Объектно-ориентированное программирование на *python*

Объектно-ориентированное программирование

Python проектировался как объектно-ориентированный язык программирования.

Построен с учетом следующих принципов (по Алану Кэю, автору объектно-ориентированного языка Smalltalk):

1. Все данные в нем представляются объектами.
2. Программу можно составить как набор взаимодействующих объектов, посылающих друг другу сообщения.
3. Каждый объект имеет собственную часть памяти и может состоять из других объектов.
4. Каждый объект имеет тип.
5. Все объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

Основные понятия

Объектно-ориентированная программа – это совокупность взаимодействующих **объектов**.

Над объектами можно производить **операции** (посылая им сообщения).

Сообщения - это запросы к объекту выполнить некоторые действия, т.е. взаимодействие объектов заключается в вызове **методов** одних объектов другими.

Характеристики объекта – это ***атрибуты***, способы поведения – это ***методы***.

Каждый объект принадлежит определенному классу (типу), который задает поведение объектов, созданных на его основе.

Класс — это описание объектов определенного типа.

Объект, созданный на основе некоторого класса, называется **экземпляром класса**.

Создание классов, методов и объектов

Объявление класса

```
class ИМЯ КЛАССА ():
    ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
    . . .
```

Объявление метода

```
    def ИМЯ МЕТОДА (self, . . .):
        self. ПЕРЕМЕННАЯ = ЗНАЧЕНИЕ
        . . .
    . . .
# Основная часть
```

Создание объекта

```
ПЕРЕМЕННАЯ = ИМЯ КЛАССА ()
. . .
```

Вызов метода

```
ОБЪЕКТ.ИМЯ МЕТОДА ()
. . .
```

Атрибуты класса — это имена переменных вне функций и имена функций. Наследуются всеми объектами, созданными на основе данного класса.

Создание классов, методов и объектов

Пример_1.

```
class Person():          # Создание пустого класса
   pass
Person.money = 150  # Создание атрибута объекта класса

obj1 = Person()    # Создание экземпляра класса
obj2 = Person()    # Создание экземпляра класса
obj1.name = 'Bob‘      # Создание атрибута экземпляра класса
obj2.name = 'Masha‘ # Создание атрибута экземпляра класса

print (obj1.name,'has',obj1.money,'dollars.')     # Вывод
print (obj2.name,'has',obj2.money,'dollars.')     # Вывод
```

Вывод:

```
>>>
Bob has 150 dollars.
Masha has 150 dollars.
```

Создание классов, методов и объектов

Пример_2.

```
# Создание класса
class Person():
    name = ""
    money = 0
# Создание объектов
obj1 = Person()
obj2 = Person()

obj1.name = 'Bob'
obj1.money = 150

obj2.name = 'Masha'

print (obj1.name,'has',obj1.money,'dollars.')
print (obj2.name,'has',obj2.money,'dollars.')
```

Вывод:

```
>>>
Bob has 150 dollars.
Masha has 0 dollars.
```

Создание классов, методов и объектов

Пример_3. # Создание класса, объекта и вызов метода

```
class Person():
    name = ""
    money = 0
    def out (self):   # self - ссылка на экземпляр класса
        print(self.name,'has',self.money,'dollars.')
    def changemoney (self,newmoney):
        self.money = newmoney

obj1 = Person()
obj2 = Person()
obj1.name = 'Bob'
obj2.name = 'Masha'
obj1.out()
obj2.out()
obj1.changemoney(150)
obj1.out()
```

Вывод:

```
>>>
Bob has 0 dollars.
Masha has 0 dollars.
Bob has 150 dollars.
```

Создание классов, методов и объектов

Пример_4.

```
class Critter():                          # создание класса
   """Виртуальный питомец"""      # строка документирования
   # создание метода
   def talk(self):
      print("Привет.  Я животное – экземпляр класса Critter.")

# основная часть
# создание объекта и вызов метода
crit = Critter()
crit.talk()

input("\nНажмите Enter, чтобы выйти.")
```

