GeekBrains

Функции высшего порядка, работа с файлами.

Курс

Оглавление

Анонимные, lambda-функции

На предыдущей лекции мы с Вами обговорили создание и использование функций в Python, но некоторые функции могут понадобиться всего раз за всю работу приложения. Как можно обойтись без их явного описания? Как сократить подобный код?

```
def f(x)
      return x ** 2
print(f(2))
```

💡 Функция в примере занимает всего две строчки кода, но в дальнейшем размеры описания функций будут увеличиваться. И тогда сокращение кода будет актуальным.

Какой тип данных у функции? → <class “function”>

У функции есть тип, значит мы можем создать переменную типа функции и положить в эту переменную какую-то другую функцию.

```
def f(x)
      return x ** 2
g = f
```

Теперь в контексте этого приложения **g** может использоваться точно так же, как и **f**.

g — это переменная, которая хранит в себе ссылку на функцию.

```
def f(x)
      return x ** 2
g = f
print(f(4)) # 16
print(g(4)) # 16
```

Зачем это может потребоваться?

Есть некая функция **calc**, которая принимает в качестве аргумента число, а в качестве результата возвращает это число + 10:

```
def calc1(x)
      return x + 10
print(calc1(10)) # 20
```

Если мы добавим в код не только сложение, но и умножение, деление и вычитание, внутри одного кода будем плодить одинаковую логику.

💡 Достаточно взять функцию calc, которая будет в качестве аргумента принимать операцию и что-то выдавать.

```
def calc2(x)
      return x * 10

def math(op, x)
      print(op, x)

math(calc2, 10) # 100
```

Попробуем описать ту же логику для функции с двумя переменными.

op — операция, воспринимаем её как отдельную функцию. В примере это либо сумма (**sum**), либо перемножение(mylt):

```
def sum(x, y):
      return x + y

def mylt(x, y):
      return x * y
```

```
def calc(op, a, b):
      print(op(a, b))


calc(mylt, 4, 5) # 20
```

Можно создать псевдоним для функции сложения (f).

```
def sum(x, y):
      return x + y
f = sum
calc(f, 4, 5) # 9
```

В Python есть механизм, который позволяет превратить подобный вызов во что-то более красивое — **lambda**.

```
def sum(x, y):
      return x + y
# ⇔ (равносильно)
sum = lambda x, y: x + y
```

Теперь, чтобы вызвать функцию суммы, достаточно просто вызвать **sum**.

Также можно пропустить шаг создания переменной sum и сразу вызвать **lambda**:

```
calc(lambda x, y: x + y, 4, 5) # 9
```

Итак:

1. Сначала мы избавились от классического описания функций.
2. Затем научились описывать лямбды, присваивая результат какой-то переменной.
3. После избавились от этой переменной, пробрасывая всю лямбду в качестве функции.

Задача для самостоятельного решения

1. В списке хранятся числа. Нужно выбрать только чётные числа и составить список пар (число; квадрат числа).

Пример: 1 2 3 5 8 15 23 38

Получить: [(2, 4), (8, 64), (38, 1444)]

Решение:

```
data = [1, 2, 3, 5, 8, 15, 23, 38]
out = []
for i in data :
      if i % 2 == 0:
          out.append((i, i ** 2))
print(out)
```

Как можно сделать этот код лучше, используя lambda?

