Функция как объект. Лямбда-функции

План урока:

1. Получение и использование объекта функции, запись различных функций в переменную.
2. Функции высшего порядка. Функция `filter`. Разворачивание итератора при помощи `list`.
3. Лямбда-функции.
4. Константные функции для согласования сигнатуры.
5. Написание собственных функций высшего порядка.
6. Функция `map`.
7. Списочные выражения.
8. Использование существующих функций и методов в качестве аргументов функций высшего порядка.

Аннотация

В языке Python всё является объектом, даже функция. Вы узнаете, как получить и использовать соответствующий объект. Кроме того, вы научитесь создавать и использовать крошечные функции буквально в полстроки, и передавать эти функции в качестве аргумента другой функции. Вы узнаете про две важные функции: `filter` и `map`, которые позволяют проводить множество преобразований над коллекциями и другими наборами объектов. Заодно мы вспомним списочные выражения и расширим знания о них.

Функция как объект

К настоящему моменту вы узнали про переменные, циклы, объекты вообще и списки в частности, а также про функции. На этом уроке мы будем в некотором смысле сокращать этот список. До сих пор мы рассматривали функции как совершенно отдельный элемент языка со своим синтаксисом и механизмами работы. Но оказывается, что функция - это что-то вроде особого типа объектов. Бывают числа, бывают строки, бывают списки. А бывают - функции. У каждого из этих типов есть свои операции, свой синтаксис, но все они являются объектами. Например есть объект, который умеет печатать текст на экране. У него есть имя `print`. Можно считать, что `print` - это что-то вроде имени переменной, которая хранит объект функции. Давайте посмотрим, насколько далеко идет эта аналогия.

Первым делом, давайте объект функции получим. Для этого достаточно написать имя функции без скобочек. Это абсолютно аналогично тому, как происходит со списками или строками: если вы пишете после имени списка квадратные скобочки с индексом - выполняется операция взятия элемента, не пишете скобочки - получаете сам список. Пишете после функции круглые скобочки с аргументами - она вызывается, не пишете - получаете саму функцию, как объект.

`input` - объект функции чтения из стандартного ввода. Давайте мы этот объект напечатаем:

```
print(input)
```

Python выдает нам, текстовое представление этой функции: строку "`<built-in function input>`", которая поясняет, что `input` - встроенная функция языка.

Вопросы для самопроверки:

- Что сделает `print(input())` и почему это отличается от `print(input)`?
- Как вывести на экран функцию печати?

Раз мы можем получить какой-то объект, мы его можем записать в переменную. Давайте попробуем!

```
vyvod = print
```

Это сработало. Теперь у функции print есть псевдоним на транслите. Эту новую переменную можно вызвать, как и обычную функцию, посредством круглых скобочек:

```
vyvod('Privet mir!')
```

Возможность записать функцию в переменную позволяет нам гибко управлять тем, какую функциональность мы хотим использовать. В одну и ту же переменную мы можем записать разные варианты поведения и менять их при необходимости. При этом нам не только не нужно будет менять код по всей программе, но не придется даже изменять код функций. Достаточно переменной присвоить вместо одной функции - другую.

Напишем программу, которая печатает списки либо просто через запятую, либо в "коробочках", в зависимости от значения переменной `formatting`.

```
def print_boxed(arr):
    arr_stringified = [str(element) for element in arr]
    mid = ' | '.join(arr_stringified)
    bar = '-' * (2 + len(mid))
    print(' ' + bar + ' ')
    print('| ' + mid + ' |')
    print(' ' + bar + ' ')

def print_simple(arr):
    arr_stringified = [str(element) for element in arr]
    print(', '.join(arr_stringified))

formatting = 'boxed'
if formatting == 'boxed':
    print_formatted = print_boxed
else:
    print_formatted = print_simple

# Дальше в программе можно использовать print_formatted повсюду
print_formatted([1,1,2,3,5,8,13,21])
print_formatted([1,2,4,8,16,32,64,128])
print_formatted(['abc', 'def', 'ghi'])
```

```
 ---------------------------------
| 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |
 ---------------------------------
 -----------------------------------
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
 -----------------------------------
 -----------------
| abc | def | ghi |
 -----------------
```

Мы один раз проверили условие, и указали, как должна вести себя функция `print_formatted` по всей программе.

Функции высшего порядка. Функция `filter`

Объект функции можно не только записать в переменную, но и передать в качестве аргумента в другую функцию (и даже вернуть из функции, о чём вы можете узнать из дополнительных материалов). Функции, которые принимают или возвращают другие функции, называются функциями высшего порядка.

Часто функции высшего порядка используются для обработки наборов данных. Например, из раза в раз встречается такая задача: взять список элементов и оставить среди них только небольшую часть, согласно какому-то критерию.

Эту задачу можно встретить в самых разных формах. В списке товаров найти только дешевые. Отобрать все слова, в которых ровно три слога - для генератора рифм. Найти среди кораблей в игре в "морской бой" все подбитые.

Это разные списки и разные критерии, а задача одна и та же: отфильтровать элементы списка. В языке Python для этой цели есть встроенная функция `filter`.

