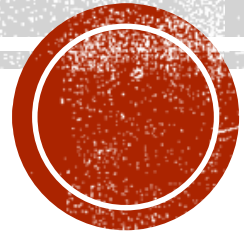
УЧЕБНАЯ ПРАКТИКА



Занятие 1

17 декабря 2018 г.

СТРУКТУРА ПРАКТИКИ

Темы:

- *Элементы функционального программирования*
- *Генератор псевдослучайных чисел (**PRNG**)*
- *Хеш-функции*

Практические задания

3. *Собственный **PRNG**, сравнительный анализ со стандартным генератором*
4. *Хеш-таблица с собственной хеш-функцией, анализ устойчивости к коллизиям*
5. *Алгоритм **SHA-1***

Результат

- Отчет о прохождении практики (половина)

РАСПРЕДЕЛЕНИЕ ТЕМ И ЗАДАНИЙ ПО ЗАНЯТИЯМ

17 декабря 2018 г.

- Элементы функционального программирования. Генератор псевдослучайных чисел (**PRNG**).
- Собственный **PRNG**, сравнительный анализ со стандартным генератором.

20 декабря 2018 г.

- Хеш-функции. Принципы работы и построения. Алгоритм SHA-1.
- Хеш-функция **SHA-1**.

25 декабря 2018 г.

- Примеры использования хеш-функций.
- Хеш-таблица с собственной хеш-функцией, анализ устойчивости к коллизиям.

ЭЛЕМЕНТЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

- объекты первого класса,
- функции высшего порядка,
- лямбда-функции,
- map/reduce,
- *итераторы*,*
- *генераторы*,*
- замыкания.

---

* не элементы ФП

ОБЪЕКТЫ ПЕРВОГО КЛАССА

Опр. 1: Объект называют «**объектом первого класса**», если он:

- может быть сохранен в переменной или структурах данных;
- может быть передан в функцию как аргумент;
- может быть возвращен из функции как результат;
- может быть создан во время выполнения программы.

Примеры «объектов», о классе которых можно говорить:

- числовые значения, объекты классов, функции, классы

ФУНКЦИИ ВЫСШЕГО ПОРЯДКА

Опр. 2: **Функция высшего порядка —** в программировании функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

Функции первого класса и функции высшего порядка – связанные, но различные понятия.

ЛЯМБДА-ФУНКЦИИ

Опр. 3: **Лямбда-функция —** это анонимная функция. Сразу после создания она:

- сохраняется в переменной,
- либо передается в качестве аргумента в функцию,
- либо возвращается из функции,
- либо вызывается.

Лямбда-функции не объявляются заранее, а создаются и сразу же используются тогда, когда в них есть необходимость.

MAP/REDUCE

Опр. 4: **map(func, list)** — это функция высшего порядка, которая:

- принимает на вход список объектов и функцию,
- применяет функцию к каждому элементу списка и сохраняет результаты в новом списке,
- возвращает новый полученный список.

`map` переводится как «отображение». Функция «отображает» элементы одного множества (списка) в элементы другого. По своей сути `map` ведет себя как табличная математическая функция.

MAP/REDUCE

Опр. 5: **reduce(func, list)** — это функция высшего порядка, которая:

- принимает на вход список объектов и функцию от двух параметров,
- выполняет свертку списка:
  - применяет функцию к первым двум элементам,
  - затем – к результату и третьему элементу,
  - затем – к результату и четвертому элементу,
  - …
- возвращает полученное в результате единственное значение.

ИТЕРАТОРЫ

Опр. 6: **Итератор —** это объект, который позволяет получить последовательный доступ к элементам другого объекта-коллекции.

Реализует следующий интерфейс:

- получить доступ к текущему объекту
- переместиться на следующий объект

Основное место использования итераторов – это цикл for.

ГЕНЕРАТОРЫ

Опр. 7: **Генератор —** это объект, реализующий класс итератора и который выдает (генерирует) значения в соответствии с каким-то алгоритмом.

Это не генераторы списков ([i for i in range(10)])!

Основное предназначение – перебрать «элементы» некоторого «списка», который по каким-то причинам нет возможности полностью держать в памяти.

Примеры:

- *генератор простых чисел (чисел Фибоначчи, …),*
- *генератор псевдо-случайных чисел,*
- *«генератор» строк из файла,*
- *асинхронный код и т.д.*

ЗАМЫКАНИЯ

Опр. 8: **Замыкание (closure)** — это функция, которая ссылается на переменные, объявленные вне тела этой функции и не лежащие в глобальной области видимости. Говорят, что она замкнута на свой лексический контекст.

В Питоне замыкание выглядит как функция, созданная внутри функции:

```
def mul(a):
    def helper(b):
        return a * b
    return helper

new_mul5 = mul(5)
new_mul5(2)
new_mul5(7)
```

ЗАМЫКАНИЯ

Опр. 8: **Замыкание (closure)** — это функция, которая ссылается на переменные, объявленные вне тела этой функции и не лежащие в глобальной области видимости. Говорят, что она замкнута на свой лексический контекст.

В Питоне замыкание выглядит как функция, созданная внутри функции:

```
def func(val):
    x = val
    def innerFunc():
        nonlocal x
        y = 1
        x += 1
        y += 1
        print(x, y)
    return innerFunc

x = func(5)
x()
x()
```

ЗАМЫКАНИЯ

Замыкание — это функция **вместе** с переменными, на которых она замкнута (которые сохраняются в памяти между вызовами функций).