Вывод:

```
Привет.  Я животное – экземпляр класса Critter.

Нажмите Enter, чтобы выйти.
```

Применение конструкторов

Конструктор класса __init__ автоматически создает атрибуты объекта при вызове класса.

```
# Демонстрирует метод-конструктор

class Critter():
    """Виртуальный питомец"""
    def __init__(self):   # метод-конструктор
        print("Появилось на свет новое животное!")

    def talk(self):
        print("\n Привет.  Я животное – экземпляр класса Critter.")

crit1 = Critter()
crit2 = Critter()

crit1.talk()
crit2.talk()
```

Вывод:

```
Появилось на свет новое животное!
Появилось на свет новое животное!

Привет.  Я животное – экземпляр класса Critter.

Привет.  Я животное – экземпляр класса Critter.
```

Применение деструкторов

В объектно-ориентированном программировании деструктор вызывается при удалении или уничтожении объекта.

Деструктор используется для выполнения действий по очистке перед разрушением объекта, таких как закрытие соединений с базой данных или дескриптор файла.

Применение деструкторов

В Python деструктор вызывается не вручную, а полностью автоматически. Это происходит в следующих двух случаях:

- когда объект выходит за пределы области видимости
- когда счетчик ссылок на объект достигает 0.

Для определения деструктора используется специальный метод _ _del_ _().

Например, когда мы выполняем

del имя_объекта, деструктор вызывается автоматически, и объект собирается в мусор.

Создание деструктора с помощью метода __del__()

Магический метод __del__() используется как деструктор в Python. Метод __del__() будет неявно вызываться, когда все ссылки на объект будут удалены, то есть когда объект подходит для сборщика мусора.

Этот метод автоматически вызывается в Python, когда экземпляр собираются уничтожить. Его также называют финализатором или (неправильно) деструктором.

Синтаксис объявления деструктора будет следующим:

```
def __del__(self):
    # тело деструктора
```

Создание деструктора с помощью метода __del__()

```
class Student:
    # конструктор
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')
    def show(self):
        print('Hello, my name is', self.name)
    # деструктор
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')
# создать объект
s1 = Student('Emma')
s1.show()
# удалить объект
del s1
```

Применение атрибутов

Усложняем программу:

```
# Демонстрирует создание атрибутов объекта

class Critter():
    """Виртуальный питомец"""
    def __init__(self, name):
        print("Появилось на свет новое животное!")
        self.name = name
    def __str__(self):                  # возвращает строку, которая
        rep = "Объект класса Critter\n"     # содержит значение
        rep += "имя: " + self.name + "\n"   # атрибута name
        return rep
    def talk(self):
        print("Привет.  Меня зовут", self.name, "\n")

# Продолжение следует
```

Применение атрибутов

```
# Демонстрирует создание атрибутов объекта (продолжение)
# Основная часть

crit1 = Critter("Бобик")
crit1.talk()

crit2 = Critter("Мурзик")
crit2.talk()

print("Вывод объекта crit1 на э
print(crit1)

print("Доступ к атрибуту crit1.n
print(crit1.name)

input ("\nНажмите Enter, чтобы
```

Вывод:

```
Появилось на свет новое животное!
Привет.  Меня зовут Бобик.

Появилось на свет новое животное!
Привет.  Меня зовут Мурзик.

Вывод объекта crit1 на экран:
Объект класса Critter
имя: Бобик

Доступ к атрибуту crit1.name:
Бобик

Нажмите Enter, чтобы выйти.
```

Применение атрибутов класса и статических методов

Значение, связанное с целым классом, - **атрибут** класса.

