Московский государственный технический университет им. Н.Э. Баумана

Факультет «Информатика и системы управления»
Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»

Отчет по лабораторной работе №3

Выполнил:
Студент группы ИУ5-33Б
Левоквич Леонид

Проверил:
Преподаватель каф. ИУ5
Гапанюк Ю. Е.

Москва, 2025 г.

Задание:

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
```

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
```

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, Абв и АБВ - разные строки
```

```
        #           ignore_case = False, Абв 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
```

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)
```

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.

Шаблон реализации:

```
# Здесь должна быть реализация декоратора

@print_result
```

```
def test_1():
    return 1


@print_result
def test_2():
    return 'iu5'


@print_result
def test_3():
    return {'a': 1, 'b': 2}


@print_result
def test_4():
    return [1, 2]


if __name__ == '__main__':
    print('!!!!!!!!')
    test_1()
    test_2()
    test_3()
    test_4()
```

Результат выполнения:

```
test_1
1
test_2
iu5
test_3
a = 1
b = 2
test_4
1
2
```

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).

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 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.

Шаблон реализации:

```
import json
import sys
# Сделаем другие необходимые импорты

path = None

# Необходимо в переменную path сохранить путь к файлу, который был передан при запуске сценария

with open(path) as f:
    data = json.load(f)

# Далее необходимо реализовать все функции по заданию, заменив `raise NotImplemented`
# Предполагается, что функции f1, f2, f3 будут реализованы в одну строку
# В реализации функции f4 может быть до 3 строк

@print_result
def f1(arg):
    raise NotImplemented



@print_result
```

```
def f2(arg):
    raise NotImplemented


@print_result
def f3(arg):
    raise NotImplemented


@print_result
def f4(arg):
    raise NotImplemented


if __name__ == '__main__':
    with cm_timer_1():
        f4(f3(f2(f1(data))))
```

Листинг программы:

__init__.py

```
"""
Пакет lab_python_fp - задачи по функциональному программированию
"""

from .field import field
from .gen_random import gen_random
from .unique import Unique
from .print_result import print_result
from .cm_timer import cm_timer_1, cm_timer_2

__all__ = ['field', 'gen_random', 'Unique', 'print_result', 'cm_timer_1',
'cm_timer_2']
```

cm_timer.py

```
"""
Задача 6: Контекстные менеджеры для замера времени
"""

import time
from contextlib import contextmanager


# Способ 1: На основе класса
class cm_timer_1:
    """
    Контекстный менеджер для замера времени (реализация через класс).
    """

    def __enter__(self):
        """Вход в контекст - запоминаем время начала."""
        self._start_time = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Выход из контекста - вычисляем и выводим время."""
        elapsed_time = time.time() - self._start_time
        print(f"time: {elapsed_time:.6f}")
        return False  # Не подавляем исключения


# Способ 2: С использованием contextlib
```

```python
@contextmanager
def cm_timer_2():
    """
    Контекстный менеджер для замера времени (реализация через contextlib).
    """
    start_time = time.time()
    try:
        yield
    finally:
        elapsed_time = time.time() - start_time
        print(f"time: {elapsed_time:.6f}")


if __name__ == '__main__':
    print("=" * 50)
    print("Тестирование контекстных менеджеров")
    print("=" * 50)

    print("\nТест cm_timer_1 (на основе класса):")
    print("Выполняем sleep(0.5)...")
    with cm_timer_1():
        time.sleep(0.5)

    print("\nТест cm_timer_2 (с использованием contextlib):")
    print("Выполняем sleep(0.3)...")
    with cm_timer_2():
        time.sleep(0.3)

    print("\nТест с вычислениями:")
    print("Вычисляем сумму чисел от 1 до 1000000...")
    with cm_timer_1():
        result = sum(range(1, 1000001))
    print(f"Результат: {result}")

    print("\nТест с генератором:")
    from .gen_random import gen_random

    print("Генерируем 100000 случайных чисел...")
    with cm_timer_2():
        numbers = list(gen_random(100000, 1, 1000))
    print(f"Сгенерировано {len(numbers)} чисел")
```

