 **Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)**

**Факультет «Информатика и системы управления» Кафедра «Системы обработки информации и управления»**

Лабораторная работа на тему

«Функциональные возможности языка Python»

Выполнил:

Студент группы ИУ5-33Б

Лобанов Иван Игоревич

Проверил:

Преподаватель БКИТ  
Гапанюк Юрий Евгеньевич

2022 г.

**Задание:**

Задание лабораторной работы состоит из решения нескольких задач.

Файлы, содержащие решения отдельных задач, должны располагаться в пакете lab\_python\_fp. Решение каждой задачи должно раполагаться в отдельном файле.

При запуске каждого файла выдаются тестовые результаты выполнения соответствующего задания.

**Задача 1 (файл field.py)**

Необходимо реализовать генератор field. Генератор field последовательно выдает значения ключей словаря. Пример:

goods = [

{'title': 'Ковер', 'price': 2000, 'color': 'green'},

{'title': 'Диван для отдыха', 'color': 'black'}

]

field(goods, 'title') должен выдавать 'Ковер', 'Диван для отдыха'

field(goods, 'title', 'price') должен выдавать {'title': 'Ковер', 'price': 2000}, {'title': 'Диван для отдыха'}

* В качестве первого аргумента генератор принимает список словарей, дальше через \*args генератор принимает неограниченное количествово аргументов.
* Если передан один аргумент, генератор последовательно выдает только значения полей, если значение поля равно None, то элемент пропускается.
* Если передано несколько аргументов, то последовательно выдаются словари, содержащие данные элементы. Если поле равно None, то оно пропускается. Если все поля содержат значения None, то пропускается элемент целиком.

Шаблон для реализации генератора:

# Пример:

# goods = [

# {'title': 'Ковер', 'price': 2000, 'color': 'green'},

# {'title': 'Диван для отдыха', 'price': 5300, 'color': 'black'}

# ]

# field(goods, 'title') должен выдавать 'Ковер', 'Диван для отдыха'

# field(goods, 'title', 'price') должен выдавать {'title': 'Ковер', 'price': 2000}, {'title': 'Диван для отдыха', 'price': 5300}

def field(items, \*args):

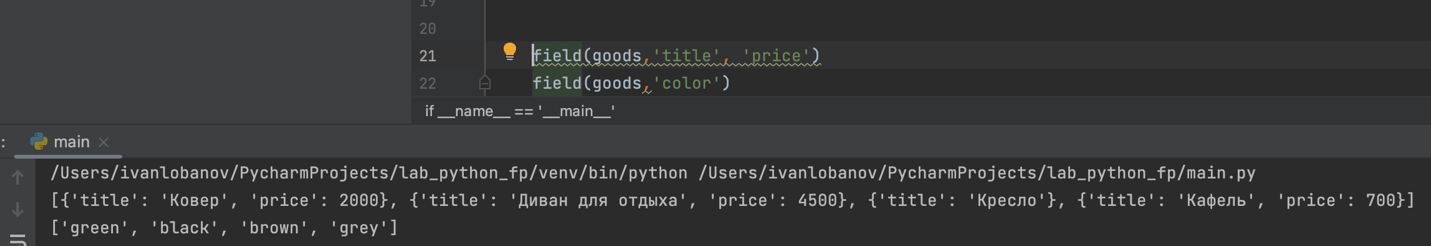
assert len(args) > 0

# Необходимо реализовать генератор

**Текст программы**

def field(list, \*args):  
 if len(args) == 0:  
 print("Введите, пожалуйста, название (-я) поля (-ей) через пробел:")  
 fields = input()  
 args = fields.split()  
 elif len(args) == 1:  
 value\_fields = []  
 for iter in list:  
 for key,word in iter.items():  
 if key==args[0]:  
 value\_fields.append(word)  
 print(value\_fields)  
 else:  
 arr = []  
 for iter in list:  
 temp = {}  
 for arg in args:  
 for key,word in iter.items():  
 if key == arg:  
 temp[key]=word  
 if len(temp)!=0:  
 arr.append(temp)  
 print(arr)  
 return 0

**Результат выполнения работы:**



### Задача 2 (файл gen\_random.py)

Необходимо реализовать генератор gen\_random(количество, минимум, максимум), который последовательно выдает заданное количество случайных чисел в заданном диапазоне от минимума до максимума, включая границы диапазона. Пример:

gen\_random(5, 1, 3) должен выдать 5 случайных чисел в диапазоне от 1 до 3, например 2, 2, 3, 2, 1