	Существует пока выполняется программа	**Существует пока выполняется функция**
Глобальная область видимости	Глобальная переменная	–
Локальная область видимости	Замкнутая (статическая) переменная	Локальная переменная

PRNG

Опр. 9: **PRNG (Pseudo Random Number Generator)** — это функция, которая при каждом своем вызове возвращает «псевдослучайное» число.

Псевдослучайным оно называется, потому что в действительности это элемент детерминированной последовательности, в которой каждый элемент вычисляется на основе предыдущего (предыдущих). Например:

$$f(x_0) = x_1$$
$$f(x_1) = x_2$$
$$f(x_2) = x_3$$
$$f(x_3) = x_4$$
$$f(x_4) = x_5$$
$$f(x_5) = x_6$$
$$\dots$$

Первое значение (x_0) называется зерном (**seed**) генератора.

PRNG

Свойство 1: Последовательность, выдаваемая **PRNG**, является **детерминированной**.

При одинаковом зерне генератор будет выдавать одинаковую последовательность.

PRNG

Свойство 2: Последовательность, выдаваемая **PRNG**, является **периодической**.

Т.к. диапазон выдаваемых чисел конечный, то рано или поздно они начнут повторяться.

Хороший генератор выдает последовательность с максимальным периодом.

```
def gen_prng(seed):
    x = seed
    def prng():
        nonlocal x
        x = (x + 7) % 10
        return x
    return prng

my_prng = gen_prng(1)
vals = [my_prng() for i in range(10)]
print(vals)
```

```
[8, 5, 2, 9, 6, 3, 0, 7, 4, 1]
```

Вопрос: Чему равен максимально возможный период данного генератора?

ЛИНЕЙНЫЙ КОНГРУЭНТНЫЙ МЕТОД

Линейный конгруэнтный метод (Д.Г. Лемер, 1949 год)

$$x_{n+1} = (ax_n + c) \,\%\, m$$

Длина периода напрямую зависит от констант.

Для $m = 10, x_0 = a = c = 7$ получим последовательность 7,6,9,0,7,6,9,0,…

Длина периода равна m тогда и только тогда, когда:

- Числа **c** и **m** взаимно простые;
- **b=a-1** кратно **p** для каждого простого **p**, являющегося делителем **m**;
- **b** кратно **4**, если **m** кратно **4**.

Например, в стандарте языка Си: $a = 1103515245, c = 12345$

PRNG В РАЗНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Пример реализации **PRNG** в библиотеке языка Си:

```c
#define RAND_MAX 32767

static unsigned long int next = 1;

int rand(void)
{
  next = next * 1103515245 + 12345;
  return (unsigned int)(next/65536) % (RAND_MAX + 1);
}

void srand(unsigned int seed)
{
  next = seed;
}
```

PRNG в Питоне реализован с помощью вихря Мерсенна (**Mersenne twister**). Генерирует последовательности с очень большим периодом ($2^{19937} - 1$).

ЗАДАНИЕ №1

- Создать свой собственный PRNG с помощью **замыкания**.
- Вывести "плотность" распределения случайных значений и среднеквадратическую ошибку:
  - Функцию анализа реализовать как **функцию высшего порядка**.
  - Передать ей также стандартный PRNG и сравнить результаты.
- Подсчитать длину периода (см. далее).

Пример работы функции анализа (второй пример – вымышленный!):

```
evaluate(my_prng,4000)
Результат:  [476, 365, 253, 448, 257, 479, 563, 254, 452, 453]
Плотность:  [11.9, 9.1, 6.3, 11.2, 6.4, 11.9, 14.0, 6.3, 11.3, 11.3]
Среднеквадратическая ошибка:  2.630399209245623
Длина периода: 42553

evaluate(lambda:random.randint(0,9),4000)
Результат:  [400, 400, 400, 400, 400, 400, 400, 400, 400, 400]
Плотность:  [10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0]
Среднеквадратическая ошибка:  0.0
Длина периода: 4294967295
```

ЗАДАНИЕ №1

- **PRNG** должен выдавать числа в диапазоне от 0 до 9
- «Плотность» – это доля нулей, единиц, двоек и т.д. в сгенерированной последовательности, выраженная в процентах
- Среднеквадратическая ошибка определяет насколько «далеко» наш результат от равномерного распределения.
- Отклонение вычисляется как $\sqrt{\frac{1}{10}\sum_0^9 (d_i - x_i)^2}$, где
  - d_i - то, что должно получиться (в нашем примере **N / 10**)
  - x_i - то, что получилось (количество чисел **i**)
- Свой **PRNG** придумать как угодно и самостоятельно. Отклонение не учитывается при оценивании. При оценивании учитывается **оригинальность** (т.е. авторство) вашей функции.
- Линейный конгруэнтный метод использовать нельзя (см. далее).

ЗАДАНИЕ №1

Вид функции **evaluate()**:

```
from functools import reduce
import math

def evaluate(my_prng, tries):
    # Создаем список из 10 нулей - в нем мы будем считать частоту чисел
    F = [0 for i in range(10)]

    # Генерируем tries случайных чисел и считаем, сколько каких получилось.
    # Результат записываем в F
    for i in range(tries):
    # ...

    # Вычисляем плотность с помощью map и lambda
    density = list(map(lambda ...,F))

    # Вычисляем среднеквадратическую ошибку с помощью reduce и lambda
    s = math.sqrt(reduce(lambda x, y: ..., density, 0)/10)
    print("Результат: ", F)
    print("Плотность: ", density)
    print("Среднеквадратическая ошибка: ", s)
```