field.py

```python
"""
Задача 1: Генератор field
"""


# 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):
    """
    Генератор, последовательно выдающий значения ключей словаря.

    Args:
        items: список словарей
        *args: ключи для извлечения

    Yields:
        Если один ключ - значение поля
```

```
        Если несколько ключей - словарь с указанными полями
    """
    assert len(args) > 0

    for item in items:
        if len(args) == 1:
            # Один аргумент - выдаем только значение
            key = args[0]
            value = item.get(key)
            if value is not None:
                yield value
        else:
            # Несколько аргументов - выдаем словарь
            result = {}
            for key in args:
                value = item.get(key)
                if value is not None:
                    result[key] = value
            # Пропускаем элемент, если все поля None
            if result:
                yield result


if __name__ == '__main__':
    print("=" * 50)
    print("Тестирование генератора field")
    print("=" * 50)

    goods = [
        {'title': 'Ковер', 'price': 2000, 'color': 'green'},
        {'title': 'Диван для отдыха', 'color': 'black'}
    ]

    print("\nИсходные данные:")
    for item in goods:
        print(f"  {item}")

    print("\nfield(goods, 'title'):")
    for item in field(goods, 'title'):
        print(f"  {item}")

    print("\nfield(goods, 'title', 'price'):")
    for item in field(goods, 'title', 'price'):
        print(f"  {item}")

    print("\nfield(goods, 'price'):")
    for item in field(goods, 'price'):
        print(f"  {item}")

    # Тест с None значениями
    goods_with_none = [
        {'title': 'Стол', 'price': None, 'color': 'white'},
        {'title': None, 'price': None, 'color': None},
        {'title': 'Стул', 'price': 500, 'color': 'brown'}
    ]

    print("\nТест с None значениями:")
    print("field(goods_with_none, 'title'):")
    for item in field(goods_with_none, 'title'):
        print(f"  {item}")

    print("\nfield(goods_with_none, 'title', 'price'):")
    for item in field(goods_with_none, 'title', 'price'):
        print(f"  {item}")
```

gen_random.py

```
"""
Задача 2: Генератор случайных чисел gen_random
```

```
"""

import random


# Пример:
# gen_random(5, 1, 3) должен 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):
    """
    Генератор случайных чисел.

    Args:
        num_count: количество чисел
        begin: минимальное значение (включительно)
        end: максимальное значение (включительно)

    Yields:
        Случайные числа в заданном диапазоне
    """
    for _ in range(num_count):
        yield random.randint(begin, end)


if __name__ == '__main__':
    print("=" * 50)
    print("Тестирование генератора gen_random")
    print("=" * 50)

    print("\ngen_random(5, 1, 3):")
    result = list(gen_random(5, 1, 3))
    print(f"  {result}")

    print("\ngen_random(10, 1, 100):")
    result = list(gen_random(10, 1, 100))
    print(f"  {result}")

    print("\ngen_random(3, -5, 5):")
    result = list(gen_random(3, -5, 5))
    print(f"  {result}")

    # Демонстрация работы с генератором напрямую
    print("\nПоследовательная итерация gen_random(5, 10, 20):")
    for i, num in enumerate(gen_random(5, 10, 20), 1):
        print(f"  Число {i}: {num}")
```

print_result.py

```
"""
Задача 5: Декоратор print_result
"""

from functools import wraps


def print_result(func):
    """
    Декоратор, который выводит на экран результат выполнения функции.

    - Печатает имя функции и результат выполнения
    - Если результат - список (list), значения выводятся в столбик
    - Если результат - словарь (dict), ключи и значения выводятся через '='

    Args:
        func: декорируемая функция
```

```
    Returns:
        Обёрнутая функция
    """

    @wraps(func)
    def wrapper(*args, **kwargs):
        # Вызываем функцию
        result = func(*args, **kwargs)

        # Печатаем имя функции
        print(func.__name__)

        # Печатаем результат в зависимости от типа
        if isinstance(result, list):
            # Список - выводим в столбик
            for item in result:
                print(item)
        elif isinstance(result, dict):
            # Словарь - выводим ключи и значения через '='
            for key, value in result.items():
                print(f"{key} = {value}")
        else:
            # Остальные типы - просто выводим
            print(result)

        return result

    return wrapper


@print_result
def test_1():
    return 1


@print_result
def test_2():
    return 'iu5'


@print_result
def test_3():
    return {'a': 1, 'b': 2}


@print_result
def test_4():
    return [1, 2]


if __name__ == '__main__':
    print("=" * 50)
    print("Тестирование декоратора print_result")
    print("=" * 50)
    print()
    print('!!!!!!!!!')
    test_1()
    test_2()
    test_3()
    test_4()
```

