Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Новосибирский национальный исследовательский государственный университет»

КЛАССЫ

ПОНЯТИЕ КЛАССА

НАСЛЕДОВАНИЕ

ИСКЛЮЧЕНИЯ

Гаврилов Денис Андреевич, преподаватель кафедры СИ ФИТ НГУ

Объектно-ориентированный подход.

Понятие класса и объекта.

Определение класса.

Атрибуты класса.

Методы класса.

Приватные элементы класса.

ПОНЯТИЕ КЛАССА

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы (используемых ею структур данных) в виде совокупности *объектов*, каждый из которых является экземпляром определённого *класса*, а классы образуют *иерархию наследования*.

Важно: выделяя *объекты* в рамках некоторой *предметной области*, программист абстрагируется (отвлекается) от большинства их свойств, преимущественно концентрируясь на тех из них, которые существенны для конкретной задачи.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

Важно: выделяя *объекты* в рамках некоторой *предметной области*, программист абстрагируется (отвлекается) от большинства их свойств, преимущественно концентрируясь на тех из них, которые существенны для конкретной задачи.

Задача А:

- Мощность
- Расход топлива
- Тип двигателя
- Тип коробки передач
- Цвет

Задача Б:

- Цвет
- Объем багажника
- Дизайн салона
- Зимний пакет
- Тип коробки передач

ПОНЯТИЕ КЛАССА

В рамках одного *класса* могут существовать различные конкретные *объекты – экземпляры класса*:

- Цвет: черный
- Двигатель: бензиновый
- Коробка передач: автомат

- Цвет: синий
- Двигатель: дизельный
- Коробка передач: автомат

- Цвет: красный
- Двигатель: бензиновый
- Коробка передач: механика

ОПРЕДЕЛЕНИЕ КЛАССА

```
class Car():

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.working = False

    def start(self):
        if self.working == False:
            self.working = True
            print(f"{self.engine} двигатель запущен")

    def shut_down(self):
        if self.working == True:
            self.working = False
            print(f"{self.engine} двигатель заглушен")
```

- Для определения класса в Python воспользуемся оператором *class*;
- Дальнейшее описание тела класса во многом схоже с описанием тела функции;
- В частности, тело класса считается законченным на том выражении, перед которым стоит наименьшее количество пробелов;

ОПРЕДЕЛЕНИЕ КЛАССА

```
class Car():

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.working = False

    def start(self):
        if self.working == False:
            self.working = True
            print(f"{self.engine} двигатель запущен")

    def shut_down(self):
        if self.working == True:
            self.working = False
            print(f"{self.engine} двигатель заглушен")
```

- Определяем *конструктор класса*;
- *Конструктор* – метод (собственная функция класса), используемая для получения объекта данного класса;
- В конструкторе можно, в том числе, задать начальные значения атрибутов класса;

ОПРЕДЕЛЕНИЕ КЛАССА

```
class Car():

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.working = False

    def start(self):
        if self.working == False:
            self.working = True
            print(f"{self.engine} двигатель запущен")

    def shut_down(self):
        if self.working == True:
            self.working = False
            print(f"{self.engine} двигатель заглушен")
```

- Определяем *методы класса*;
- Методы (собственные функции класса) служат для описания поведения объектов данного класса;
- Метод «завести машину» - если двигатель не запущен, запускает его;

ОПРЕДЕЛЕНИЕ КЛАССА

```
class Car():

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.working = False

    def start(self):
        if self.working == False:
            self.working = True
            print(f"{self.engine} двигатель запущен")

    def shut_down(self):
        if self.working == True:
            self.working = False
            print(f"{self.engine} двигатель заглушен")
```

- Определяем *методы класса*;
- Методы (собственные функции класса) служат для описания поведения объектов данного класса;
- Метод «заглушить машину» - если двигатель запущен, останавливает его;

СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА

```
my_car = Car("Красный", "Бензиновый", "Автомат")

print(f"Цвет моей машины {my_car.color}.")

my_car.start()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")

my_car.shut_down()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")
```

- После определения *класса*, мы можем вызывать его *конструктор* для формирования конкретных экземпляров класса - *объектов*;
- Вызов *конструктора* осуществляется по имени *класса*.

СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА

```
my_car = Car("Красный", "Бензиновый", "Автомат")

print(f"Цвет моей машины {my_car.color}.")

my_car.start()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")

my_car.shut_down()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")
```

- После того, как объект получен, мы можем обратиться к его атрибутам напрямую;

- В данном примере мы выводим на печать текст, включив в него значение атрибута *«color»* объекта *«my_car»*.

СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА

```
my_car = Car("Красный", "Бензиновый", "Автомат")

print(f"Цвет моей машины {my_car.color}.")

my_car.start()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")

my_car.shut_down()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")
```

- После того, как объект получен, мы можем обратиться к его методам;
- В данном примере мы заводим машину (к тому же, изменив состояние объекта, то есть значение одного из атрибутов);
- Проверим значение затронутого атрибута;

СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА

```
my_car = Car("Красный", "Бензиновый", "Автомат")

print(f"Цвет моей машины {my_car.color}.")

my_car.start()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")

my_car.shut_down()

if my_car.working == True:
    print("Машина заведена")
else:
    print("Машина заглушена")
```

- После того, как объект получен, мы можем обратиться к его методам;
- В данном примере мы глушим машину (к тому же, изменив состояние объекта, то есть значение одного из атрибутов);
- Проверим значение затронутого атрибута;

СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА

Разумеется, можно создавать более одного объекта одного и того же класса:

```
car_1 = Car("Красный", "Бензиновый", "Автомат")

car_2 = Car("Синий", "Дизельный", "Автомат")

print(f"Цвет первой машины {car_1.color}.")
print(f"Двигатель первой машины {car_1.engine}.")

print(f"Цвет второй машины {car_2.color}.")
print(f"Двигатель второй машины {car_2.engine}.")
```

создание первой машины;

создание второй машины;

получение значений атрибутов первой машины

получение значений атрибутов второй машины

ПРИВАТНЫЕ АТРИБУТЫ КЛАССА

```
class Car():
    __working = False
    color = "White"

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission


car_1 = Car("Красный", "Бензиновый", "Автомат")

print(car_1.working)
print(car_1.__working)
```

Иногда возникает необходимость скрывать атрибуты класса так, чтобы к ним не было возможности обратиться напрямую.

Такие атрибуты являются *приватными*.

Ошибка;

Ошибка;

ПРИВАТНЫЕ АТРИБУТЫ КЛАССА

```
class Car():
    __working = False
    color = "White"

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission



    def is_working(self):
        return self.__working

car_1 = Car("Красный", "Бензиновый", "Автомат")

print(car_1.is_working())
```

Получить доступ к таким атрибутам можно при помощи соответствующих методов класса:

False;

ПРИВАТНЫЕ МЕТОДЫ КЛАССА

```
class Car():

    . . .

    def start(self):
        if self.__working == False:
            __change_working_state(True)
            print(f"{self.engine} двигатель запущен")

    def __change_working_state(self, value):
        self.__working = value


car_1 = Car("Красный", "Бензиновый", "Автомат")

car_1. __change_working_state(self, True)
car_1.start()
```

Методы класса также могут быть *приватными*:

Ошибка;

Успешный вызов;

ПРИВАТНЫЕ МЕТОДЫ КЛАССА

```
class Car():
    __working = False
    working = False
    color = "White"

    def __init__(self, color, engine, transmission):
        self.color = color
        self.engine = engine
        self.transmission = transmission
        self.__working = False

car_1 = Car("Красный", "Бензиновый", "Автомат")

car_1.__working = True
car_1.working = True
```

Важно: старайтесь не дублировать имена *приватных* и *публичных* атрибутов и методов.

Ошибка;

Успешный вызов;

Иерархия классов.

Наследование классов.

Абстрактные методы.

Абстрактные классы.

