PYTHON

Лекция 13

ГЕНЕРАТОРЫ

- Есть ключевое слово `yield`
- Его наличие в функции полностью меняет смысл происходящего
- Вызов такой функции начинает работать как конструктор специального объекта
- Код такой функции в момент вызова даже не исполняется

ПРИМЕР

```
 1 def f():
 2     print('hello')
 3     yield 123
 4
 5 v1 = f()
 6 print(v1)
 7 print(type(v1))
 8
 9 v2 = f()
10 print(v2)
11 print(type(v2))
```

МОДЕЛЬ ИСПОЛНЕНИЯ

- Вызов метода `__next__` над любым из полученных объектов запускает исполнение
- Код исполняется до `yield` или до выхода
- Если дошли до `yield`, то исполнение приостанавливается
- Выражение при `yield` вычисляется и передается в точку вызова `__next__` как его результат

ПРИМЕР

```
1 def f():
2     print('hello')
3     yield 123
4
5 v1 = f()
6 print(v1.__next())
7
8 v2 = f()
9 print(v2.__next__())
```

МОДЕЛЬ ИСПОЛНЕНИЯ

- Если дошли до конца - бросается исключение `StopIteration`
- Вместо `__next__` можно использовать встроенную функцию `next`
- Значение, передаваемое через `yield`, не обязано быть константой
- Оно может зависеть от параметров, от предыдущих действий

ПРИМЕР

```
 1 def fibo():
 2     prev, curr = 0, 1
 3     while True:
 4         yield prev
 5         prev, curr = curr, prev + curr
 6
 7 v1 = fibo()
 8 v2 = fibo()
 9 for _ in range(10):
10     print(next(v1))
11     print(next(v2))
12     print(next(v1))
```

ПРИМЕР

```
 1 def recur(prev=0, curr=1):
 2     while True:
 3         yield prev
 4         prev, curr = curr, prev + curr
 5
 6 v1 = recur(2, 3)
 7 v2 = fibo(-1, 2)
 8 for _ in range(10):
 9     print(next(v1))
10     print(next(v2))
11     print(next(v1))
```

ОБЩАЯ СХЕМА ИТЕРИРОВАНИЯ

- Вызовы `next` создают эффект итерирования
- Это в чем-то похоже на итерирование по списку или множеству
- Но есть важное отличие - для итерирования по списку или множеству нужно состояние
- Которое не является частью объекта `list`

ПРИМЕР

```
 1 def iter_list(data):
 2     index = 0
 3     while index < len(data): # специально без for
 4         yield data[index]
 5         index += 1
 6
 7 v = iter_list([1, 56, 10])
 8 print(next(v))
 9 print(next(v))
10 print(next(v))
11
12 # ............................
```

ПРИМЕР

```
1 # ............................
2 try:
3     next(v)
4 except StopIteration:
5     print('ok')
```

ТЕРМИНОЛОГИЯ

- Объект, получающийся при вызове функции с `yield` - генератор
- Объект, реализующий метод `__next__` - итератор
- Объект, реализующий метод `__iter__` - итерируемый (`iterable`)
- Подразумевается, что `__iter__` возвращает итератор

ФАКТЫ

- Генератор является итератором
- Генератор является итерируемым
- `__iter__` генератора возвращает ссылку на него самого
- Не всякий итерируемый является итератором

ФАКТЫ

- Список НЕ является итератором
- Список является итерируемым
- `__iter__` списка возвращает ссылку объекту служебного класса-итератора
- К методу `data.__iter__` можно обратиться через `iter(data)`

ПРИМЕР

```
1 v = iter([1, 56, 10])
2 print(next(v))
3 print(next(v))
4 print(next(v))
5
6 try:
7     next(v)
8 except StopIteration:
9     print('ok')
```

ЦИКЛ FOR

- Справа от `in` - итерируемый объект
- В частности - может быть генератор
- Можно создать wrapper-класс для списка
- В нем определить `__iter__`, возвращающий итератор с обходом в другом порядке
- Или сделать это через генератор

ПРИМЕР

```
1 def filter(data, f):
2     for v in data:
3         if f(v):
4             yield(v)
5
6 data = filter([1, 56, 10], lambda v: v % 2 == 0)
7 for v in data:
8     print(v)
```