process_data.py

```
"""
Задача 7: Обработка данных с использованием всех инструментов
"""

import json
import sys
```

```python
import os

# Импортируем наши инструменты
from .field import field
from .gen_random import gen_random
from .unique import Unique
from .print_result import print_result
from .cm_timer import cm_timer_1

# Получаем путь к файлу данных
if len(sys.argv) > 1:
    path = sys.argv[1]
else:
    # Путь по умолчанию - в той же директории, что и скрипт
    path = os.path.join(os.path.dirname(__file__), '..', 'data_light.json')


@print_result
def f1(arg):
    """
    Выводит отсортированный список профессий без повторений.
    Строки в разном регистре считаются равными.
    Сортировка игнорирует регистр.
    """
    return sorted(Unique(field(arg, 'job-name'), ignore_case=True),
key=str.lower)


@print_result
def f2(arg):
    """
    Фильтрует входной массив, возвращая только элементы,
    начинающиеся со слова "программист" (без учета регистра).
    """
    return list(filter(lambda x: x.lower().startswith('программист'), arg))


@print_result
def f3(arg):
    """
    Модифицирует каждый элемент массива, добавив строку "с опытом Python".
    """
    return list(map(lambda x: f"{x} с опытом Python", arg))


@print_result
def f4(arg):
    """
    Генерирует для каждой специальности зарплату от 100000 до 200000 рублей
    и присоединяет её к названию специальности.
    """
    salaries = gen_random(len(arg), 100000, 200000)
    return [f"{job}, зарплата {salary} руб." for job, salary in zip(arg,
salaries)]


if __name__ == '__main__':
    print("=" * 60)
    print("Обработка данных из файла")
    print("=" * 60)

    # Проверяем наличие файла
    if not os.path.exists(path):
        print(f"\nОшибка: Файл '{path}' не найден!")
        print("\nСоздаём тестовый файл data_light.json...")

        # Создаём тестовые данные
        test_data = [
```

```
            {"job-name": "Программист Python"},
            {"job-name": "Программист Java"},
            {"job-name": "программист C#"},
            {"job-name": "ПРОГРАММИСТ JavaScript"},
            {"job-name": "Дизайнер"},
            {"job-name": "Менеджер проектов"},
            {"job-name": "Программист Python"},  # дубликат
            {"job-name": "Аналитик данных"},
            {"job-name": "Программист Go"},
            {"job-name": "DevOps инженер"}
        ]

        # Сохраняем тестовые данные
        test_path = os.path.join(os.path.dirname(__file__), '..',
'data_light.json')
        with open(test_path, 'w', encoding='utf-8') as f:
            json.dump(test_data, f, ensure_ascii=False, indent=2)

        path = test_path
        print(f"Тестовый файл создан: {path}\n")

    # Загружаем данные
    print(f"Загружаем данные из: {path}\n")
    with open(path, encoding='utf-8') as f:
        data = json.load(f)

    print(f"Загружено {len(data)} записей\n")
    print("-" * 60)

    # Выполняем цепочку функций с замером времени
    with cm_timer_1():
        f4(f3(f2(f1(data))))
```

