



ООП

Python 6.0

ООП

Несмотря на то, что Python - мультипарадигменный язык (сочетает в себе элементы структурного, объектно-ориентированного и функционального программирования), основной парадигмой все-таки является ООП

Первое, что нужно помнить о Python

... в Python все является объектом!

Строки - это объекты.

Списки являются объектами.

Функции являются объектами.

Классы являются объектами.

Экземпляры класса являются объектами.

Свойства являются объектами.

Модули являются объектами.

Файлы являются объектами.

Сетевые подключения являются объектами и тд

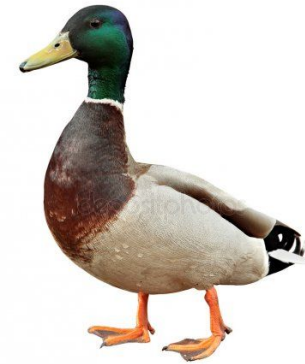
Динамическая типизация

Python - это динамический язык с динамической системой типов. Однако, несмотря на наличие динамической системы типов, Python строго типизирован

Динамический язык - это такой язык программирования и такой транслятор, которые позволяют определять типы данных и осуществлять синтаксический анализ и трансляцию "на лету", непосредственно на этапе выполнения.

Утиная типизация

- ▶ Порой знатоки Питона напускают на себя таинственный вид и говорят об "Утиной типизации".
- ▶ Утиная типизация (Duck typing) — это применение "утиного теста" в программировании:



Если объект крикает как утка, летает как утка и ходит как утка, то скорее всего это утка.

Утиная типизация

- ▶ Это один из видов динамической типизации при которой принадлежность объекта к тому или иному классу (интерфейсу) определяется путем проверки на наличие всех свойств искомого класса в созданном объекте.
- ▶ Иначе говоря если объект реализует все методы какого-то интерфейса, то говорят, что он реализует этот интерфейс.
- ▶ Принцип утиной типизации гласит, что вам не важно, какой у вас тип объекта - важно можете ли вы выполнить необходимые действия с вашим объектом или нет.

Классы

- ▶ Так как Python поддерживает объектно-ориентированную парадигму программирования, это значит, что мы можем определить компоненты программы в виде классов.
- ▶ Класс является шаблоном или формальным описанием объекта, а объект представляет экземпляр этого класса, его реальное воплощение.
- ▶ Можно провести следующую аналогию: у всех у нас есть некоторое представление о человеке - наличие двух рук, двух ног, головы, пищеварительной, нервной системы, головного мозга и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Классы

- ▶ С точки зрения кода класс объединяет набор функций и переменных, которые выполняют определенную задачу. Функции класса еще называют **методами**. Они определяют поведение класса. А переменные класса называют **атрибутами**- они хранят состояние класса
- ▶ Класс определяется с помощью ключевого слова **class**:

class название_класса:
 методы_класса

- ▶ Для создания объекта класса используется следующий синтаксис:
название_объекта = название_класса([параметры])

Пример

- Определим простейший класс `Person`, который будет представлять человека:

```
class Person:
```

```
    name = "Tom"
```

```
def display_info(self):
```

```
    print("Привет, меня зовут", self.name)
```

```
person1 = Person()
```

```
person1.display_info()      # Привет, меня зовут Tom
```

```
person2 = Person()
```

```
person2.name = "Sam"
```

```
person2.display_info()      # Привет, меня зовут Sam
```

Пояснение

- ▶ Класс **Person** определяет атрибут **name**, который хранит имя человека, и метод **display_info**, с помощью которого выводится информация о человеке.
- ▶ При определении методов любого класса следует учитывать, что все они должны принимать в качестве первого параметра ссылку на текущий объект, который согласно условностям называется **self** (в ряде языков программирования есть своего рода аналог - ключевое слово **this**). Через эту ссылку внутри класса мы можем обратиться к методам или атрибутам этого же класса. В частности, через выражение **self.name** можно получить имя пользователя.
- ▶ После определения класс **Person** создаем пару его объектов - **person1** и **person2**. Используя имя объекта, мы можем обратиться к его методам и атрибутам. В данном случае у каждого из объектов вызываем метод **display_info()**, который выводит строку на консоль, и у второго объекта также изменяем атрибут **name**. При этом при вызове метода **display_info** не надо передавать значение для параметра **self**.

