Элементы функционального программирования

Лямбда функции

Лямбда функции в Python – это такие функции, которые не имеют названия. Их также называют анонимными. Слово «lambda» является служебным, и не отражает сути конкретной функции. Не требуют returnи записываются в одной строке. Используется в коде единожды, может входить в состав других языковых конструкций.

Создание лямбда функций происходит с помощью ключевого слова lambda следующим образом:

Раздаточный материал № 114

lambda <аргумент(ы)>: <выражение>

Лямбда функции могут иметь сколько угодно аргументов или не иметь их вовсе, но обязательно должны содержать лишь одно выражение.

Лямбда функции лучше использовать в связке с обычными функциями, например, для работы с итерируемыми объектами (map(), reduce(), zip(), filter()).

map() — это встроенная функция Python, принимающая в качестве аргумента функцию и последовательность. Она работает так, что применяет переданную функцию к каждому элементу.

Предположим, есть список целых чисел, которые нужно возвести в квадрат с помощью map.

Раздаточный материал № 115

```
# список целых чисел, которые нужно возвести в квадрат
L = [1, 2, 3, 4]
print(list(map(lambda x: x**2, L)))
```

filter() — отфильтровывает некоторые элементы итерируемого объекта (например, списка) на основе какого-то критерия. Критерий определяется за счет передачи функции в качестве аргумента. Она же применяется к каждому элементу объекта.

Раздаточный материал № 116

print(list(filter(lambda x: x % 2 == 0, [1, 3, 2, 5, 20, 21])))

reduce() принимает два параметра — функцию и список. Сначала она применяет стоящую первым аргументом функцию для двух начальных элементов списка, а затем использует в качестве аргументов этой функции полученное значение вместе со следующим элементом списка и так до тех пор, пока весь список не будет пройден, а итоговое значение не будет возвращено. Для того, чтобы использовать reduce(), ее необходимо сначала импортировать ее из модуля functools.

Раздаточный материал № 117

```
fromfunctools import reduce
print(reduce(lambda x,y: y-x, L)) # работа reduce
                                  # 3 - 1 = 2
                                  # 2 - 2 = 0
                                  # 5 - 0 = 5
                                  # 20 - 5 = 15
                                  # 21 - 15 = 6
```

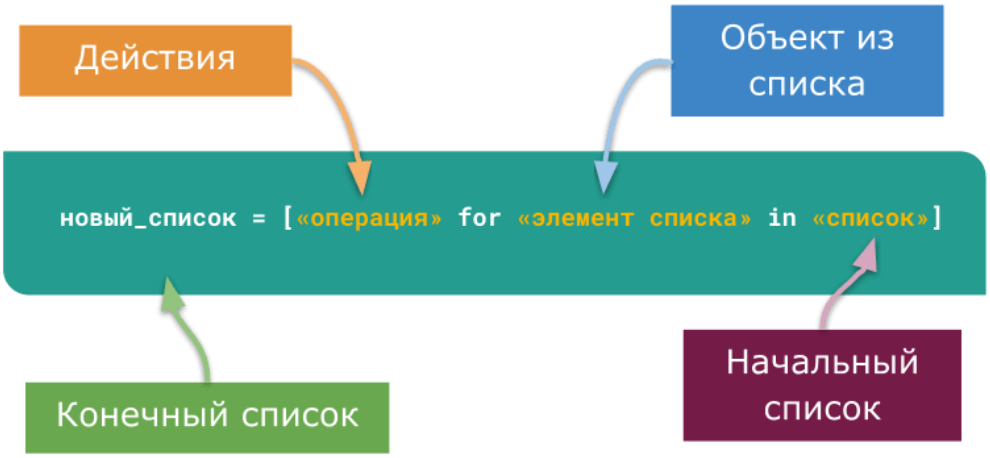
Списковое включение (List comprehension)

List comprehension — это упрощенный подход к созданию списка, который задействует цикл for, а также инструкции if-else для определения того, что в итоге окажется в финальном списке.Перевод: списковое включение или представление списка.

У list comprehension есть три основных преимущества:

- Простота. List comprehension позволяют избавиться от циклов for, а также делают код более понятным.
- Скорость. List comprehension быстрее for-циклов, которые он и заменяет.
- Принципы функционального программирования. Поскольку list comprehensions создают новый список, не меняя существующий, их можно отнести к функциональному программированию.

Раздаточный материал № 118



– операция подразумевает некие действия, которые необходимо применить к каждому элементу списка;

– элемент списка – каждый отдельный объект списка;

– список – последовательность, т.е. элементы, к которым будет применятся операция

Раздаточный материал № 119

#1

```
# в  интернет-магазине сегодня 10% скидка на ряд товаров

price = [500, 1200, 800, 600, 150]
price_new = [n * (1 - 0.1) for n in price]
```

```
print('Старый прайс',price)
print('Новый прайс', price_new)
```

#2
>>>nums = [n for n in range(1,6)]
>>> print(nums)
[1, 2, 3, 4, 5]

Условие в конце включения позволяет отсечь часть элементов итератора, это своего рода фильтрация элементов.

