Федеральное государственное автономное образовательное учреждение высшего образования

«ОМСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Прикладная математика и фундаментальная информатика»

Практическое занятие №10

по дисциплине «Практикум по программированию»

на тему: «Основы ООП в Python»

Вариант №14

Выполнил

Студент гр. **ФИТ-212** **Курпенов К.И.**

группа Фамилия И,О. студента подпись

Принял:

Преподаватель **Моисеева Н.А.**

Фамилия И.О, преподавателя дата, подпись

Омск 2022

Задача 1.

Условие:

Квадратные уравнения – это уравнения вида: ax2+bx+c=0ax2+bx+c=0.

Функция вида f(x)=ax2+bx+cf(x)=ax2+bx+c называется квадратичной.

Напишите класс **Quadratic**, который поможет рассчитать некоторые характеристики такой функции.

Экземпляр класса инициализируется с аргументами – коэффициентами a, b и c (a≠0a≠0). Класс реализует методы:

---

branch – возвращает **up**, если ветви параболы направлены вверх, или **down**, если вниз;

x_sect – возвращает количество точек пересечения функции с осью OX;

sections – возвращает один или два набора координат точек пересечения с осью OX (если два, то в порядке возрастания x; сначала *x*, затем *y*) или *None*, если пересечений нет;

top – возвращает кортеж координат вершины параболы;

y_sect – возвращает точку пересечения графика функции с осью OY.

Все значения округлять не нужно.

Решение:

```python
class Quadratic:
    def __init__(self, a: float, b: float, c: float) -> None:
        self.a = a
        self.b = b
        self.c = c

        self.d = self.b**2 - 4 * self.a * self.c

        self.solution = []

        if self.d < 0:
            self.solution.append(None)
        elif self.d == 0:
            self.solution.append(-self.b / (2 * self.a))
        else:
            self.solution.append((-self.b + self.d**0.5) / (2 * self.a))
            self.solution.append((-self.b - self.d**0.5) / (2 * self.a))

    def branch(self) -> str:
        if self.a < 0:
            return "dowm"
        else:
            return "up"

    def sections(self) -> tuple:
        return tuple(self.solution)

    def top(self) -> tuple:
        x = -self.b / (2 * self.a)
        y = self.a * x * x + self.b * x + self.c
        return x, y

    def x_sect(self) -> int:
        if self.solution[0] is None:
            return 0
        elif len(self.solution) == 1:
            return 1
        else:
            return 2

    def y_sect(self) -> tuple:
        return 0, self.c


if __name__ == "__main__":
    quadratic = Quadratic(1, -6, 9)
    print(quadratic.x_sect())
    print(quadratic.branch())
    print(quadratic.top())
    print(quadratic.y_sect())
    print(quadratic.sections())

~
~
```

Задача 2.

Условие:

a) Реализуйте следующие классы.

- Студент (**Student**). Инициализируется с аргументами: имя (*name*) и название университета (*university*). Такие же имена имеют атрибуты класса. Кроме них, класс имеет атрибут *курс*, который при инициализации равен 1.

Имеет методы:

get_name() – возвращает имя;

get_university() – возвращает название университета;

get_year() – возвращает курс обучения;

study() – увеличивает курс обучения на 1, пока не станет 6, дальше не увеличивается.

- Сотрудник (**Employee**). Инициализируется с аргументами: имя (*name*) и название компании (*company*). Такие же имена имеют атрибуты класса. Кроме того имеет аргумент, показывающий положение в компании, один элемент из списка *junior, middle, senior, lead*. При инициализации первый (*junior*).

Имеет методы:

get_name() – возвращает имя;

get_company() – возвращает название компании;

work() – увеличивает положение в компании в соответствии со списком роста, пока не достигнет последней позиции, дальше изменения не происходят;

get_position() – возвращает положение в компании.

- Просто человек (**Human**). Инициализируется с одним аргументом – именем (*name*), совпадающим по названию с аргументом. Умеет только возвращать своё имя методом *get_name()*.

b) Трансцендентное число **e** является основанием натурального логарифма, используется во многих областях математики и физики, поэтому важно знать его значение с высокой точностью. Есть формулы, по которым можно получить практически любую точность, если достаточно долго считать.

Воспользуемся формулой разложения числа **e** в ряд:

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Напишите класс **EulerNumber**, экземпляр которого инициализируется с аргументом **n** – количеством слагаемых в разложении числа **e** в ряд.

Класс имеет единственный метод **get_number()**, который может вызваться без аргументов, тогда возвращается число e в 1-й степени, или с аргументом **x** – степенью числа e.