Конструктор

- ▶ В объектно-ориентированном программировании конструктором класса называют метод, который автоматически вызывается при создании объектов.
- ▶ Его также можно назвать конструктором объектов класса. Имя такого метода обычно регламентируется синтаксисом конкретного языка программирования.
- ▶ Необходимость конструкторов связана с тем, что нередко объекты должны иметь собственные свойства сразу.
- ▶ Однако бывает, что надо допустить создание объекта, даже если никакие данные в конструктор не передаются. В таком случае параметрам конструктора класса задаются значения по умолчанию:

Значения по умолчанию

```
class Rectangle:
```

```
    def __init__(self, w = 0.5, h = 1):
```

```
        self.width = w
```

```
        self.height = h
```

```
    def square(self):
```

```
        return self.width * self.height
```

```
rec1 = Rectangle(5, 2)
```

```
rec2 = Rectangle()
```

```
rec3 = Rectangle(3)
```

```
rec4 = Rectangle(h = 4)
```

```
print(rec1.square())
```

```
print(rec2.square())
```

```
print(rec3.square())
```

```
print(rec4.square())
```

Вывод:

```
10
0.5
3
2.0
```

Конструкторы

- ▶ Так, выше когда мы создавали объекты класса `Person`, мы использовали конструктор по умолчанию, который неявно имеет все классы:

```
person1 = Person()
```

```
person2 = Person()
```

- ▶ Однако мы можем явным образом определить в классах конструктор с помощью специального метода, который называется `__init()`. К примеру, изменим класс `Person`, добавив в него конструктор:

`class Person:`

`# конструктор`

`def __init__(self, name):`

`self.name = name # устанавливаем имя`

`def display_info(self):`

`print("Привет, меня зовут", self.name)`

`person1 = Person("Tom")`

`person1.display_info()` `# Привет, меня зовут Tom`

`person2 = Person("Sam")`

`person2.display_info()` `# Привет, меня зовут Sam`

Пояснение

- ▶ В качестве первого параметра конструктор также принимает ссылку на текущий объект - `self`. Нередко в конструкторах устанавливаются атрибуты класса. Так, в данном случае в качестве второго параметра в конструктор передается имя пользователя, которое устанавливается для атрибута `self.name`. Причем для атрибута необязательно определять в классе переменную `name`, как это было в предыдущей версии класса `Person`. Установка значения `self.name = name` уже неявно создает атрибут `name`.

```
person1 = Person("Tom")
```

```
person2 = Person("Sam")
```

```
Привет, меня зовут Tom
```

```
Привет, меня зовут Sam
```

- ▶ В итоге мы получим следующий консольный вывод:



Деструктор

- ▶ После окончания работы с объектом мы можем использовать оператор **del** для удаления его из памяти:

```
person1 = Person("Tom")
```

```
del person1    # удаление из памяти
```

```
# person1.display_info() # Этот метод работать не будет, так как person1 уже  
удален из памяти
```

- ▶ Стоит отметить, что в принципе это необязательно делать, так как после окончания работы скрипта все объекты автоматически удаляются из памяти.

Деструктор

- ▶ Кроме того, мы можем определить в классе деструктор, реализовав встроенную функцию `__del__`, который будет вызываться либо в результате вызова оператора `del`, либо при автоматическом удалении объекта.

Например:

```
class Person:
```

```
    # конструктор
```

```
    def __init__(self, name):
```

```
        self.name = name # устанавливаем имя
```

```
    def __del__(self):
```

```
        print(self.name, "удален из памяти")
```

```
    def display_info(self):
```

```
        print("Привет, меня зовут", self.name)
```

```
person1 = Person("Tom")
```

```
person1.display_info() # Привет, меня зовут Tom
```

```
del person1 # удаление из памяти
```

```
person2 = Person("Sam")
```

```
person2.display_info() # Привет, меня зовут Sam
```

```
del person2
```

```
Привет, меня зовут Tom
```

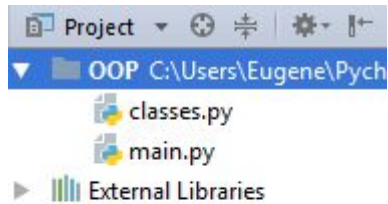
```
Tom удален из памяти
```

```
Привет, меня зовут Sam
```

```
Sam удален из памяти
```

Определение классов в модулях и подключение

- ▶ Как правило, классы размещаются в отдельных модулях и затем уже импортируются в основной скрипт программы. Пусть у нас будет в проекте два файла: файл `main.py` (основной скрипт программы) и `classes.py` (скрипт с определением классов).