ПРИМЕР

```
 1 def zip_with_next(data):
 2     it = iter(data)
 3     prev = next(it)
 4     for v in it:
 5         yield((prev, v))
 6         prev = v
 7
 8 data = zip_with_next([1, 56, 10, 12, 21])
 9 for v in data:
10     print(v)
```

ВСТРОЕННЫЕ ГЕНЕРАТОРЫ

- Много встроенных функций: `map, filter, enumerate, zip`
- Отдельный пакет: `itertools`
- `reduce` - в `functoools`
- `fold` - из коробки нет (?!)

ГЕНЕРАТОРНЫЕ ВЫРАЖЕНИЯ

- Есть списочные выражения
- Есть множественные и словарные
- А кортежных - нет
- Потому что круглые скобки отдали генераторным

ПРИМЕР

```
1 data = (v for v in range(10000000000000000000000000000))
2 data = (v for v in data if v % 7)
3 data = (v >> 2 for v in data)
4 for _ in range(10):
5     print(next(data))
```

ХОРОШИЙ СТИЛЬ

- Хороший стиль - выражать логику в терминах пребразований над последовательностями
- (numpy - отдельная история)
- Если последовательности небольшие - лучше использовать списочные выражения
- Если есть риск, что большие или бесконечные - генераторные выражения

ОСОБЫЙ СИНТАКСИС

- Есть функции, принимающие коллекции
- Примеры: `all, any, map, filter`
- Они могут принимать ленивые коллекции, в том числе генераторы
- Формально надо писать одну пару скобок для обозначения вызова
- И еще одну - для обозначения ленивого генератора
- Можно обойтись одной парой

ПРИМЕР

```
 1 def take(data, n):
 2     data_iter = iter(data)
 3     for _ in range(n):
 4         yield next(data_iter)
 5
 6 max_word_size = max(len(w) for w in input().split())
 7
 8 with open('data.txt') as f:
 9     numbered = enumerate(s.strip() for s in f)
10     selected = (index for index, w in numbered
11                 if len(w.split() > max_word_size))
12     head = take(selected, 10)
13     print(' '.join(str(v) for v in head))
```

РАБОТА С БОЛЬШИМИ ФАЙЛАМИ

- Можно открыть большой файл
- И использовать его в ленивом генераторе
- Или в генераторном выражении
- Но у файла как у источника есть особенность
- Его надо закрыть

ВАРИАНТЫ

- Вариант 1: уверены, что дочитаем до конца
- Ленивость нужна, чтобы не зачитывать все в память
- И чтобы оптимизировать операции над коллекцией
- Отличное решение - функция с `yield`

ПРИМЕР

```
1 def lazy_read(fname):
2     with f = open(fname):
3         while True:
4             s = f.readline()
5             if not s:
6                 return
7
8
9 # ..................................
```

ПРИМЕР

```
# ..................................


data = lazy_read(fname)
data = (v.strip() for v in data)
data = map(int, data)
data = (v * 2 for v in data % 10 == 4)
for e in data:
    print(e)

# и тут файл закрывается
```

ИСКЛЮЧЕНИЯ

- Механизм обработки ошибок
- Чисто технически исключение - Python-объект
- Может быть порожден "изнутри" Python (например, деление на 0)
- Может быть порожден программно - в случае обнаружения фатальной ошибки

ИСКЛЮЧЕНИЯ

- Факт создания объекта-исключения на исполнение кода не влияет
- Влияем - факт бросания исключения
- Программно бросается с помощью ключевого слова `raise`
- В момент бросания обычное исполнение кода прекращается
- Ищется обработчик исключения

ИСКЛЮЧЕНИЯ

- Обработчик исключения - конструкция `try/except`
- После `try` - блок кода для исполнения
- После `except` - блок для обработки исключения
- В простейшем варианте - для любых исключений

ИСКЛЮЧЕНИЯ

- В момент исполнения мы находимся где-то на стеке вызовов
- Каждая точка вызова либо находится непосредственно внутри обработчика, либо нет
- Это статически известный факт про любой точку исходного кода
- А конкретный стек вызовов - это явление динамическое