Функция `filter` принимает список элементов и критерий отбора. Возвращает она список из элементов, удовлетворяющих критерию.

Чтобы этой функцией воспользоваться, нужно сообщить функции `filter` критерий, который говорит, брать ли элемент в результирующий список или нет. Давайте напишем простую функцию, которая проверяет, что слово длиннее шести букв, и затем отберем с её помощью длинные слова.

```
def isWordLong(word):
    return len(word) > 6

words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные', 'слова']
for word in filter(isWordLong, words):
    print(word)
# => останутся
# => длинные
```

С методом `filter` вам не нужно вручную создавать и заполнять список, достаточно указать условие отбора.

Но если вы попробуете распечатать результат функции `filter` при помощи функции `print` (а не перебирая элементы по одному в цикле `for`), то будете удивлены: будет выведен не список, а специальный объект "`<filter object at 0x...>`".

Он похож на список тем, что его можно перебирать циклом `for`, т.е. итерировать. Такие объекты называют итераторами. Чтобы получить из итератора список, можно воспользоваться функцией `list`:

```
long_words = list(filter(isWordLong, words))
```

Описанный способ отфильтровать список пока далек от удобного, поскольку нам приходится заводить функцию для каждой проверки, что занимает две лишних строки кода. Для каждой такой маленькой функции приходится придумывать имя (и загромождать пространство имен).

Для того чтобы создавать такие короткие функции "на один раз" в языке Python есть специальный синтаксис.

Лямбда-функции

Очень часто в качестве аргумента для функций высшего порядка часто мы хотим использовать совсем простую функцию. Причем нередко такая функция нужна в программе только в одном месте, поэтому ей необязательно даже иметь имя.

Такие короткие безымянные (анонимные) функции можно создавать инструкцией

```
lambda <аргументы>: <выражение>
```

Такая инструкция создаст функцию, принимающую указанный список аргументов и возвращающую результат вычисления выражения. В языке Python тело лямбда-функции имеет ровно одно выражение. Инструкция `return` подразумевается, писать её не требуется, да и нельзя. Скобочки вокруг аргументов не пишутся, аргументы от выражения отделяет двоеточие.

Теперь мы можем записать функцию, проверяющую длину слова следующим образом:

```
lambda word: len(word) > 6
```

И список длинных слов теперь извлечь очень просто:

```
long_words = list(filter(lambda word: len(word) > 6, words))
```

Лямбда-функция - полноценная функция. Её можно использовать в составе любых конструкций. Например, если вы хотите использовать её несколько раз, но не хотите определять функцию с помощью `def`, вы можете присвоить созданную лямбда-функцию какой-либо переменной.

```
add = lambda x, y: x + y
add(3, 5) # => 8
add(1, add(2, 3)) # => 6
```

Задачи: Набор юного арифметика, Таблица операции.

А теперь рассмотрим вариант, что вам нужно взять все элементы списка, но в программе уже стоит `filter` и вам не хочется его удалять. В этой ситуации вам поможет функция с особенно простым выражением:

```
lambda x: True
```

Покажем:

```
def print_some_primes(criterion):
    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
    for number in filter(criterion, primes):
        print(number)

print_some_primes(lambda x: True)
# => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Обратите внимание, что лямбда-функция принимает аргумент, хотя и не использует его. Функция `filter` всегда передает в критерий элемент, который проверяет. Если бы мы написали лямбда-функцию без аргументов `lambda: True`, то функция `filter` вызвала бы ошибку, потому что передала бы аргумент в функцию, которая аргументов не принимает.

Как написать свой filter

Чтобы функция `filter` не казалась магической, напишем свой упрощенный аналог. Наша функция `simple_filter` будет принимать критерий и список и возвращать новый список.

```
def simple_filter(criterion, arr):
    result = []
    for element in arr:
        if criterion(element):
            result.append(element)
    return result
```

Мы передали критерий как функцию, а потому можем его вызвать, что мы и сделали в условном операторе.

Так как для перечисления элементов мы использовали конструкцию `for`, мы можем вместо списка в функцию передать любой итерируемый объект. Например строку (элементами будут отдельные символы) или интервал `range`.

Например, найдем все числа от 1 до 99, которые при делении на 12 дают 7 в остатке.

```
simple_filter(lambda x: x % 12 == 7, range(1,100))
# => [7, 19, 31, 43, 55, 67, 79, 91]
```

Функция map

Другая популярная функция высшего порядка называется `map`. Функция `map` преобразует каждый элемент списка по некоторому общему правилу и в результате создает список (вернее, как и `filter`, специальный итерируемый объект, похожий на список) из преобразованных значений.

Функция, которую `map` принимает - преобразование одного элемента. Зная как преобразуется один элемент, `map` выполняет превращение целых списков.

Например, возьмем набор из чисел от 1 до 10 и применим к ним функцию возведения в квадрат.

```
list(map(lambda x: x**2, range(1,10)))
# => [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Задача: Просто map.