Шаблон для реализации генератора:

# Пример:

# gen\_random(5, 1, 3) должен выдать выдать 5 случайных чисел

# в диапазоне от 1 до 3, например 2, 2, 3, 2, 1

# Hint: типовая реализация занимает 2 строки

def gen\_random(num\_count, begin, end):

pass

# Необходимо реализовать генератор

**Текст программы:**

import random  
  
  
def gen\_random(num,min,max):  
 my\_list = []  
 for iter in range (num):  
 my\_list.append(random.randint(min,max))  
 return my\_list

**Результат выполнения работы: **

**Задача 3 (файл unique.py)**

* Необходимо реализовать итератор Unique(данные), который принимает на вход массив или генератор и итерируется по элементам, пропуская дубликаты.
* Конструктор итератора также принимает на вход именованный bool-параметр ignore\_case, в зависимости от значения которого будут считаться одинаковыми строки в разном регистре. По умолчанию этот параметр равен False.
* При реализации необходимо использовать конструкцию \*\*kwargs.
* Итератор должен поддерживать работу как со списками, так и с генераторами.
* Итератор не должен модифицировать возвращаемые значения.

Пример:

data = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2]

Unique(data) будет последовательно возвращать только 1 и 2.

data = gen\_random(10, 1, 3)

Unique(data) будет последовательно возвращать только 1, 2 и 3.

data = [‘a’, ‘A’, ‘b’, ‘B’, ‘a’, ‘A’, ‘b’, ‘B’]

Unique(data) будет последовательно возвращать только a, A, b, B.

Unique(data, ignore\_case=True) будет последовательно возвращать только a, b.

Шаблон для реализации класса-итератора:

# Итератор для удаления дубликатов

class Unique(object):

def \_\_init\_\_(self, items, \*\*kwargs):

# Нужно реализовать конструктор

# В качестве ключевого аргумента, конструктор должен принимать bool-параметр ignore\_case,

# в зависимости от значения которого будут считаться одинаковыми строки в разном регистре

# Например: ignore\_case = True, Aбв и АБВ - разные строки

# ignore\_case = False, Aбв и АБВ - одинаковые строки, одна из которых удалится

# По-умолчанию ignore\_case = False

pass

def \_\_next\_\_(self):

# Нужно реализовать \_\_next\_\_

pass

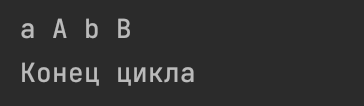
def \_\_iter\_\_(self):

return self

**Текст программы:**

class Unique:  
 def \_\_init\_\_(self, items, ignore\_case=False, \*\*kwargs):  
 self.items = items  
 self.ignore\_case = ignore\_case  
 self.set = set()  
 self.index = 0  
  
 def \_\_next\_\_(self):  
 # Если игнорируем верхний регистр, то пробегаемся по списку и приводим всё к нижнему регистру.  
 if self.ignore\_case:  
 for i, elem in enumerate(self.items):  
 if type(elem) is str:  
 self.items[i] = elem.lower()  
 while True:  
 if self.index >= len(self.items):  
 raise StopIteration  
 else:  
 current = self.items[self.index]  
 self.index += 1  
 # если текущего числа ещё не было, добавляем и возвращаем.  
 if current not in self.set:  
 self.set.add(current)  
 return current  
  
 def \_\_iter\_\_(self):  
 return self

**Результат выполнения программы:**

****

**Задача 4 (файл sort.py)**

Дан массив 1, содержащий положительные и отрицательные числа. Необходимо **одной строкой кода** вывести на экран массив 2, которые содержит значения массива 1, отсортированные по модулю в порядке убывания. Сортировку необходимо осуществлять с помощью функции sorted. Пример:

data = [4, -30, 30, 100, -100, 123, 1, 0, -1, -4]

Вывод: [123, 100, -100, -30, 30, 4, -4, 1, -1, 0]

Необходимо решить задачу двумя способами:

1. С использованием lambda-функции.
2. Без использования lambda-функции.