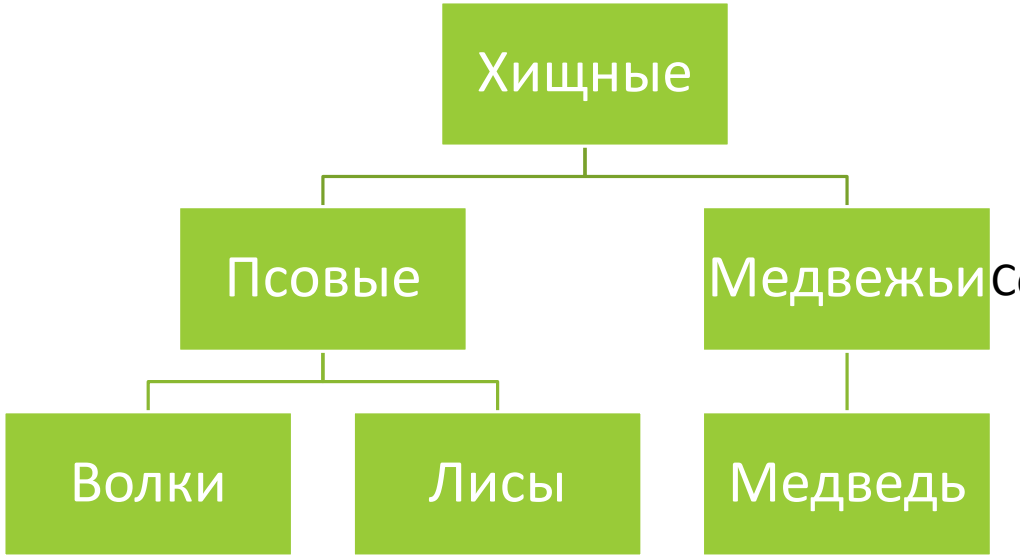
Множественное наследование.

НАСЛЕДОВАНИЕ

ИЕРАРХИЯ КЛАССОВ

Классы объектов, в свою очередь, могут быть организованы в иерархическую структуру.

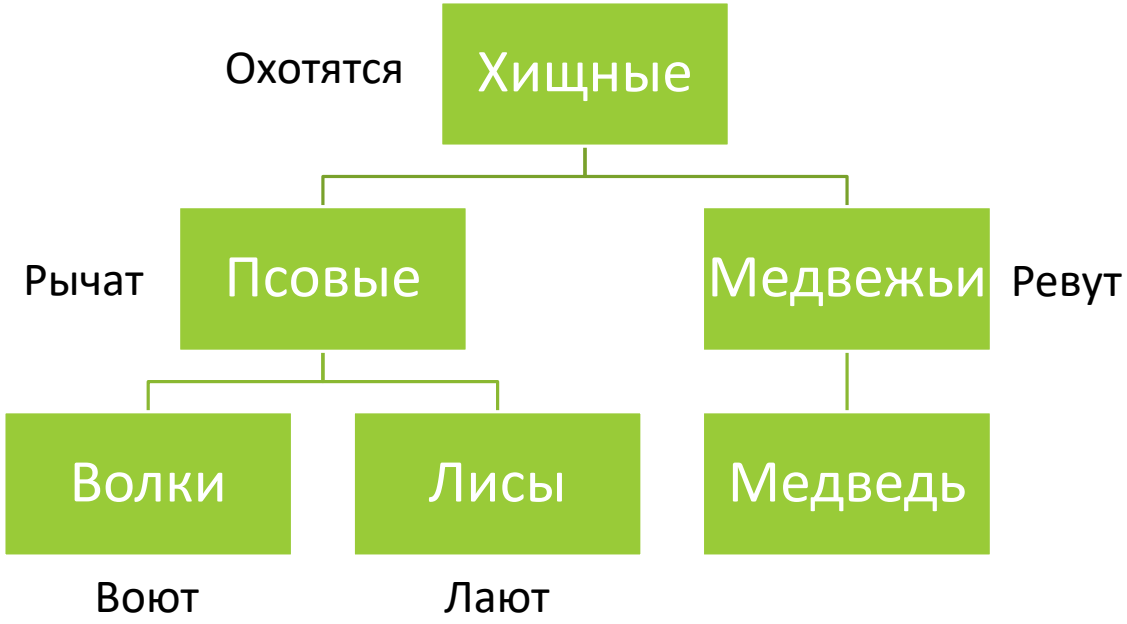
При этом, методы и атрибуты старшего класса (родителя) будут доступны для младших классов (потомков), но не наоборот.