Определение класса

- ▶ В файле classes.py определим два класса:

```
class Person:
```

```
    # конструктор
```

```
    def __init__(self, name):
```

```
        self.name = name # устанавливаем имя
```

```
    def display_info(self):
```

```
        print("Привет, меня зовут", self.name)
```

```
class Auto:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def move(self, speed):
```

```
        print(self.name, "едет со скоростью", speed, "км/ч")
```

- ▶ В дополнение к классу Person здесь также определен класс Auto, который представляет машину и который имеет метод move и атрибут name. Подключим эти классы и используем их в скрипте main.py:

```
from classes import Person, Auto
```

```
tom = Person("Tom")
```

```
tom.display_info()
```

```
bmw = Auto("BMW")
```

```
bmw.move(65)
```

Подключение классов

- ▶ Подключение классов происходит точно также, как и функций из модуля. Мы можем подключить весь модуль выражением:
- ▶ Либо подключить отдельные классы, как в примере выше.
- ▶ В итоге мы получим следующий консольный вывод:

```
Привет, меня зовут Tom  
BMW едет со скоростью 65 км/ч
```

Инкапсуляция

- ▶ Инкапсуляция — в ООП размещение в одном компоненте данных и методов, которые с ними работают. Также может означать скрытие внутренней реализации от других компонентов.
- ▶ По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его. Например:

Инкапсуляция

```
class Person:
```

```
    def __init__(self, name):
```

```
        self.name = name    # устанавливаем имя
```

```
        self.age = 1        # устанавливаем возраст
```

```
    def display_info(self):
```

```
        print("Имя:", self.name, "\tВозраст:", self.age)
```

```
tom = Person("Tom")
```

```
tom.name = "Человек-паук"    # изменяем атрибут name
```

```
tom.age = -129                # изменяем атрибут age
```

```
tom.display_info()           # Имя: Человек-паук    Возраст: -129
```


Инкапсуляция

- ▶ Но в данном случае мы можем, к примеру, присвоить возрасту или имени человека некорректное значение, например, указать отрицательный возраст. Подобное поведение нежелательно, поэтому возникает вопрос о контроле за доступом к атрибутам объекта.
- ▶ С данной проблемой тесно связано понятие инкапсуляции. Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объекта из вызывающего кода.
- ▶ Касательно инкапсуляции непосредственно в языке программирования Python скрыть атрибуты класса можно сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются свойствами.

Изменим выше определенный класс, определив в нем свойства:

class Person:

def __init__(self, name):

self.__name = name # устанавливаем имя

self.__age = 1 # устанавливаем возраст

def set_age(self, age):

if age in range(1, 100):

self.__age = age

else:

print("Недопустимый возраст")

def get_age(self):

return self.__age

def get_name(self):

return self.__name

def display_info(self):

print("Имя:", self.__name, "\tВозраст:", self.__age)

tom = Person("Tom")

tom.__age = 43

tom.display_info()

tom.set_age(-3486)

tom.set_age(25)

tom.display_info()

Атрибут age не изменится

Имя: Том Возраст: 1

Недопустимый возраст

Имя: Том Возраст: 25



Инкапсуляция

- ▶ Для создания приватного атрибута в начале его наименования ставится двойной прочерк: `self.__name`. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса. Например, присвоение значения этому атрибуту ничего не даст:

```
tom.__age = 43
```

- ▶ А попытка получить его значение приведет к ошибке выполнения:

```
print(tom.__age)
```

Инкапсуляция

- ▶ Однако все же нам может потребоваться устанавливать возраст пользователя из вне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута:

```
def get_age(self):  
    return self.__age
```

метод, который используется в ООП для доступа к частным атрибутам класса.

Данный метод еще часто называют **геттер**.

Инкапсуляция

- ▶ Для изменения возраста определено другое свойство:

```
def set_age(self, value):  
    if value in range(1, 100):  
        self.__age = value  
    else:  
        print("Недопустимый возраст")
```

Здесь мы уже можем решить в зависимости от условий, надо ли переустанавливать возраст. Данный метод еще называют **setter**

Необязательно создавать для каждого приватного атрибута подобную пару свойств. Так, в примере выше имя человека мы можем установить только из конструктора. А для получения определен метод **get_name**.

Аннотации свойств

- ▶ Выше мы рассмотрели, как создавать свойства. Но Python имеет также еще один - более элегантный способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом @.
- ▶ Для создания свойства-геттера над свойством ставится аннотация **@property**.
@property - один из встроенных декораторов Python. Основная цель любого декоратора - изменить методы или атрибуты класса, чтобы пользователю класса не нужно было изменять свой код.
- ▶ Для создания свойства-сеттера над свойством устанавливается аннотация **@имя_свойства_геттера.setter**.