Шаблон реализации:

data = [4, -30, 100, -100, 123, 1, 0, -1, -4]

if \_\_name\_\_ == '\_\_main\_\_':

result = ...

print(result)

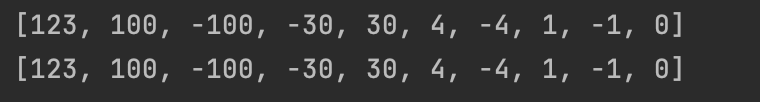
result\_with\_lambda = ...

print(result\_with\_lambda)

**Текст программы**

# The fourth task  
# with lambda-function  
 result\_with\_lambda = [4, -30, 30, 100, -100, 123, 1, 0, -1, -4]  
 print(sorted(result\_with\_lambda, key=lambda el: abs(el), reverse=True))  
  
# without lambda-function  
 result = [4, -30, 30, 100, -100, 123, 1, 0, -1, -4]  
 print(sorted(result\_with\_lambda, key=abs, reverse=True))

**Результаты выполнения программы**

****

**Задача 5 (файл print\_result.py)**

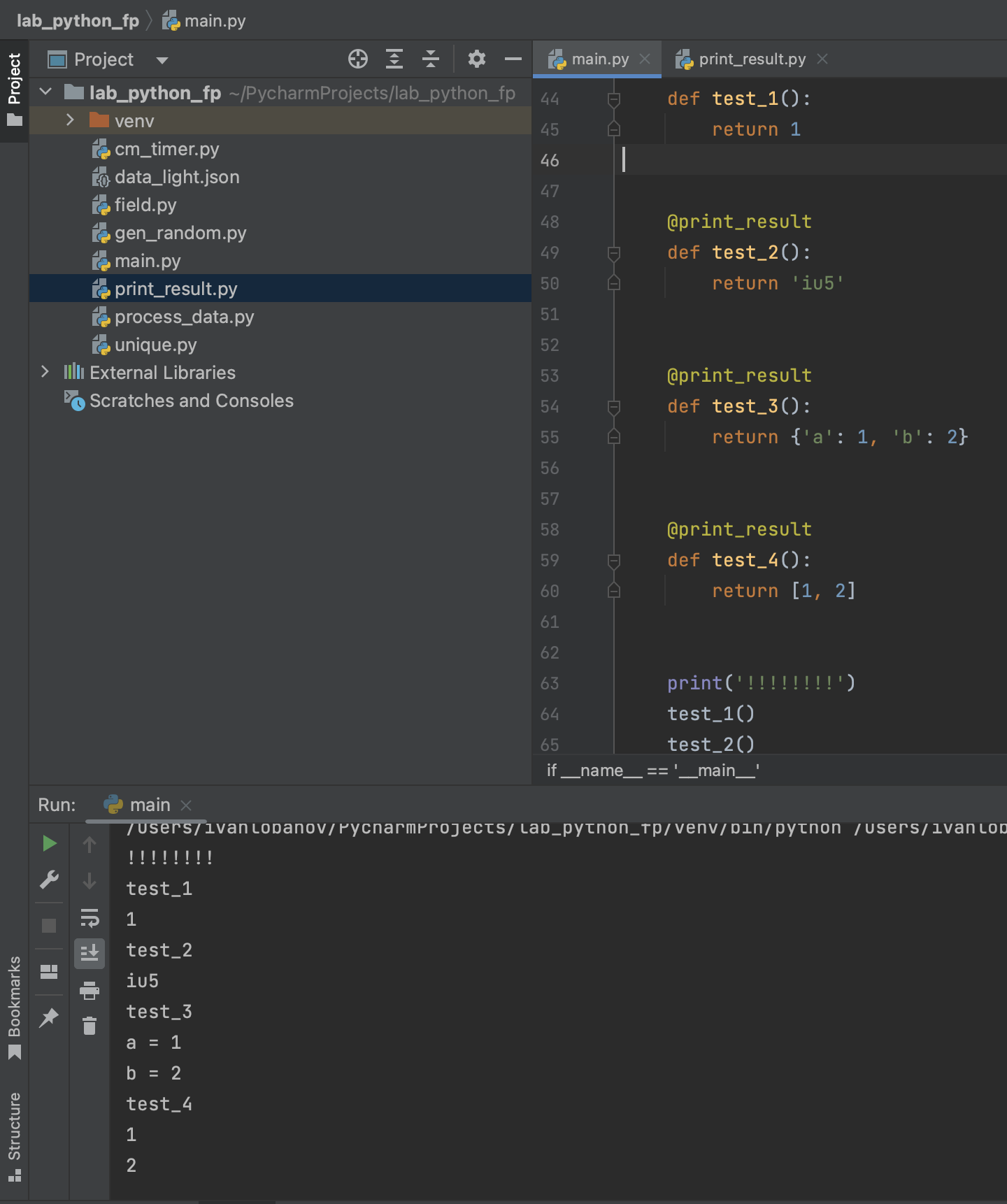
Необходимо реализовать декоратор print\_result, который выводит на экран результат выполнения функции.