ИСКЛЮЧЕНИЯ

- Идем по стеку вызовов и ищем первый вызов внутри подходящего обработчика
- Чей `except` обрабатывает наше исключение
- В текущем примере до любого обработчика
- По ходу движения стековые фреймы сворачиваются (вызовы "завершаются")

ИСКЛЮЧЕНИЯ

- Дойдя до обработчика - передаем управление в его `except`-блок
- В этом момент исключение считается обработанным
- Возобновляется обычное исполнение кода
- В предельном случае можем сразу найти обработчик, без очистки фреймов
- В противоположном - обработчика не найдем, процесс завершится, напечатается ошибка со стеком вызовов

ПРИМЕР

```
1 v = int(input())
2
3 try:
4     print(1/v)
5 except:
6     print('got exception')
```

ПРИМЕР

```
1 v = int(input())
2
3 try:
4     print(1/v)
5 except Exception as exc:
6     print('got exception', exc, type(exc))
```

ИСКЛЮЧЕНИЯ

- Факт создания объекта-исключения на исполнение кода не влияет
- Влияет - факт бросания исключения
- Программно бросается с помощью ключевого слова `raise`
- В момент бросания обычное исполнение кода прекращается
- Ищется обработчик исключения

ИСКЛЮЧЕНИЯ

- Обработчик исключения - конструкция `try/except`
- После `try` - блок кода для исполнения
- После `except` - блок для обработки исключения
- В простейшем варианте - для любых исключений

ОБРАБОТКА ИСКЛЮЧЕНИЯ

- Исключение обрабатывается ближайшим обработчиком
- Ближайшим по вложенности из обработчиков внутри функции
- Или по вложенности вызовов
- Сначала пытаемся найти внутри функции, потом идем вверх по стеку

ОБРАБОТКА ИСКЛЮЧЕНИЯ

- Внутри одного `try/except` может быть много веток `except`
- Перебираются все по очереди
- Выбирается первая подошедшая
- Возможна ветка `else`
- И ветка `finally`

ОБРАБОТКА ИСКЛЮЧЕНИЯ

- Подходящая `except`-ветка - та, в которой указан класс, экземпляром которого является обрабатываемое исключение
- Класс, указанный в одной ветке, может быть подклассом того, что указан в другой
- Это имеет смысл только, если ветка с подклассом идет раньше
- Иначе она никогда не исполнится

ВСТРОЕННЫЕ ИСКЛЮЧЕНИЯ

- В корне иерархии - класс `BaseException`
- У него пять прямых потомков, основной - `Exception`
- Другие - `BaseExceptionGroup, GeneratorExit, KeyboardInterrupt, SystemExit`
- Они все - в чем-то "особенные"

EXCEPTION VS BASEEXCEPTION

- Если в except не указать исключения, то это ветка ловит `BaseException`
- Иногда это может приводить к странностям
- Например, когда в программной логике есть основной цикл
- И мы сильно не хотим, чтобы программа выпала из-за ошибки
- Массово подавлять исключения не очень правильно - но на уровне главного цикла это может быть оправдано

КАРКАС ПРИЛОЖЕНИЯ

```
 1 import sys
 2
 3 def read_request():
 4     ...
 5
 6 def write_response(resp):
 7    ...
 8
 9 def process(req):
10     return ...
11
12
13 # ...................
```

КАРКАС ПРИЛОЖЕНИЯ

```
 1 # ...................
 2
 3 def main_loop():
 4     while True:
 5         try:
 6             req = read_request()
 7             resp = process(req)
 8             write_response(resp)
 9         except:
10             print('got exception')
11             pass
12
13 main_loop()
```

ЧТО НЕ ТАК

- Мы нее выйдем по Ctrl-C
- Если внутри `process` или `read_request` вызвать `sys.exit` - мы не выйдем
- Про `BaseExceptionGroup`, `GeneratorExit` - разберем попозже
- Кто потомок `BaseException`, но не потомок `Exception` - те, кого скорее не хотелось бы массово обрабатывать наравне со всеми
- Надо хорошо думать над каждым `'except:'` или `'except BaseException:'`