Перепишем класс с использованием аннотаций

```
class Person:
```

```
    def __init__(self, name):
```

```
        self.__name = name # устанавливаем имя
```

```
        self.__age = 1     # устанавливаем возраст
```

```
    @property
```

```
    def age(self):
```

```
        return self.__age
```

```
    @age.setter
```

```
    def age(self, age):
```

```
        if age in range(1, 100):
```

```
            self.__age = age
```

```
        else:
```

Потылицина Е.М.

```
            print("Недопустимый возраст")
```

@property

```
def name(self):
```

```
    return self.__name
```

```
def display_info(self):
```

```
    print("Имя:", self.__name, "\tВозраст:", self.__age)
```

```
tom = Person("Tom")
```

```
tom.display_info()    # Имя: Том  Возраст: 1
```

```
tom.age = -3486       # Недопустимый возраст
```

```
print(tom.age)        # 1
```

```
tom.age = 36
```

```
tom.display_info()    # Имя: Том  Возраст: 36
```



```
@age.deleter
```

```
def age(self):
```

```
    del self.__age
```

метод вызывается при удалении свойства

Аннотации

- ▶ Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.
- ▶ Во-вторых, и сеттер, и геттер называются одинаково - `age`. И поскольку геттер называется `age`, то над сеттером устанавливается аннотация `@age.setter`.
- ▶ После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение `tom.age`.

Наследование

- ▶ Наследование позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является одним из краеугольных камней объектно-ориентированного дизайна.
- ▶ Ключевыми понятиями наследования являются **подкласс** и **суперкласс**. Подкласс наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).
- ▶ Синтаксис для наследования классов выглядит следующим образом:

```
class подкласс (суперкласс):  
    методы_подкласса
```

Наследование

- ▶ Например, ранее был создан класс `Person`, который представляет человека. Предположим, нам необходим класс работника, который работает на некотором предприятии.
- ▶ Мы могли бы создать с нуля новый класс, к примеру, класс `Employee`. Однако он может иметь те же атрибуты и методы, что и класс `Person`, так как сотрудник - это человек.
- ▶ Поэтому нет смысла определять в классе `Employee` тот же функционал, что и в классе `Person`. И в этом случае лучше применить наследование.

Унаследуем класс Employee от класса Person

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.__name = name # устанавливаем имя
```

```
        self.__age = age # устанавливаем возраст
```

```
    @property
```

```
    def age(self):
```

```
        return self.__age
```

```
    @age.setter
```

```
    def age(self, age):
```

```
        if age in range(1, 100):
```

```
            self.__age = age
```

```
        else:
```

```
            print("Недопустимый возраст")
```

@property

```
def name(self):  
    return self.__name
```

```
def display_info(self):  
    print("Имя:", self.__name, "\tВозраст:", self.__age)
```

```
class Employee(Person):
```

```
    def details(self, company):  
        # print(self.__name, "работает в компании", company) # так нельзя, self.__name -  
        # приватный атрибут  
        print(self.name, "работает в компании", company)
```

```
tom = Employee("Tom", 23)
```

```
tom.details("Google")
```

```
tom.age = 33
```

```
tom.display_info()
```

Потылицина Е.М.

Наследование

- ▶ Класс `Employee` полностью перенимает функционал класса `Person` и в дополнении к нему добавляет метод **`details()`**.
- ▶ Стоит обратить внимание, что для `Employee` доступны через ключевое слово `self` все методы и атрибуты класса `Person`, кроме закрытых атрибутов типа `__name` или `__age`.
- ▶ При создании объекта `Employee` мы фактически используем конструктор класса `Person`. И кроме того, у этого объекта мы можем вызвать все методы класса `Person`.

Полиморфизм

- ▶ Полиморфизм является еще одним базовым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса.
- ▶ Например, пусть у нас будет следующая иерархия классов:

class Person:

def __init__(self, name, age):

self.__name = name # устанавливаем имя

self.__age = age # устанавливаем возраст

@property

def name(self):

return self.__name

@property

def age(self):

return self.__age

@age.setter

def age(self, age):

if age in range(1, 100):

self.__age = age

else:

print("Недопустимый возраст")

```
def display_info(self):  
    print("Имя:", self.__name, "\tВозраст:", self.__age)
```