* Декоратор должен принимать на вход функцию, вызывать её, печатать в консоль имя функции и результат выполнения, после чего возвращать результат выполнения.
* Если функция вернула список (list), то значения элементов списка должны выводиться в столбик.
* Если функция вернула словарь (dict), то ключи и значения должны выводить в столбик через знак равенства.

**Текст программы:**

def print\_result(test):  
 def wrapper(\*args):  
 print(test.\_\_name\_\_)  
 test(\*args)  
 arr = test(\*args)  
 if type(arr) == list:  
 for iter in arr:  
 print(iter)  
 elif type(arr) == dict:  
 for iter in arr.keys():  
 print(f"{iter} = {arr[iter]}")  
 else:  
 print(test())  
 return arr  
 return wrapper

**Результат выполнения:**

****

**Задача 6 (файл cm\_timer.py)**

Необходимо написать контекстные менеджеры cm\_timer\_1 и cm\_timer\_2, которые считают время работы блока кода и выводят его на экран. Пример:

with cm\_timer\_1():

sleep(5.5)

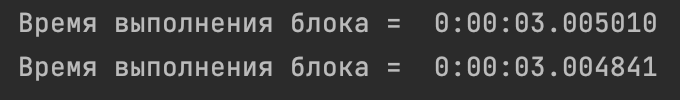
После завершения блока кода в консоль должно вывестись time: 5.5 (реальное время может несколько отличаться).

cm\_timer\_1 и cm\_timer\_2 реализуют одинаковую функциональность, но должны быть реализованы двумя различными способами (на основе класса и с использованием библиотеки contextlib).

**Текст программы:**

from datetime import datetime  
from time import sleep  
from contextlib import contextmanager  
  
  
class cm\_timer\_1:  
 def \_\_enter\_\_(self):  
 self.start\_time = datetime.now()  
 return self  
  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 print("Время выполнения блока = ", datetime.now() - self.start\_time)  
  
  
@contextmanager  
def cm\_timer\_2():  
 start\_time = datetime.now()  
 yield None  
 print("Время выполнения блока = ", datetime.now() - start\_time)

**Результат выполнения:**

****

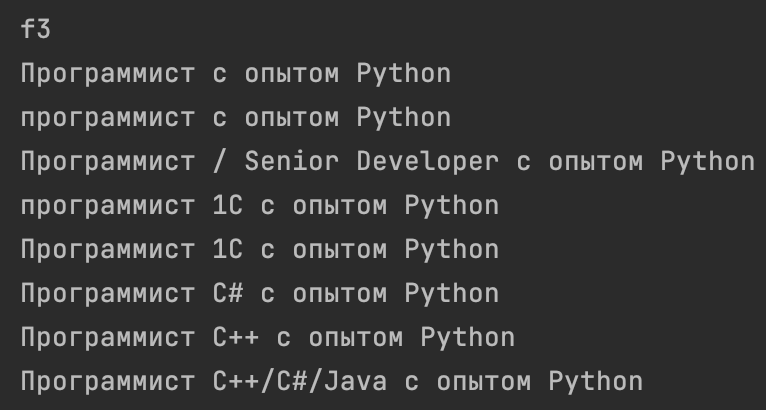
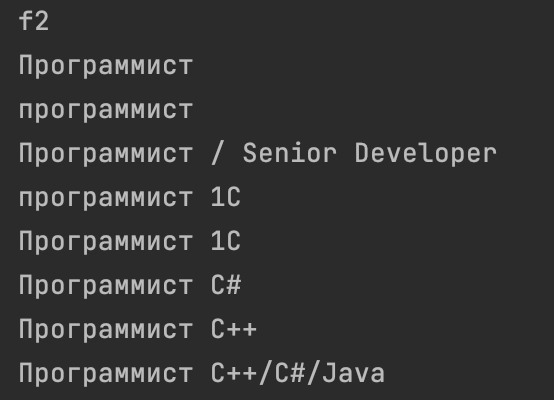
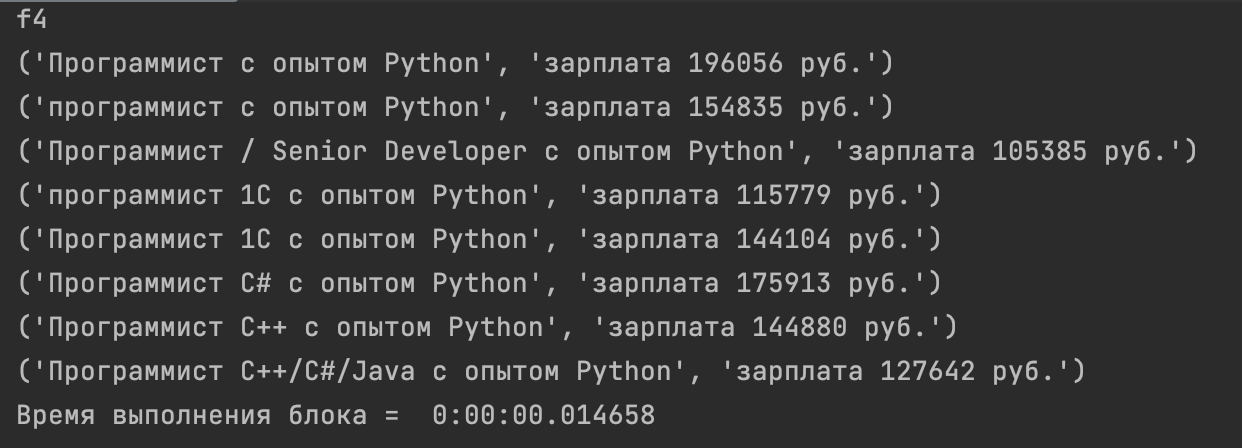
### Задача 7 (файл process\_data.py)