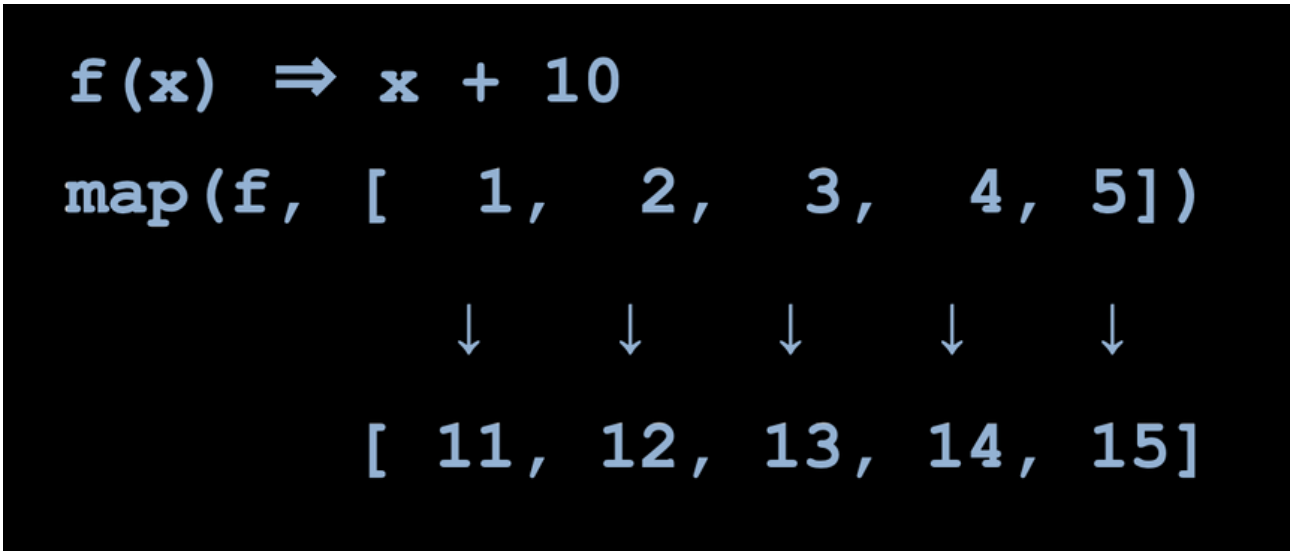
```
def select(f, col):
     return [f(x) for x in col]


 def where(f, col):
     return [x for x in col if f(x)]


data = [1, 2, 3, 5, 8, 15, 23, 38]
res = select(int, data)
```

```
res = where(lambda x: x % 2 == 0, res)

print(res) # [2, 8, 38]

res = list(select(lambda x: (x, x ** 2), res))
```

Функция map

💡 Функция map() применяет указанную функцию к каждому элементу итерируемого объекта и возвращает итератор с новыми объектами.

Есть набор данных. Функция map позволяет увеличить каждый объект на 10.




```
list_1 = [x for x in range (1,20)]

list_1 = list(map(lambda x: x + 10, list_1 ))

print(list_1)
```

Задача: С клавиатуры вводится некий набор чисел, в качестве разделителя используется пробел. Этот набор чисел будет считан в качестве строки. Как превратить list строк в list чисел?

1. Маленькое отступление, функция **строка.split()** - убирает все пробелы и создаем список из значений строки, пример:

```
data = '1 2 3 5 8 15 23 38'.split()
print(data) # ['1', '2', '3', '5', '8', '15', '23', '38']
```

2. Теперь вернемся к задаче. С помощью функции **map():**

```
data = list(map(int,input().split()))
```

Результат работы map() — это итератор. По итератору можно пробежаться только один раз. Чтобы работать несколько раз с одними данными, нужно сохранить данные (например, в виде списка).

Как можно сделать этот код лучше, используя map()?

💡 map() позволит избавиться от функции select.

```
def where(f, col):
     return [x for x in col if f(x)]


data = '1 2 3 5 8 15 23 38'.split()

res = map(int, data)

res = where(lambda x: x % 2 == 0, res)
res = list(map(lambda x: (x, x ** 2), res))
print(res)
```

Функция filter

💡 Функция filter() применяет указанную функцию к каждому элементу итерируемого объекта и возвращает итератор с теми объектами, для которых функция вернула True.

```
data = [x for x in range(10)]
res = list(filter(lambda x: x % 2 == 0, data)))
print(res) # [0, 2, 4, 6, 8]
```

Как в данном случае работает функция **filter()**? Все данные, которые находятся внутри проходят через функцию, которая указана следующим образом:

```
lambda x: x % 2 == 0
```

То есть делает проверка на те числа, которые при делении на 2 дают в остатке 0. Тем самым мы ищем только четные числа. Действительно преобразовав наши итоговые данные в список, с помощью функции **list()**, мы с Вами можем наблюдать такой красивый результат 🙂