ИЕРАРХИЯ КЛАССОВ

Классы объектов, в свою очередь, могут быть организованы в иерархическую структуру.

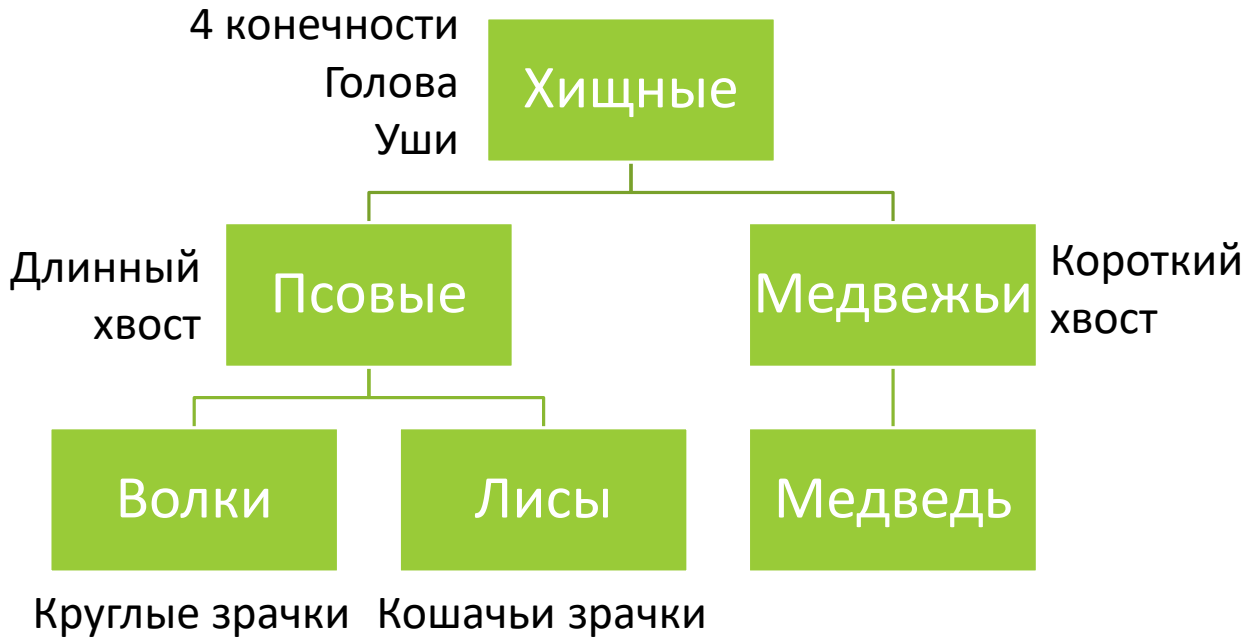
При этом, методы и атрибуты старшего класса (родителя) будут доступны для младших классов (потомков), но не наоборот.



ИЕРАРХИЯ КЛАССОВ

Классы объектов, в свою очередь, могут быть организованы в иерархическую структуру.

При этом, методы и атрибуты старшего класса (родителя) будут доступны для младших классов (потомков), но не наоборот.