* В предыдущих задачах были написаны все требуемые инструменты для работы с данными. Применим их на реальном примере.
* В файле [data\_light.json](https://github.com/ugapanyuk/BKIT_2021/blob/main/notebooks/fp/files/data_light.json) содержится фрагмент списка вакансий.
* Структура данных представляет собой список словарей с множеством полей: название работы, место, уровень зарплаты и т.д.
* Необходимо реализовать 4 функции - f1, f2, f3, f4. Каждая функция вызывается, принимая на вход результат работы предыдущей. За счет декоратора @print\_result печатается результат, а контекстный менеджер cm\_timer\_1 выводит время работы цепочки функций.
* Предполагается, что функции f1, f2, f3 будут реализованы в одну строку. В реализации функции f4 может быть до 3 строк.
* Функция f1 должна вывести отсортированный список профессий без повторений (строки в разном регистре считать равными). Сортировка должна игнорировать регистр. Используйте наработки из предыдущих задач.
* Функция f2 должна фильтровать входной массив и возвращать только те элементы, которые начинаются со слова “программист”. Для фильтрации используйте функцию filter.
* Функция f3 должна модифицировать каждый элемент массива, добавив строку “с опытом Python” (все программисты должны быть знакомы с Python). Пример: Программист C# с опытом Python. Для модификации используйте функцию map.
* Функция f4 должна сгенерировать для каждой специальности зарплату от 100 000 до 200 000 рублей и присоединить её к названию специальности. Пример: Программист C# с опытом Python, зарплата 137287 руб. Используйте zip для обработки пары специальность — зарплата.

**Текст программы:**

from print\_result import print\_result  
from cm\_timer import cm\_timer\_1  
from gen\_random import gen\_random  
from unique import Unique  
import json  
import sys  
  
path = '/Users/ivanlobanov/PycharmProjects/lab\_python\_fp/data\_light.json'  
with open(path) as f:  
 data = json.load(f)  
  
  
@print\_result  
def f1(arg):  
 return sorted(list(set([el['job-name'] for el in arg])), key=lambda x: x.lower())  
  
  
@print\_result  
def f2(arg):  
 return list(filter(lambda text: (text.split())[0].lower() == 'программист', arg))  
  
  
@print\_result  
def f3(arg):  
 return list(map(lambda lst: lst + ' с опытом Python', arg))  
  
  
@print\_result  
def f4(arg):  
 return list(zip(arg, ['зарплата ' + str(el) + ' руб.' for el in gen\_random(len(arg), 100000, 200000)]))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with cm\_timer\_1():  
 f4(f3(f2(f1(data))))

**Результат выполнения:**

****