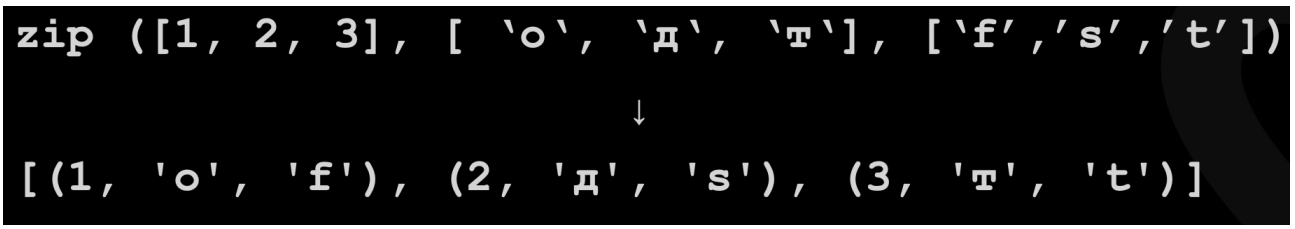
Как можно сделать этот код лучше, используя **filter()**?

💡 filter() позволит избавиться от функции where, которую мы писали ранее

```
data = '1 2 3 5 8 15 23 38'.split()
res = map(int, data)
res = filter(lambda x: x % 2 == 0, res)
res = list(map(lambda x: (x, x ** 2), res))
print(res)
```

Функция zip

💡 Функция zip() применяется к набору итерируемых объектов и возвращает итератор с кортежами из элементов входных данных



На выходе получаем набор данных, состоящий из элементов соответствующих исходному набору.

Пример:

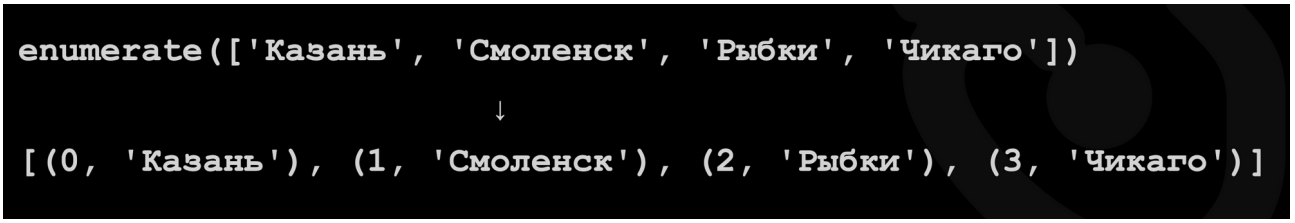
```
users = ['user1', 'user2', 'user3', 'user4', 'user5']
ids = [4, 5, 9, 14, 7]
data = list(zip(users, ids))
print(data)    #  [('user1',  4),  ('user2',  5),  ('user3',  9),  ('user4',  14),
('user5', 7)]
```

Функция zip () пробегает по минимальному входящему набору:

```
users = ['user1', 'user2', 'user3', 'user4', 'user5']
ids = [4, 5, 9, 14, 7]
salary = [111, 222, 333]
data = list(zip(users, ids, salary))
print(data) # [('user1', 4, 111), ('user2', 5, 222), ('user3', 333)]
```

Функция enumerate

💡 Функция enumerate() применяется к итерируемому объекту и возвращает новый итератор с кортежами из индекса и элементов входных данных.



Функция enumerate() позволяет пронумеровать набор данных.

```
users = ['user1', 'user2', 'user3']

data = list(enumerate(users)

print(data) # [(0, 'user1'), (1, 'user2'), (2, 'user3))]
```

Файлы

Файлы в текстовом формате используются для:

- Хранения данных
- Передачи данных в клиент-серверных проектах
- Хранения конфигов
- Логирования действий

Что нужно для работы с файлами:

1. Завести переменную, которая будет связана с этим текстовым файлом.
2. Указать путь к файлу.
3. Указать, в каком режиме мы будем работать с файлом.