Раздаточный материал № 120
новый_список = [«операция» for «элемент списка» in «список» if «условие»]
Важно: здесь невозможно использовать elif, else или другие if

```
price_new1 = [n * (1 - 0.1) for n in price if n <1000]
print('Новый прайс со стоимостью менее 1 тыс. руб.', price_new1)
```

Если требуется не фильтрация данных по какому-то критерию, а изменение типа операции над элементами последовательности, условия могут использоваться в начале генератора списков.

Раздаточный материал № 121
новый_список = [«операция» if «условие» for «элемент списка» in «список»]
Важно: условие может дополняться вариантом else (но elifневозможен).

Раздаточный материал № 122

```
# Дана строка, в которой могут присутствовать буквы любых алфавитов.
# Составить новый список, где напротив каждой буквы будет отмечено,
# является ли она английской или нет

from string import ascii_letters #string импортирован объект ascii_letters,
                                 # в котором содержатся только буквы
английского алфавита

letters = 'hытфтгцзqп'  # набор букв из разных алфавитов

# Разграничиваем буквы на английские и не английские
is_eng = [f'{letter}-ДА' if letter in ascii_letters else f'{letter}-НЕТ' for
letter in letters]
print(is_eng)
```

Генераторы списков могут иметь несколько уровней вложенности. Но такие конструкции являются громоздкими и неудобно читаемыми.

Раздаточный материал № 123

```
# Генерация таблицы умноженияот 3 до 7
table = [
    [x * yforxinrange(3, 8)]
foryinrange(3, 8)]
print(table)
```

Подобные конструкции позволяют создавать не только списки, но и множества (set comprehension – при помощи «{}»), генераторы (generator expression – при помощи «()»), а также словари (dictionary comprehension – при помощи «{}»). Принцип везде один и тот же.

Раздаточный материал № 124

```
# выбрать все гласные буквы из исходной фразы
fraza = "я изучаю язык Питон"
new_fraza = {i for i in fraza if i in 'аеёиоуэюя'}
print(new_fraza)
```

```
#в словаре в качестве значения ключа поместить его квадрат
squares = {i: i * i for i in range(10)}
print(squares)
```

Zip() - объединяет элементы различных итерируемых объектов (таких как списки, кортежи или множества) и возвращает итератор кортежей, где i-й кортеж содержит i-й элемент из каждого списка. В качестве аргумента принимает список итерируемых объектов. Если аргумента нет, функция zip возвращает пустой итератор.

Раздаточный материал № 125

```
id = [1, 2, 3, 4]
name = ['Меркурий', 'Венера', 'Земля', 'Марс']

rec = zip(id, name) # объединение для двух списков
print(list(rec))

radius = [2439, 6051, 3678, 3376]
rec1 = zip(id, name, radius) # объединение для трех списков
print(list(rec1))

radius_1 = [2439, 6051, 3678]
rec2 = zip(id, name, radius_1) # объединение для трех списков по длине
наименьшего
print(list(rec2))

name_dict_1 = {i: nd for i, nd in zip(id, name)} # создание словаря с
использованием dict comprehension
print(name_dict_1)

name_dict_2 = dict(zip(id, name)) # создание словаря с использованием dict
comprehension
print(name_dict_2)

# добавим в словарь новые значения

new_id = [5]
new_name = ['Юпитер']
name_dict_2.update(zip(new_id, new_name))
print(name_dict_2)

# zip и выполнение расчетов
diff = [a-b for a, b in zip(radius, radius[1:])]
print(diff)
```

Итераторы и итерируемые объекты. Генераторы.

Итерабельный объект представляет собой объект, элементы которого можно перебирать в цикле или иными доступными способами.

Итератор — это объект, который выполняет фактическую итерацию.

Можно создать итератор из любого итерабельного объекта, вызвав встроенную функцию iter(). Чтобы получить следующий элемент из него используется встроенная функцияnext, если элементов больше нет, то выбрасывается исключение StopIteration.