Решение:

```python
from math import factorial

class Student:
    def __init__(self, name: str, university: str) -> None:
        self.name = name
        self.university = university
        self.year = 1

    def getName(self) -> str:
        return self.name

    def getUniversity(self) -> str:
        return self.university

    def getYear(self) -> int:
        return self.year

    def study(self) -> None:
        if self.year < 6:
            self.year += 1

class Employee:
    def __init__(self, name: str, company: str) -> None:
        self.positions = ["junior", "middle", "senior", "lead"]

        self.name = name
        self.company = company
        self.status = (0, self.positions[0])

    def getName(self) -> str:
        return self.name

    def getCompany(self) -> str:
        return self.company

    def getPosition(self) -> str:
        return self.status[1]

    def work(self) -> None:
        if self.status[0] < 4:
            self.status = (self.status[0]+1,self.positions[self.status[0] + 1])

class EulerNumber:
    def __init__(self, n: int) -> None:
        self.n = n

        self.e = 1

        for i in range(1, n):
            self.e += i / factorial(i)

    def getNumber(self, x=1) -> float:
        return self.e**x

```

Задача 3.

Условие:

a) Хорошо бы автоматизировать подсчет лайков. Напишите базовый класс **Liked**, который подсчитывает количество эмодзи разных типов, например, таких:

:)) (;) :(;(

При инициализации экземпляр класса принимает произвольное количество строк, а при вызове метода *likes()* возвращается словарь: ключи – эмодзи, значения – количество штук. Если что-то не встречалось, то ключ не создается.

Производный от базового класс **MiMiMi** считает лайки только в строках с котиками (*cat, kitten*). Не создавайте метод *likes* в производном классе, используйте метод базового класса!

b) В жизненном цикле комара есть несколько стадий. Если немного упростить, то это стадия личинки и взрослой особи. Личинка живет в воде (*in water*) и питается водорослями (*algae*). Самец комара живет на суше (*on land*) и питается нектаром (*nectar*). Самка тоже живет на суше, но предпочитает кусать людей и животных и питаться их кровью (*blood*).

Напишите класс **Mosquito** (Комар), который инициализируется с одним аргументом – возрастом. Имеет метод __str__, который возвращает строку *Mosquito, <age> days*.

От этого класса отнаследуйте классы:

MaleMosquito (самец комара)

FemaleMosquito (самка комара)

А класс **MosquitoLarva** (личинка) наследется от двух предыдущих.

В производных классах самца, самки и личинки переопределите методы __str__ (или в исходном классе определите метод так, чтобы его не пришлось переопределять), они должны возвращать свои названия вместо общего названия вида.

У всех трех классов добавляются атрибуты: *feed* (чем питается) и *lives* (где живет).

Самцы хорошо слышат (метод **hearing**, возвращает строку *I hear and see everything <где живет>*).

Самки издают писк (метод **squeak**, возвращает строку *The thin squeak of a mosquito after eating <что ест>*).

c) В физике часто так случается, что разные вроде бы явления описываются математически одинаковыми зависимостями. Например, колебания и движение по окружности. Или колебания в механической системе – пружинный, математический или физический маятник – или электромагнитные колебания.

Напишите базовый класс **Oscillations** (Колебания), принимающий в качестве аргумента амплитуду (A) колебаний. Имеет метод:

__repr__ – возвращает строку имя класса и в скобках все аргументы через запятую и пробел.

Унаследуйте от базового класса следующие классы:

SpringPendulum

MathPendulum

EMPendulum

Каждый из производных классов кроме амплитуды принимает две характеристики колебательной системы:

Решение:

```python
from math import sqrt, sin, pi


class Liked:
    def __init__(self, *args) -> None:
        self.data = []

        for arg in args:
            for line in arg:
                self.data.append(line)

    def likes(self) -> dict:
        self.emojis = [":)", ";)", ")", ":(", ";(", "("]
        output = dict()

        for line in self.data:
            for emoji in self.emojis:
                count = line.count(emoji)
                if count:
                    if emoji in output:
                        output[emoji] += count
                    else:
                        output[emoji] = count

        return output


class MiMiMi(Liked):
    def __init__(self, *args) -> None:
        super().__init__(*args)

    def likes(self) -> dict:
        self.emojis = [":)", ";)", ")", ":(", ";(", "("]
        output = dict()

        for line in self.data:
            if ["cat", "kitten"] in line:
                for emoji in self.emojis:
                    count = line.count(emoji)
                    if count:
                        if emoji in output:
                            output[emoji] += count
                        else:
                            output[emoji] = count

        return output


```

```python
class Mosquito:
    def __init__(self, age: int) -> None:
        self.age = age

    def __str__(self) -> str:
        return f"Mosquito, {self.age} age"


class MaleMosquito(Mosquito):
    def __init__(self, age: int, lives: str) -> None:
        self.age = age
        self.lives = lives

    def __str__(self) -> str:
        return f"I hear and see everything {self.lives}"


class FemaleMosquito(Mosquito):
    def __init__(self, age: int, feed: str) -> None:
        self.age = age
        self.feed = feed

    def __str__(self) -> str:
        return f"The thin squeak of a mosquito after eating {self.feed}"


class MosquitoLarva(Mosquito):
    def __str__(self) -> str:
        return f""


class Oscillations:
    def __init__(self, a: int) -> None:
        self.a = a


class SpringPendulum(Oscillations):
    def __init__(self, a: int, m: int, k: int) -> None:
        self.a = a
        self.m = m
        self.k = k

    def period(self) -> float:
        return 2 * pi * sqrt(self.m / self.k)

    def cyclic_frequeny(self, w: int) -> float:
        return 2 * pi / w

    def __str__(self, t: int, w: int) -> str:
        return f"X = {self.a * sin(w * t)}"