Варианты режима (мод):

1. **a** – открытие для добавления данных.
    - Позволяет дописывать что-то в имеющийся файл.
    - Если вы попробуете дописать что-то в несуществующий файл, то файл будет создан и в него начнётся запись.
2. **r** – открытие для чтения данных.
    - Позволяет читать данные из файла.
    - Если вы попробуете считать данные из файла, которого не существует, программа выдаст ошибку.
3. **w** – открытие для записи данных.
    - Позволяет записывать данные и создавать файл, если его не существует.

Миксованные режимы:

4. w+
    - Позволяет открывать файл для записи и читать из него.
    - Если файла не существует, он будет создан.
5. r+
    - Позволяет открывать файл для чтения и дописывать в него.
    - Если файла не существует, программа выдаст ошибку.

Примеры использования различных режимов в коде:

1. **Режим a**

```
colors = ['red', 'green', 'blue']

data = open('file.txt', 'a') # здесь указываем режим, в котором будем работать

data.writelines(colors) # разделителей не будет

data.close()
```

- data.close() — используется для закрытия файла, чтобы разорвать подключение файловой переменной с файлом на диске.
- exit() — позволяет не выполнять код, прописанный после этой команды в скрипте.
- В итоге создаётся текстовый файл с текстом внутри: redbluedreen.
- При повторном выполнении скрипта redbluedreenredbluedreen — добавление в существующий файл, а не перезапись файлов.

Ещё один способ записи данных в файл:

```
with open('file.txt', 'w') as data:

    data.write('line 1\n')

    data.write('line 2\n')
```

2. **Режим r**

- Чтение данных из файла:

```
path = 'file.txt'

data = open(path, 'r')

for line in data:

    print(line)

data.close()
```

3. **Режим w**

```
 colors = ['red', 'green', 'blue']
 data = open('file.txt', 'w')
 data.writelines(colors) # разделителей не будет
data.close()
```

- В итоге создаётся текстовый файл с текстом внутри: **‘redbluedreen’**.
- В случае перезаписи новые данные записываются, а старые удаляются.

Модуль os

Модуль os предоставляет множество функций для работы с операционной системой, причем их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми.

Для того, чтобы начать работать с данным модулем необходимо его импортировать в свою программу:

```
import os
```

Познакомимся с базовыми функциями данного модуля:

- **os.chdir**(path) - смена текущей директории.

```
import os
os.chdir('C:/Users/79190/PycharmProjects/GB')
```

- **os.getcwd()** - текущая рабочая директория

```
import os
print(os.getcwd()) # 'C:\Users\79190\PycharmProjects\webproject'
```

- **os.path -** является вложенным модулем в модуль os и реализует некоторые полезные функции для работы с путями, такие как:
    - **os.path.basename**(path) - базовое имя пути

```
import os

print(os.path.basename('C:/Users/79190/PycharmProjects/webproject/main.py')) #
'main.py'
```

- **os.path.abspath(path)** - возвращает нормализованный абсолютный путь.

```
import os

print(os.path.abspath('main.py'))

# 'C:/Users/79190/PycharmProjects/webproject/main.py'
```

Это лишь малая часть возможностей модуля **os**.

Модуль shutil

Модуль **shutil** содержит набор функций высокого уровня для обработки файлов, групп файлов, и папок. В частности, доступные здесь функции позволяют копировать, перемещать и удалять файлы и папки. Часто используется вместе с модулем os.

Для того, чтобы начать работать с данным модулем необходимо его импортировать в свою программу:

```
import shutil
```

Познакомимся с базовыми функциями данного модуля:

- **shutil.copyfile**(src, dst) - копирует содержимое (но не метаданные) файла src в файл dst.
- **shutil.copy**(src, dst) - копирует содержимое файла src в файл **или папку** dst.
- **shutil.rmtree**(path) - Удаляет текущую директорию и все поддиректории; path должен указывать на директорию, а не на символическую ссылку.

Итоги:

1. Изучали функции высшего порядка, такие как:
    - **map**
    - **filter**
    - **zip**
    - **lambda**
    - **enumerate**

2. Научили работать с файлами

3. Изучили библиотеки для работы с операционной системой и файлами