НАСЛЕДОВАНИЕ КЛАССОВ

```
class Predator:
    limb_fr = "передняя правая"
    limb_fl = "передняя левая"
    limb_hr = "задняя правая"
    limb_hl = "задняя левая"
    head = "голова"
    ear_left = "левое ухо"
    ear_right = "правое ухо"

    def hunt(self):
        print("Зверь охотится")
```

```
class Canid(Predator):
    long_tail = "длинный хвост"

    def growl(self):
        print("Зверь рычит")

class Wolf(Canid):
    round_eyes = "голубые"

    def howl(self):
        print("Зверь воет")
```

```
grey_wolf = Wolf()
grey_wolf.hunt()
grey_wolf.growl()
grey_wolf.howl()
```

```
# Зверь охотится
# Зверь рычит
# Зверь воет
```

НАСЛЕДОВАНИЕ КЛАССОВ

```
class Predator:
    limb_fr = "передняя правая"
    limb_fl = "передняя левая"
    limb_hr = "задняя правая"
    limb_hl = "задняя левая"
    head = "голова"
    ear_left = "левое ухо"
    ear_right = "правое ухо"

    def hunt(self):
        print("Зверь охотится")
```

```
class Canid(Predator):
    long_tail = "длинный хвост"

    def growl(self):
        print("Зверь рычит")

class Wolf(Canid):
    round_eyes = "голубые"

    def howl(self):
        print("Зверь воет")
```

```
some_canid = Canid()
some_canid.hunt()
some_canid.growl()
some_canid.howl()
```

```
# Зверь охотится
# Зверь рычит
# Ошибка; howl отсутствует в классе Canid
```

НАСЛЕДОВАНИЕ КЛАССОВ

```
class Predator:
    limb_fr = "передняя правая"
    limb_fl = "передняя левая"
    limb_hr = "задняя правая"
    limb_hl = "задняя левая"
    head = "голова"
    ear_left = "левое ухо"
    ear_right = "правое ухо"

    def hunt(self):
        print("Зверь охотится")
```

```
class Canid(Predator):
    long_tail = "длинный хвост"

    def growl(self):
        print("Зверь рычит")

class Wolf(Canid):
    round_eyes = "голубые"

    def howl(self):
        print("Зверь воет")
```

```
some_canid = Canid()
print(some_canid.head)
print(some_canid.long_tail)
print(some_canid.round_eyes)
```

```
# голова
# длинный хвост
# Ошибка; round_eyes отсутствует в классе Canid
```

ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ

```
class Predator:
    limb_fr = "передняя правая"
    limb_fl = "передняя левая"
    limb_hr = "задняя правая"
    limb_hl = "задняя левая"
    head = "голова"
    ear_left = "левое ухо"
    ear_right = "правое ухо"

    def hunt(self):
        print("Зверь охотится (не определено)")
```

```
class Canid(Predator):
    long_tail = "длинный хвост"

    def growl(self):
        print("Зверь рычит")

class Wolf(Canid):
    round_eyes = "голубые"

    def howl(self):
        print("Зверь воет")

    def hunt(self):
        print("Зверь охотится (определено)")
        print("в стае")
```

```
grey_wolf = Wolf()

grey_wolf.hunt()
```

```
# Зверь охотится (определено)
# в стае
```

АБСТРАКТНЫЕ КЛАССЫ

Абстрактным классом называется класс, предназначенный исключительно для наследования, то есть содержащий только *абстрактные (нефункциональные)* методы:

```
class Class1(object):
    def test(self, x):
        raise NotImplementedError("Необходимо переопределить метод")

class Class2(Class1):
    def test(self, x):
        print(x)
```

Абстрактный метод
Возбуждаем исключение (ошибку)

Наследуем абстрактный метод
Переопределяем метод

АБСТРАКТНЫЕ КЛАССЫ

Несмотря на то, что встроенной явной поддержки абстрактных методов в Python нет, с этой целью можно воспользоваться модулем *abc*:

```
from abc import ABCMeta, abstractmethod

class Class1(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def test(self, x):
        pass

class Class2(Class1):
    def test(self, x):
        print(x)
```