Методы, связанные с целым классом, - **статические**.

```
# Демонстрирует атрибуты класса и статические методы

class Critter():
    """Виртуальный питомец"""
    total = 0           # атрибут класса
    @staticmethod       # декоратор меняет смысл метода
    def status():       # статический метод, отсутствует self
        print("\nВсего животных сейчас", Critter.total)
    def __init__(self, name):
        print("Появилось на свет новое животное!")
        self.name = name
        Critter.total += 1

# Продолжение следует
```

Применение атрибутов класса и статических методов

```
# Демонстрирует атрибуты класса и статические методы
(продолжение)
# Основная часть

print("Значение атрибута класса Critter.total:", end=" ")
print(Critter.total)

print("\nСоздаю животн
crit1 = Critter("животное
crit2 = Critter("животное
crit3 = Critter("животное

Critter.status()          # вы

print("\nНахожу значен
")
print(crit1.total))

input ("\nНажмите Enter,
```

Вывод:

```
Значение атрибута класса Critter.total: 0

Создаю животных.
Появилось на свет новое животное!
Появилось на свет новое животное!
Появилось на свет новое животное!

Всего животных сейчас 3

Нахожу значение атрибута класса через
объект: 3

Нажмите Enter, чтобы выйти.
```

Инкапсуляция объектов. Применение закрытых атрибутов и методов

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным).

Атрибуты и методы класса делятся на ***открытые*** из вне (public) и ***закрытые*** (private).

Открытые атрибуты также называют *интерфейсом объекта*, т. к. с их помощью с объектом можно взаимодействовать.

Закрытые атрибуты *нельзя изменить*, находясь вне класса.

Инкапсуляция призвана обеспечить надежность программы.

Инкапсуляция объектов. Применение закрытых атрибутов и методов

Одиночное подчеркивание в начале имени атрибута указывает, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.

```
class A:
    def _private(self):
        print("Это закрытый метод!")

>>> a = A()
>>> a._private()
Это закрытый метод!
```

Инкапсуляция объектов. Применение закрытых атрибутов и методов

Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени.

```
class B:
    def __private(self):
        print("Это закрытый метод!")

>>> b = B()
>>> b.__private()
Traceback (most recent call last):
    File "", line 1, in
        b.__private()
AttributeError: 'B' object has no attribute '__private'
```

Атрибут будет доступным под именем _ИмяКласса__ИмяАтрибута:

```
>>> b._B__private()
Это закрытый метод!
```

Инкапсуляция объектов. Применение закрытых атрибутов и методов

```
# Демонстрирует закрытые переменные и методы

class Critter():
    """Виртуальный питомец"""
    def __init__(self, name, mood):
        print("Появилось на свет новое животное!")
        self.name = name          # открытый атрибут
        self.__mood = mood        # закрытый атрибут

    def talk(self):
        print("\nМеня зовут", self.name)
        print("Сейчас я чувствую себя", self.__mood, "\n")

    def __private_method(self):
        print("Это закрытый метод!")

    def public_method(self):
        print("Это открытый метод!")
        self.__private_method()
# Продолжение следует
```

Инкапсуляция объектов. Применение закрытых атрибутов и методов

```
# Демонстрирует закрытые переменные и методы (продолжение)
# основная часть
crit = Critter(name = "Бобик", mood = "прекрасно")
crit.talk()
crit.public_method()

input ("\nНажмите Enter, чтобы выйти.")
```

Вывод:

```
Появилось на свет новое животное!

Меня зовут Бобик
Сейчас я чувствую себя прекрасно

Это открытый метод!
Это закрытый метод!

Нажмите Enter, чтобы выйти.
```

Управление доступом к атрибутам

Свойство **–** объект с методами, которые позволяют косвенно обращаться к закрытым атрибутам.

```
# Демонстрирует свойства
class Critter():
    """Виртуальный питомец"""
    def __init__(self, name):
        print("Появилось на свет новое животное!")
        self.__name = name        # закрытый атрибут

    @property                     # декоратор
    def name(self):               # свойство (позволяет узнать
        return self.__name        # значение закрытого атрибута
                                  # __name этого объекта внутри #
или вне объявления класса)
# Продолжение следует
```

Управление доступом к атрибутам

```
# Демонстрирует свойства
(продолжение)
    @name.setter          # метод устанавливает новое
    def name(self, new_name):   # значение свойства name
        if new_name == "":
            print("Имя животного не может быть пустой строкой.")
        else:
            self.__name = new_name
            print("Имя успешно изменено.")

    def talk(self):
        print("\nПривет, меня зовут", self.name)

# Продолжение следует
```