sort.py

```
"""
Задача 4: Сортировка по модулю в порядке убывания
"""

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

if __name__ == '__main__':
    print("=" * 50)
    print("Тестирование сортировки по модулю")
    print("=" * 50)

    print(f"\nИсходные данные: {data}")
    print(f"Ожидаемый результат: [123, 100, -100, -30, 30, 4, -4, 1, -1, 0]")

    # Способ 1: Без использования lambda-функции
    result = sorted(data, key=abs, reverse=True)
    print(f"\n1. Без lambda (key=abs): {result}")

    # Способ 2: С использованием lambda-функции
    result_with_lambda = sorted(data, key=lambda x: abs(x), reverse=True)
    print(f"2. С lambda: {result_with_lambda}")

    # Дополнительная демонстрация
    print("\n" + "-" * 50)
    print("Демонстрация промежуточных вычислений:")
    print("-" * 50)
    for num in sorted(data, key=abs, reverse=True):
        print(f"  {num:>4} -> |{num}| = {abs(num)}")
```

unique.py

```
"""
Задача 3: Итератор Unique для удаления дубликатов
"""
```

```python
from .gen_random import gen_random


class Unique:
    """
    Итератор для удаления дубликатов.

    Поддерживает работу как со списками, так и с генераторами.
    """

    def __init__(self, items, **kwargs):
        """
        Конструктор итератора.

        Args:
            items: список или генератор
            **kwargs:
                ignore_case (bool): если True, строки в разном регистре
                                    считаются одинаковыми. По умолчанию False.
        """
        self._items = iter(items)
        self._ignore_case = kwargs.get('ignore_case', False)
        self._seen = set()

    def __next__(self):
        """
        Возвращает следующий уникальный элемент.

        Returns:
            Следующий уникальный элемент

        Raises:
            StopIteration: когда элементы закончились
        """
        while True:
            item = next(self._items)  # Может выбросить StopIteration

            # Определяем ключ для сравнения
            if self._ignore_case and isinstance(item, str):
                key = item.lower()
            else:
                key = item

            # Проверяем, видели ли мы этот элемент
            if key not in self._seen:
                self._seen.add(key)
                return item  # Возвращаем оригинальное значение

    def __iter__(self):
        return self


if __name__ == '__main__':
    print("=" * 50)
    print("Тестирование итератора Unique")
    print("=" * 50)

    # Тест 1: числа
    print("\nТест 1: Числа с дубликатами")
    data = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2]
    print(f"  Исходные данные: {data}")
    print(f"  Unique(data): {list(Unique(data))}")

    # Тест 2: работа с генератором
    print("\nТест 2: Работа с генератором gen_random(10, 1, 3)")
    result = list(Unique(gen_random(10, 1, 3)))
    print(f"  Unique(gen_random(10, 1, 3)): {result}")
```

```
    # Тест 3: строки без ignore_case
    print("\nТест 3: Строки без ignore_case")
    data = ['a', 'A', 'b', 'B', 'a', 'A', 'b', 'B']
    print(f"  Исходные данные: {data}")
    print(f"  Unique(data): {list(Unique(data))}")

    # Тест 4: строки с ignore_case=True
    print("\nТест 4: Строки с ignore_case=True")
    data = ['a', 'A', 'b', 'B', 'a', 'A', 'b', 'B']
    print(f"  Исходные данные: {data}")
    print(f"  Unique(data, ignore_case=True): {list(Unique(data,
ignore_case=True))}")

    # Тест 5: смешанные данные
    print("\nТест 5: Смешанные данные")
    data = ['Привет', 'привет', 'ПРИВЕТ', 'Мир', 'МИР']
    print(f"  Исходные данные: {data}")
    print(f"  Unique(data): {list(Unique(data))}")
    print(f"  Unique(data, ignore_case=True): {list(Unique(data,
ignore_case=True))}")

    # Тест 6: проверка, что значения не модифицируются
    print("\nТест 6: Проверка сохранения регистра при ignore_case=True")
    data = ['ABC', 'abc', 'Abc', 'DEF', 'def']
    print(f"  Исходные данные: {data}")
    result = list(Unique(data, ignore_case=True))
    print(f"  Unique(data, ignore_case=True): {result}")
    print(f"  (Сохранен оригинальный регистр первого вхождения)")
```