```
class Employee(Person):  
    # определение конструктора  
    def __init__(self, name, age, company):  
        Person.__init__(self, name, age)  
        self.company = company  
  
    # переопределение метода display_info  
    def display_info(self):  
        Person.display_info(self)  
        print("Компания:", self.company)
```

```
class Student(Person):
    # определение конструктора
    def __init__(self, name, age, university):
        Person.__init__(self, name, age)
        self.university = university

    # переопределение метода display_info
    def display_info(self):
        print("Студент", self.name, "учится в университете", self.university)

people = [Person("Tom", 23), Student("Bob", 19, "Harvard"), Employee("Sam", 35,
"Google")]

for person in people:
    person.display_info()
    print()
```

Полиморфизм

- ▶ В производном классе Employee, который представляет служащего, определяется свой конструктор. Так как нам надо устанавливать при создании объекта еще и компанию, где работает сотрудник. Для этого конструктор принимает четыре параметра: стандартный параметр self, параметры name и age и параметр company.
- ▶ В самом конструкторе Employee вызывается конструктор базового класса Person. Обращение к методам базового класса имеет следующий синтаксис:

суперкласс.название_метода(self [, параметры])

Полиморфизм

- ▶ Поэтому в конструктор базового класса передаются имя и возраст. Сам же класс `Employee` добавляет к функционалу класса `Person` еще один атрибут - `self.company`.
- ▶ Кроме того, класс `Employee` переопределяет метод `display_info()` класса `Person`, поскольку кроме имени и возраста необходимо выводить еще и компанию, в которой работает служащий. И чтобы повторно не писать код вывода имени и возраста здесь также происходит обращение к методу базового класса - методу `get_info`: `Person.display_info(self)`.
- ▶ Похожим образом определен класс `Student`, представляющий студента. Он также переопределяет конструктор и метод `display_info` за тем исключением, что вместо в методе `display_info` не вызывается версия этого метода из базового класса.

Полиморфизм

- В основной части программы создается список из трех объектов Person, в котором два объекта также представляют классы Employee и Student. И в цикле этот список перебирается, и для каждого объекта в списке вызывается метод `display_info`. На этапе выполнения программы Python учитывает иерархию наследования и выбирает нужную версию метода `display_info()` для каждого объекта. В итоге мы получим следующий консольный вывод:

Имя: Tom Возраст: 23

Студент Bob учится в университете Harvard

Имя: Sam Возраст: 35

Компания: Google

Проверка типа объекта

- ▶ При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции `isinstance()` мы можем проверить тип объекта. Эта функция принимает два параметра:

`isinstance(object, type)`

- ▶ Первый параметр представляет объект, а второй - тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает `True`

Пример

- ▶ Например, возьмем выше описанную иерархию классов:

```
for person in people:
    if isinstance(person, Student):
        print(person.university)
    elif isinstance(person, Employee):
        print(person.company)
    else:
        print(person.name)
    print()
```


Проектирование программы

В ООП очень важно предварительное проектирование. В общей сложности можно выделить следующие этапы разработки объектно-ориентированной программы:

1. Формулирование задачи.
2. Определение объектов, участвующих в ее решении.
3. Проектирование классов, на основе которых будут создаваться объекты. В случае необходимости установление между классами наследственных связей.
4. Определение ключевых для данной задачи свойств и методов объектов.
5. Создание классов, определение их полей и методов.
6. Создание объектов.
7. Решение задачи путем организации взаимодействия объектов.

Практика

1. Напишите программу по следующему описанию. Есть класс "Воин". От него создаются два экземпляра-юнита. Каждому устанавливается здоровье в 100 очков. В случайном порядке они бьют друг друга. Тот, кто бьет, здоровья не теряет. У того, кого бьют, оно уменьшается на случайное количество очков от одного удара (можно также указать куда пришелся удар). После каждого удара надо выводить сообщение, какой юнит атаковал, и сколько у противника осталось здоровья. Как только у кого-то заканчивается ресурс здоровья, программа завершается сообщением о том, кто одержал победу.

Практика

2. Создайте класс «Animal» с тремя атрибутами и двумя методами. Создайте подклассы «Elefant», «Dog», «Cat» с дополнительными атрибутами и методами (свойственными каждому животному). Выведите информацию о них на экран.

3. Создайте класс Person, подкласс Student, у студентов есть средний балл по успеваемости. Создайте несколько объектов этого класса. Выведите на экран информацию о них с указанием получают ли они стипендию в следующем семестре (если средний балл ≥ 4)