Еще немного о списочных выражениях

В уроке, посвященном списочным выражениям, вы разбирали конструкции, которые очень похожи на действие функции `map`. Например, список квадратов цифр можно посчитать так:

```
[x**2 for x in range(10)]
# => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Оказывается, что для фильтрации списка тоже есть специальный вид списочного выражения: достаточно приписать `if <условие>` в конце выражения. Такая конструкция отберет только те элементы, которые удовлетворяют условию. Например, сгенерируем список чётных цифр, не делящихся на 3:

```
[x for x in range(10) if x % 2 == 0 and x % 3 != 0]
# => [2, 4, 8]
```

Списочное выражение можно рассматривать как комбинацию фильтрации и трансформации: сначала выполняется фильтрация, затем - трансформирование. Давайте, например, возьмем список слов, оставим только длинные слова и преобразуем их в слова, написанные большими буквами:

```
words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные', 'слова']
long_words = list(map(lambda word: word.upper(),
                      filter(lambda word: len(word) > 6, words)))
# => ['ОСТАНУТСЯ', 'ДЛИННЫЕ']
# или
long_words = [word.upper() for word in words if len(word) > 6]
# => ['ОСТАНУТСЯ', 'ДЛИННЫЕ']
```

Как видите, оба способа позволяют добиться результата, но списочные выражения выглядят немного проще. В зависимости от ситуации, бывает удобно использовать либо одну форму, либо другую.

Практика

Посмотрим, как функции высшего порядка комбинируются для решения более сложных задач. Разберем, как можно было бы при помощи этих функций решить задачу **Длинношеее** из домашнего задания к уроку **Функции**.

Напомним, что нам нужно найти слова, имеющие четыре и более слогов. Мы можем разбить эту задачу на такие этапы: выделить список слов, посчитать число гласных букв и в зависимости от их числа взять слово или отбросить. Перед тем как выделять слова мы еще дополнительно отбросим пунктуацию. Каждый этап обернем в одну небольшую функцию.

```
EnglishAlphabet = [chr(c) for c in range(ord('a'), ord('z') + 1)]
RussianAlphabet = [chr(c) for c in range(ord('а'), ord('я') + 1)] + ['ё']
Alphabet = EnglishAlphabet + RussianAlphabet

EnglishVowels = ['a', 'e', 'i', 'o', 'u']
RussianVowels = ['а', 'о', 'э', 'и', 'у', 'ы', 'е', 'ё', 'ю', 'я']
Vowels = EnglishVowels + RussianVowels

def removePunctuation(text):
  return ''.join(filter(lambda c: c in Alphabet + [' '], text))

def getWords(text):
  return removePunctuation(text).split()

def numberOfVowels(word):
  return len(list(filter(lambda c: c in Vowels, word)))

def longWords(text):
  return filter(lambda word: numberOfVowels(word) >= 4, getWords(text))

def printLongWords(text):
  for word in longWords(text):
    print(word.lower())
```

Функции `ord` и `chr` служат для того, чтобы по символу получить его код (`ord`) и наоборот, по коду получить символ (`chr`). Мы применили их, потому что все буквы алфавита от "а" до "я" и от "a" до "z" расположены в таблице кодировки подряд, а значит их можно перебрать при помощи `range`. Но `range` работает с числами, а не с символами, поэтому мы сначала получаем коды начала и конца алфавита, и затем для каждого кода между начальным и конечным получаем соответствующий символ при помощи `chr`.

Функцию `filter` мы используем трижды:

- чтобы оставить только буквы и пробелы
- чтобы выделить в строке только гласные
- чтобы из всех слов отобрать только нужные

Функции получились не очень сложные и, что важно, их легко комбинировать друг с другом.

Задача: Комбинируй и властвуй.

Обратите внимание, что не всегда требуется создавать лямбда-функцию при вызове, поскольку иногда нужная функция уже существует. Например, если мы хотим преобразовать список слов в список длин слов, мы можем использовать любой из двух вариантов (но, конечно, удобнее использовать более короткий):

```
words = 'the quick brown fox jumps over the lazy dog'.split()
list(map(lambda word: len(word), words))
# => [3, 5, 5, 3, 5, 4, 3, 4, 3]
list(map(len, words))
# => [3, 5, 5, 3, 5, 4, 3, 4, 3]
```

Еще один пример - считывание списка чисел с клавиатуры:

```
numbers = list(map(float, input().split()))
```

Аналогично можно использовать в качестве передаваемой функции методы объектов. Но при этом нужно указать не только название метода, но и название типа объекта, к которому эта функция относится. Т.е. для метода строк это будет `str`, для списков `list` и т.д. Про методы и типы вы еще будете много говорить в будущих уроках, пока что только простой пример - преобразовать каждое слово в списке к верхнему регистру:

```
words = ['list', 'of', 'several', 'words']
list(map(lambda word: word.upper(), words))
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
list(map(str.upper, words))
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
```

На следующем занятии мы изучим еще множество полезных функций, которые принимают функцию в качестве аргумента. Оказывается, что даже давно известные вам функции `min`, `max`, `sort` могут принимать функцию одним из аргументов, что изменяет их поведение.