Методические указания

Урок 20.1. 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. Генераторы

Учитель: Сегодня мы с вами продолжим изучение темы генераторов в python. На прошлом занятии мы с уже разобрали работу с простейшим генератором и выяснили, что генераторы работают только один раз. Можно ли изменить данное поведение у генераторов?
Чтобы создать объект, допускающий повторные итерации, стоит определить его как класс и сделать метод __iter__() генератором:

```
class MyClass:
  def __init__(self, start):
    self.start = start

  def __iter__(self):
    n = self.start
    while n > 0:
      yield n
      n -= 1

obj = MyClass(5)
for n in obj.__iter__():
  print(n)

print(next(obj.__iter__()))
print(next(obj.__iter__()))
print(next(obj.__iter__()))
```

Результат

5

4

3

2

1

5

5

5

Такое решение работает, но мы видим, что в конце при вызове next() у нас выводится только первое число, все дело в том, что при каждом переборе метод __iter__() создает новый генератор.

Неотъемлемое свойство генераторов в том, что функция с yield никогда не выполняется сама по себе. Ею всегда должен управлять другой код с циклом for или явными вызовами функции next(). Это усложняет написание библиотечных функций с yield, ведь вызова функции-генератора недостаточно для обеспечения ее выполнения. Но можно решить эту проблему с помощью команды yield from:

```
def gen1(stop):
  n = 1
  while n <= stop:
    yield n
    n += 1
def gen2(start):
  n = start
  while n > 0:
    yield n
    n -= 1
def func(n):
  yield from gen1(n)
  yield from gen2(n)
```

Можно написать следующий код для управления перебором:

```
for n in func(5):
  print(n, end=' ')
```

Результат

1 2 3 4 5 5 4 3 2 1

yield from помогает избавится от необходимости управлять перебором вручную. Иначе вам пришлось бы записать func(n) так:

```
def func(n):
  for x in gen1(n):
    yield x
  for x in gen2(n):
```

```
    yield x
```

Рассмотрим еще один пример, с вложенными списками

```
numbers = [1, 2, [3, [4, 5], 6, 7], 8]

def func(n):
  for i in n:
    if isinstance(i, list):
      yield from func(i)
    else:
      yield i

for i in func(numbers):
  print(i, end=' ')
```

Результат:

1 2 3 4 5 6 7 8

Учитель: Внутри функции-генератора команда yield может использоваться как выражение в правой части оператора присваивания:

```
def func():
  while True:
    n = yield
    print(n)
```

Функция, так использующая yield, иногда называется расширенным генератором. Функция, использующая yield как выражение, остается генератором, но используется иначе. Вместо того чтобы создавать значения, она выполняется в ответ на значения, которые ей передаются:

```
r = func()
r.send(None)
r.send(1)
r.send(2)
r.send(6)
```

Результат:

1

2

6

Здесь исходный вызов r.send(None) нужен, чтобы генератор выполнил команды, ведущие к первому выражению yield. В этот момент он приостанавливается, ожидая отправки ему значения методом send() связанного объекта-генератора r. Значение, переданное send(), возвращается выражением yield в генераторе. При получении значения генератор выполняет команды до обнаружения следующего yield.

В примере выше функция выполняется бесконечно. Для закрытия генератора можно воспользоваться методом close():

```
r = func()
r.send(None)
r.send(1)
r.close()
r.send(2)
```

Результат

```
Traceback (most recent call last):
  File "C:\Users\python\main.py", line 11, in <module>
    r.send(2)
StopIteration
1
```

Операция close() выдает исключение GeneratorExit внутри генератора в текущей команде yield. Это приводит к незаметному завершению генератора. Если после закрытия генератору будут передаваться дополнительные значения, выдается исключение StopIteration.

Исключения могут выдаваться внутри генератора методом throw(ty [,val [,tb]]), где ty — тип исключения, val — аргумент исключения (или кортеж аргументов), а tb — необязательная трассировка:

```
r = func()
r.throw(RuntimeError, "Ошибка")
```

Результат:

```
Traceback (most recent call last):
  File "C:\Users\python\main.py", line 8, in <module>
    r.throw(RuntimeError,"Ошибка")
  File "C:\Users\python\main.py", line 1, in func
    def func():
```

RuntimeError: Ошибка

Выдаваемые так исключения распространяются от текущей выполняемой команды yield в генераторе. Он может решить перехватить исключение и обработать его по своему усмотрению.

Генератор продолжает существовать до явного закрытия или уничтожения. Поэтому можно воспользоваться им для создания задачи с долгим сроком жизни. Пример генератора, который получает байтовые фрагменты и собирает из них строки:

```
def func():
  data = bytearray()
  line = None
  linecount = 0
  while True:
    part = yield line
    linecount += part.count(b'\n')
    data.extend(part)
    if linecount > 0:
      index = data.index(b'\n')
      line = bytes(data[:index+1])
      data = data[index+1:]
      linecount -= 1
    else:
      line = None
```

Здесь генератор получает байтовые фрагменты, которые собираются в байтовый массив. Если массив содержит символ новой строки, строка извлекается и возвращается. В противном случае возвращается None.

```
r = func()
print(r.send(None))
print(r.send(b'hello'))
print(r.send(b'world\nit '))
print(r.send(b'works!'))
print(r.send(b'\n'))
```

Результат

None

None

b'helloworld\n'

None

b'it works!\n'

Данный код можно записать в виде класса:

```
class MyClass:
  def __init__(self):
    self.data = bytearray()
    self.linecount = 0
  def send(self, part):
    self.linecount += part.count(b'\n')
    self.data.extend(part)
    if self.linecount > 0:
      index = self.data.index(b'\n')
      line = bytes(self.data[:index+1])
      self.data = self.data[index+1:]
      self.linecount -= 1
      return line
    else:
      return None
```

Важно помнить, что если вы встречаете использование yield в контексте, не связанном с перебором, скорее всего, это связано с использованием send() или throw().

2. Решение задач

Задача 1

Написать генератор кортежа чисел от 5 до 25 включительно. Вывести весь кортеж, первое число, последнее число и последние 5 чисел

```
tuple1 = tuple([i for i in range(5, 26)])
print(tuple1)
print(tuple1[0])
print(tuple1[-1])
print(tuple1[-5:])
```

Задача 2

Написать программу, которая считываем числа с консоли через запятую и выводит только числа кратные 3 или 5 через запятую в одну строку

```
print([int(i) for i in input().split(',') if int(i) % 3 == 0 or int(i) % 5
== 0])
```

Дополнительно

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

Домашняя работа

Задача 1