Раздаточный материал № 126

```
>>> lst = [1, 6, 8, 10, 20, 2, 5]
>>> type(lst)
<class 'list'>
>>> lst_it = iter(lst)
>>> type(lst_it)
<class 'list_iterator'>
>>> dir(lst)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format_
>>> dir(lst_it)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__'
>>> next(lst_it)
1
>>> next(lst_it)
6
>>> next(lst_it)
8
>>> next(lst_it)
10
>>> next(lst_it)
20
>>> next(lst_it)
2
>>> next(lst_it)
5
>>> next(lst_it)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Есть еще одно правило об итераторах: итераторы также являются итераторабельными объектами, а их итератор — это они сами.

Итераторы позволяют создать итерабельный объект, который перебирает свои элементы по мере выполнения итерации. Это означает, что можно создавать «ленивые» итераторы, которые не определяют следующий элемент, пока их не попросят об этом.

Использование итератора вместо списка, множества или другого итерируемого объекта позволяет экономить память и время обработки элементов.

> Например, можно создать итератор, который предоставит нам 100 миллионов четверок и это займет на ПК примерно 60 байт памяти, соответствующий список – 800 Мб.Или если необходимо посчитать сумму продаж по гамбургерам за 15 лет – нет смысла загружать файл себе на ПК, проще «пробежать» один раз по этому файлу и выбрать нужную информацию.

Файловые объекты в Python реализованы как итераторы. При итерации по файлу данные считываются в память по одной строке за раз. Если бы вместо этого использовался метод readlines для хранения всех строк в памяти, мы могли бы исчерпать всю системную память и убить процесс.

Кроме того, у итераторов есть возможности, которых нет у других итерабельных объектов. «Ленивые» итераторы можно использовать для создания итерабельных объектов неизвестной длины (значит, len применять нельзя).

Итерабельный объект можно перебирать вручную, воспользоваться for или создать класс, с котором переопределить __iter__, __next__. Все это не самые хорошие варианты создания итератора. Как правило, в этом случае программист пишет генератор.

Генераторы особенно полезны для веб-скрапинга и увеличения эффективности поиска. Они позволяют получить одну страницу, выполнить какую-то операцию и двигаться к следующей. Этот подход куда эффективнее чем получение всех страниц сразу и использование отдельного цикла для их обработки.

В Python есть два способа создания генераторов.
1. Использование выражения-генератора (generator expression) – при помощи «()».

Раздаточный материал № 127

numbers = [6, 57, 4, 7, 68, 95]

sq = (n**2 for n in numbers)

2. Использование функции-генератора. Это обычная функция, но вместо return она использует yield (один или несколько). Инструкция yield уведомляет интерпретатор Python о том, что это генератор, и возвращает итератор. Инструкция временно приостанавливает исполнение, сохраняет состояние и затем может продолжить работу позже.

Раздаточный материал № 128

```
# Вариант 1
defsq_all(numbers):
for n in numbers:
yield n ** 2

numbers = [6, 57, 4, 7, 68, 95]
squares = sq_all(numbers)

for i in squares:
print(i)
# Вариант 2
def sq_all(numbers):
# оператор for убирается как самостоятельная конструкция
yield from [n ** 2 for n in numbers]


numbers = [6, 57, 4, 7, 68, 95]
squares = sq_all(numbers)

for i in squares:
print(i)
```

Оба этих объекта-генератора работают одинаково. Они оба имеют тип generator и оба являются итераторами.

Раздаточный материал № 129

```
# В заданной строке найти все прописные буквы, посчитать их количество.
# Использовать библиотеку string

from string import ascii_uppercase

string_new = 'In PyCharm, you can specify third-party standalone applications
and run them as External Tools'
str_1 = [i for i in string_new if i in ascii_uppercase]
print(len(str_1))
print(str_1)
```

Раздаточный материал № 130 Константы библиотеки srting (справочно)

string.ascii_letters

Объединение констант ascii_lowercase и ascii_uppercase описано ниже. Значение не зависит от языкового стандарта.

string.ascii_lowercase

Строчные буквы 'abcdefghijklmnopqrstuvwxyz'.

string.ascii_uppercase

Заглавные буквы 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.

string.digits

Строка '0123456789'.

string.hexdigits

Строка '0123456789abcdefABCDEF'.

string.octdigits

Строка '01234567'.

string.punctuation

Строка символов ASCII, которые считаются символами пунктуации: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

string.printable

Строка символов ASCII, которые считаются печатаемыми. Комбинация digits, ascii_letters, punctuation и whitespace.

string.whitespace

Строка, содержащая все символы ASCII, считающиеся пробелами. Включает в себя пространство символов, табуляцию, перевод строки, возврат, перевод страницы и вертикальную табуляцию.

Вопросы:

- Понятие, применение лямбда функций.
- Назначение функции map().
- Пример map() и лямбда функции (на доске).
- Назначение функции filter().
- Пример filter () и лямбда функции (на доске).
- Назначение функции reduce().
- Пример reduce() и лямбда функции (на доске).
- Преимущества List comprehension
- Пример списковых включений (на доске).
- Пример списковых включений с if (на доске).
- Назначение функции zip().
- Пример работы zip с тремя списками (на доске).
- Понятие и пример итерабельного объекта.
- Понятие и пример итератора.
- Пример выражения-генератора (на доске).
- Отличие функции-генератора от обычной функции. Описание работы.
- Пример функции-генератора (на доске).