Абстрактный класс

Абстрактный метод

Наследуем абстрактный метод

Переопределяем метод

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Множественное наследование позволяет наследовать класс от нескольких родительских (необязательно связанных общим предком) классов:

```
class Canid(Predator):
    long_tail = "длинный хвост"

    def growl(self):
        print("Зверь рычит")

    def hunt(self):
        print("Зверь охотится стаей")
```

```
class Feline(Predator):
    flex_tail = "гибкий хвост"

    def meow(self):
        print("Зверь мякает")

    def hunt(self):
        print("Зверь охотится из засады")
```

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

```
class Hybrid(Canid, Feline):
    is_hybrid = "Это гибрид"

hybrid = Hybrid()

print(hybrid.is_hybrid)

print(hybrid.long_tail)

print(hybrid.flex_tail)

hybrid.growl()

hybrid.meow()

hybrid.hunt()
```

Важно: обращайте внимание на состав наследуемых методов и параметров.

```
# Это гибрид
# длинный хвост
# гибкий хвост
# Зверь рычит
# Зверь мяукает
# Зверь охотится стаей
```

Понятие исключения.

Обработка исключений.

Определение исключений.

ИСКЛЮЧЕНИЯ

ИСКЛЮЧЕНИЯ

Исключения (exceptions) используются для того, чтобы сообщать программисту о runtime-ошибках, то есть об ошибках, возникающих во время исполнения программы.

В общем случае, такие ошибки не связаны с алгоритмическими ошибками (такие выявляются программистом или пользователем).

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Если в процессе работы программы *исключение* было получено, но не перехвачено (обработано), программа останавливает свое выполнение.

Чтобы перехватить - и обработать - исключение используется конструкция *try-except*:

```
delim = int(input("Введите делитель:"))

k = 1 / delim

print(k)
```

Без перехвата, при вводе «0», мы получаем ошибку:

ZeroDivisionError: division by zero

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Если в процессе работы программы *исключение* было получено, но не перехвачено (обработано), программа останавливает свое выполнение.

Чтобы перехватить - и обработать - исключение используется конструкция *try-except*:

```
delim = int(input("Введите делитель:"))

try:
    k = 1 / delim
except ZeroDivisionError:
    k = 0

print(k)
```

```
# После «пробы» кода на возникновение ошибки
# ZeroDivisionError
# Мы можем обработать её;


# Результат при вводе «0»: «0»
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Важно: полная форма конструкции выглядит как *try-except-else-finally*:

```
try:
    delim = int(input("Введите делитель:"))
    k = 1 / delim

except ZeroDivisionError:
    k = 0

else:
    print("Все корректно")

finally:
    print("Конец конструкции try")

print(k)
```

Перехват исключения;

Обработка конкретного исключения;

Выполняется, если исключений поймано не было;

Блок кода, который выполнится в любом случае
Даже если исключений перехвачено не было

ИЕРАРХИЯ ИСКЛЮЧЕНИЙ

. . .

- Exception
  - StopIteration
  - ArithmeticError
    - FloatingPointError
    - OverflowError
    - ZeroDivisionError
  - AssertionError
  - AttributeError

. . .

Встроенные исключения являются *классами* и организованы в иерархическую структуру.

Таким образом, если мы хотим отловить не только исключение *«ZeroDivisionError»*, но и все ему подобные, нам достаточно поставить в блок *except* исключение *«ArithmeticError»*.

ОПРЕДЕЛЕНИЕ СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ

Также возможно определять собственные исключения на основе встроенных:

```
class ValueTooSmallError(Exception):
    """Отправляется, если значение слишком маленькое"""
    pass

delim = int(input("Введите делитель (неотрицательный):"))
```

```
try:
    if delim < 0:
        raise ValueTooSmallError

    k = 1 / delim

except ZeroDivisionError:
    k = 0
except ValueTooSmallError:
    k = 0
    print("Введено отрицательное значение")
else:
    print("Все корректно")
finally:
    print("Конец конструкции try")

print(k)
```

БЛАГОДАРЮ ЗА ВНИМАНИЕ!

Гаврилов Денис Андреевич, преподаватель кафедры СИ ФИТ НГУ