Управление доступом к атрибутам

```
# Демонстрирует свойства
# основная часть
crit = Critter("Бобик")
crit.talk()

print("\nМ
print(crit.na

print("\nП
crit.name =
print("Мое
print(crit.na

print("\nП
crit.name =
print("Мое
print(crit.na

input ("\nН
```

Вывод:

```
Появилось на свет новое животное!

Привет, меня зовут Бобик

Мое животное зовут: Бобик

Попробую изменить имя животного на Шарик...
Имя успешно изменено.
Мое животное зовут: Шарик

Попробую изменить имя животного на пустую строку...
Имя животного не может быть пустой строкой.
Мое животное зовут: Шарик

Нажмите Enter, чтобы выйти.
```

Пример программы «Мое животное»

```
# Мое животное
# Виртуальный питомец, от котором пользователь может заботиться

class Critter():                    # класс Critter
    """Виртуальный питомец"""

# метод-конструктор класса инициализирует три открытых
# атрибута
    def __init__(self, name, hunger = 0, boredom = 0):
        self.name = name
        self.hunger = hunger
        self.boredom = boredom

# закрытый метод, увеличивающий уровень голода и уныния
    def __pass_time(self):
        self.hunger += 1
        self.boredom += 1

# Продолжение следует
```

Пример программы «Мое животное»

```
# Мое животное (продолжение)

# свойство, отражающее самочувствие животного
    @property
    def mood(self):
        unhappiness = self.hunger + self.boredom
        if unhappiness < 5:
            m = "прекрасно"
        elif 5 <= unhappiness <= 10:
            m = "неплохо"
        elif 11 <= unhappiness <= 15:
            m = "так себе"
        else:
            m = "ужасно"
        return m

# Продолжение следует
```

Пример программы «Мое животное»

```
# Мое животное (продолжение)

# метод сообщает о самочувствии животного
    def talk(self):
        print("Меня зовут", self.name, end=" ")
        print("и сейчас я чувствую себя", self.mood, "\n")
        self.__pass_time()

# метод уменьшает уровень голода животного
    def eat(self, food = 4):
        print("Мррр. Спасибо.")
        self.hunger -= food
        if self.hunger < 0:
            self.hunger = 0
        self.__pass_time()

# Продолжение следует
```

Пример программы «Мое животное»

```
# Мое животное (продолжение)

# метод снижает уровень уныния животного
    def play(self, fun = 4):
        print("Уиии!")
        self.boredom -= fun
        if self.boredom < 0:
            self.boredom = 0
        self.__pass_time()
# основная часть программы
def main():
    crit_name = input("Как вы назовете свое животное?: ")
    crit = Critter(crit_name)

# Продолжение следует
```

Пример программы «Мое животное»

```
# Мое животное (продолжение)
# основная часть программы (продолжение)
# создание меню
    choice = None
    while choice != "0":
        print \
        ("""
        Мое животное

        0 – Выйти
        1 – Узнать о самочувствии животного
        2 – Покормить животное
        3 – Поиграть с животным
        """)

        choice = input("Ваш выбор: ")
        print()

# Продолжение следует
```

Пример программы «Мое животное»

```
# Мое животное (продолжение)

# создание меню (продолжение)
    # выход
    if choice == "0":
        print("До свидания.")

    # беседа с животным
    elif choice == "1":
        crit.talk()

    # кормление животного
    elif choice == "2":
        crit.eat()

    # игра с животным
    elif choice == "3":
        crit.play()

# Продолжение следует
```

Пример программы «Мое животное»

```
# Мое животное (продолжение)

# создание меню (продолжение)

        # непонятный ввод
        else:
            print("\nИзвините, в меню нет пункта", choice)

# запуск программы
main()
("\nНажмите Enter, чтобы выйти.")
```

