будет вывод:
1
4
9
16

Как мы видит после вывода чисел в первый раз, второй перебор нам не выдал никакого результата.

Между списковыми включениями и выражениями-генераторами есть важные, но не очевидные различия. Со списковым включением Python создает реальный список с итоговыми данными. С выражением-генератором Python создает генератор, способный только производить данные по требованию. В некоторых приложениях это может сильно

улучшить производительность и эффективность использования памяти, например при чтении файла.

```
f = open('test.txt')
lines = (t.strip() for t in f)
comments = (t for t in lines if t[0] == '#')
for c in comments:
  print(c)
```

В примере выше выражение-генератор, извлекающее строки и удаляющее пропуски, обходится без чтения и хранения всего файла в памяти компьютера. То же касается и выражения, извлекающего комментарии. Вместо этого строки файла читаются по одной, когда программа начинает перебор в следующем цикле for. В ходе этого перебора строки файла создаются по запросу и соответственно фильтруются. Соответственно во время выполнения программы файл никогда не будет находиться в памяти целиком. А значит, такой способ работы, например с большими данными очень эффективен.

В отличие от списковых включений, выражение-генератор не создает объект, работающий как последовательность. Он не может индексироваться, и никакие обычные операции списков (например, append) с ним не работают. Но элементы, созданные выражением-генератором, можно преобразовать в список с помощью list():

```
comment_list = list(comments)
```

2. Генераторы

Учитель: Функции-генераторы — одна из самых интересных и мощных возможностей Python. Если функция использует ключевое слово yield, она определяет объект, называемый генератором. Цель применения генератора — создание значений, используемых в переборе. Пример:

```
def func(num):
  while num > 0:
    yield num
    num -= 1

for num in func(5):
  print(num)
```

Результат:

5

4

3

2

1

При вызове этой функции мы увидим, что ее код не выполняется

```
def func(num):
  while num > 0:
    yield num
    num -= 1

result = func(5)
print(result)
```

Вместо этого создается объект-генератор. Генератор выполняет функцию только тогда, когда вы начнете перебор по нему. Как вариант, мы также можем использовать функцию next():

```
def func(num):
  while num > 0:
    yield num
    num -= 1

result = func(5)
print(next(result))
print(next(result))
print(next(result))
```

Результат:

5

4

3

Учитель: При вызове next() функция-генератор выполняет команды до достижения yield. Команда yield возвращает результат, после чего выполнение функции приостанавливается до следующего вызова next(). При возобновлении выполнение продолжается с команды, следующей за yield.

next() — сокращение вызова метода __next__() для генератора. Можно написать так:

```
def func(num):
  while num > 0:
    yield num
    num -= 1

result = func(5)
print(result.__next__())
print(result.__next__())
```

```
print(result.__next__())
```

Обычно не используется функция next() для генератора напрямую, а используете команду for или другую операцию, использующую элементы:

```
for num in func(5):
  print(num)

# или например
a = sum(func(5))
```

Функция-генератор производит элементы, пока не вернет управление из-за достижения конца функции или выполнения команды return. При этом возюуждается исключение StopIteration, завершающее цикл for. Если функция-генератор возвращает значение, отличное от None, оно присоединяется к исключению StopIteration. Следующая функция-генератор использует как yield, так и return:

```
def func():
  yield 37
  return 42

result = func()
print(result)
print(next(result))
print(next(result))
```

Результат:

<generator object func at 0x000001A7EC1F2960>

37

Traceback (most recent call last):

File "C:\Users\python\main.py", line 8, in <module>

print(next(result))

StopIteration: 42

Как мы видим возвращаемое значение присоединяется к StopIteration.

С генераторами связан еще один интересный факт, проявляющийся при частичном потреблении функции-генератора. Создадим пример, в которым мы преждевременно выходим из цикла с помощью break

```
def func(num):
  while num > 0:
    yield num
    num -= 1
```

```
for num in func(5):
  if num == 2:
    break
  print(num)
```

Здесь цикл for прерывается вызовом break и связанный с ним генератор никогда не отрабатывает до завершения.

3. Решение задач

Задача 1

Числа Фибоначчи представляют последовательность, получаемую в результате сложения двух предыдущих элементов.

Начинается коллекция с чисел 1 и 1.

Она достаточно быстро растет, поэтому вычисление больших значений занимает немало времени.

Создайте функцию fib(n), генерирующую n чисел Фибоначчи с минимальными затратами ресурсов.

Для реализации этой функции потребуется обратиться к инструкции yield.

Решение

```
def fib(n):
    fib0 = 1
    yield fib0
    fib1 = 1
    yield fib1
    for i in range(n - 2):
        fib0, fib1 = fib1, fib0 + fib1
        yield fib1


for num in fib(112121):
      pass
print(num)
```

Дополнительно

Если на уроке остается время, то ученикам можно предложить начать прорешивать домашнее задание.

Домашняя работа

Задача 1

Данные об email-адресах учеников хранятся в словаре:

```
emails = {'mgu.edu': ['andrei_serov', 'alexander_pushkin', 'elena_belova',
'kirill_stepanov'],
      'gmail.com': ['alena.semyonova', 'ivan.polekhin', 'marina_abrabova'],
      'msu.edu': ['sergei.zharkov', 'julia_lyubimova', 'vitaliy.smirnoff'],
      'yandex.ru': ['ekaterina_ivanova', 'glebova_nastya'],
      'harvard.edu': ['john.doe', 'mark.zuckerberg', 'helen_hunt'],
      'mail.ru': ['roman.kolosov', 'ilya_gromov', 'masha.yashkina']}
```

Нужно дополнить код таким образом, чтобы он вывел все адреса в алфавитном порядке и в формате имя_пользователя@домен. При решении использовать ‘генератор’ словарей

Решение

```
print(*sorted({i + '@' + k for k, v in emails.items() for i in v}), sep =
'\n')
```