```

```
100
101 class MathPendulum(Oscillations):
102     def __init__(self, a: int, l: int, g: int) -> None:
103         self.a = a
104         self.l = l
105         self.g = g
106
107     def period(self) -> float:
108         return 2 * pi * sqrt(self.l / self.g)
109
110     def cyclic_frequeny(self, w: int) -> float:
111         return 2 * pi / w
112
113     def __str__(self, t: int, w: int) -> str:
114         return f"X = {self.a * sin(w * t)}"
115
116
117 class EMPendulum(Oscillations):
118     def __init__(self, a: int, l: int, c: int) -> None:
119         self.a = a
120         self.l = l
121         self.c = c
122
123     def period(self) -> float:
124         return 2 * pi * sqrt(self.l * self.c)
125
126     def cyclic_frequeny(self, w: int) -> float:
127         return 2 * pi / w
128
129     def __str__(self, t: int, w: int) -> str:
130         return f"X = {self.a * sin(w * t)}"
131
132
133 if __name__ == "__main__":
134     liked = Liked(["Hi, Kuat! :)", "Well ;)", "How are you?))"])
135     print(liked.likes())
136
~
~
```

Задача 4.

Условие:

a) Есть корзина с грушами и несколько детей. Нужно поделить груши так, чтобы никому из детей не было обидно, т.е. поровну.

Напишите класс **PearsBasket**, экземпляр которого при инициализации получает целое число – количество груш в корзине.

В классе должны быть методы:

- *pb // n* – деление нацело, возвращает список объектов класса со значениями количества груш в каждой корзинке, если есть остаток – он должен находиться в дополнительной последней корзинке.
- *pb % n* – получение остатка от деления, возвращает число: остаток от деления.
- *pb_1 + pb_2* – складываются две корзинки, получается новая корзина;
- *pb_1 - n* – число вычитается из корзинки, если там есть такое количество груш; если нет – вычитается сколько есть, остается 0;

- *__str__* – возвращает количество груш в корзине;
- *__repr__* – возвращает строку в формате **PearsBasket(<количество>)**.

b) Количество чисел в модульной арифметике ограничено, как на циферблате часов, например. Как только часовая стрелка доходит до 12, следующее число не 13, а 1.

В арифметике по модулю 12 почти так же, только числа там меняются от 0 до 11, а не от 1 до 12, как на циферблате часов. То есть они означают остатки от деления на 12.

Если взять другой модуль, то и остатки будут от деления на это число.

В модульной арифметике нет отрицательных чисел.

Напишите класс **ModularArithmetic**, экземпляр *ma* которого при инициализации принимает два числа – значение и модуль.

Должен реализовывать методы:

ma(x) – при вызове возвращает целую часть от деления числа x на модуль;

ma_1 + ma_2 – возвращает экземпляр класса, полученный при сложении чисел по модулю (числа складываются в одной и той же модульной арифметике);

ma_1 - ma_2 – вычитание аналогично;

ma_1 >>>> n – сдвиг вправо на n – то же, что сложить число с n;

ma_1 <<<< n – сдвиг влево – то же, что вычесть из числа n;

__str__ и *__repr__* – возвращают число в виде *<число>(<модуль>)*.

Решение:

```python
class PearsBasket:
    def __init__(self, n: int) -> None:
        self.n = n

    def __add__(self, other):
        return PearsBasket(self.n + other.n)

    def __sub__(self, other):
        return PearsBasket(self.n - other.n)

    def __mul__(self, other):
        return PearsBasket(self.n * other.n)

    def __floordiv__(self, other):
        return PearsBasket(self.n // other.n)

    def __str__(self) -> str:
        return f"{self.n}"

    def __repr__(self) -> str:
        return f"PearsBasket({self.n})"


class ModularArithmetic:
    def __init__(self, x: int, m: int) -> None:
        self.x = x
        self.m = m

    def __call__(self, m) -> int:
        return self.x % m

    def __add__(self, other):
        return ModularArithmetic((self.x + other.x) % self.m, self.m)

    def __sub__(self, other):
        return ModularArithmetic((self.x - other.x) % self.m, self.m)

    def __lshift__(self, n):
        return ModularArithmetic((self.x - n) % self.m, self.m)

    def __rshift__(self, n):
        return ModularArithmetic((self.x - n) % self.m, self.m)

    def __str__(self) -> str:
        return f"{self.x % self.m}({self.m})"

    def __repr__(self) -> str:
        return f"{self.x % self.m}({self.m})"
```