Наследование

Наследование – одна из ключевых идей ООП. Можно создать класс и унаследовать все атрибуты и методы *родительского класса*. Родительский класс должен быть более обобщённой, абстрактной версией дочернего класса.

```
# Расширение класса через наследование
class Person:
    def __init__(self,n):
        self.name=n
    def write(self):
        print(self.name)

class Student(Person):
    def __init__(self,gr,n,):
        Person.__init__(self,n)
        self.group=gr
    def write(self):
        print(self.name,self.group)
```

Ввод и вывод:

```
>>> p=Person("Petya")
>>> p.write()
Petya
>>> s=Student(23,"Vasya")
>>> s.write()
Vasya 23
>>>
```

Полиморфизм

Полиморфизм – разное поведение одного и того же метода в разных классах, при этом действия, совершаемые с объектами, могут существенно различаться.

```
class T1:
    n=10
    def total(self,N):
        self.total = int(self.n) + int(N)
class T2:
    def total(self,s):
        self.total = len(str(s))

t1 = T1()
t2 = T2()
t1.total(45)
t2.total(45)
print (t1.total)     # Вывод: 55
print (t2.total)     # Вывод: 2
```

Пример программы «Банк»

Реализовать работу банка (сотрудник, клиент, вклад, кредит). В программе должны быть классы и объекты, принадлежащие разным классам; один объект с помощью метода своего класса должен так или иначе изменять данные другого объекта.

```
# Банк
# Расширение класса через наследование
# Использование методов

class Person():                 # родительский класс
  def __init__(self,fio):       # метод-конструктор
    self.fio = fio
    #print("Person", self.fio, "created")

  def __str__(self):            # возвращает строку, которая
    return self.fio             # содержит значение атрибута fio

# Продолжение следует
```

Пример программы «Банк»

```
# Банк (продолжение)

class Sotrudnik(Person):          # дочерний класс
  def __init__(self,fio,job_title,):     # унаследовал атрибут
    Person.__init__(self,fio)            # родительского класса fio
    self.job_title = job_title
    print(self.job_title,"-",self.fio,"\n")

  def kredit(self,client):
    client.dolg()
    print(self.fio,"оформил кредит \n")

  def vklad(self,client):
    client.dohod()
    print(self.fio,"оформил вклад \n")

# Продолжение следует
```

Пример программы «Банк»

```
# Банк (продолжение)
class Client(Person):
  def __init__(self,fio,sum_vklada,sum_kr):
    Person.__init__(self,fio)
    self.sum_vklada = sum_vklada
    self.sum_kr = sum_kr
    print("Клиент:", self.fio,"Сумма кредита:", self.sum_kr,
        «Сумма вклада:", self.sum_vklada,"\n")
# Продолжение следует
```

Пример программы «Банк»

```
# Банк (продолжение)

  def dohod(self,persent=0.03):
    self.sum_vklada = float(self.sum_vklada*(1+persent))

  def dolg(self,persent=0.18,amount=0):
    self.sum_kr = float(self.sum_kr*(1+persent))
    self.sum_kr -= amount

  def out(self):
    print("Клиент:", self.fio,"Сумма долга:", self.sum_kr)
    print("Клиент:", self.fio,"Доход:", self.sum_vklada)

# Продолжение следует
```

Пример программы «Банк»

```
# Банк (продолжение)

name_s = input("Введите ФИО сотрудника:")
job_s  = input("Введите должность:")

name_c = input("Введите ФИО клиента:")
svk = int(input("Введите сумму вклада:"))
skr = int(input("Введите сумму кредита:"))

obj1 = Sotrudnik(name_s,job_s)
obj2 = Client(name_c,svk,skr)
obj1.vklad(obj2)
obj1.kredit(obj2)
obj2.out()

input("\n\nНажмите Enter чтобы выйти.")
```

Пример программы «Банк»

Вывод:

Введите ФИО сотрудника:Иванов
Введите должность:операционист
Введите ФИО клиента:Петров
Введите сумму вклада:100
Введите сумму кредита:1000
операционист - Иванов

Клиент: Петров Сумма кредита: 1000 Сумма вклада: 100

Иванов оформил вклад

Иванов оформил кредит

Клиент: Петров Сумма долга: 1180.0
Клиент: Петров Доход: 103.0