




Программирование на языке Python

Кондаков Айсен Алексеевич

к.ф.-м.н., директор Центра развития цифровых
компетенций и онлайн образования
СВФУ имени М.К. Аммосова



Объектно-ориентированное программирование в Python

► Парадигмы программирования

Существует несколько подходов к программированию — парадигм программирования.

Python реализует не только процедурное (структурное) программирование, но и объектно-ориентированное.

Императивное программирование — программа пишется как последовательность инструкций для компьютера.

Процедурное (структурное) программирование — это развитие императивной парадигмы программирования. В программе создаются вспомогательные блоки (функции, модули, пакеты) каждый из которых выполняет небольшую часть работы программы.

Объектно-ориентированное программирование — программист описывает поведение программных объектов (которые моделируют поведение реальных объектов), программа является взаимодействием таких объектов.

Терминология ООП

Класс — это тип данных. Например строки (str) или списки (list) — это классы. Классы могут моделировать категории реальных или абстрактных объектов. Например можно написать класс Auto для описания различных автомобилей или можно написать класс Character для описания персонажей игры.

Имена классов следует начинать с заглавной буквы. Если имя класса состоит из нескольких слов — начинайте каждое новое слово с заглавной буквы (CamelCase).

Объект — это переменная некоторого типа данных (конкретный экземпляр класса). В программе можно создать несколько переменных одного и того же класса (например несколько строк или списков). Все они будут вести себя одинаково, но будут хранить разные данные.

Имена объектов (переменных) следует писать строчными буквами. Если имя состоит из нескольких слов — соедините их с помощью символа подчеркивания (snake case).

Атрибут — это часть данных, которые хранит в себе объект. Например если объект описывает автомобиль, то это может быть мощность двигателя или количество бензина в баке.

Атрибуты могут быть индивидуальные для каждого объекта, а могут быть общие для всего класса (статические).

Чтобы обратиться к атрибуту, нужно написать его имя после точки.

Метод — это функция, которая привязана к классу. Методы описывают поведение объектов класса. Чтобы обратиться к методу, нужно написать его имя после точки. Чтобы вызвать его, нужно написать справа круглые скобки.

Имена объектов, атрибутов и методов следует писать строчными буквами. Если имя состоит из нескольких слов — соедините их с помощью символа подчеркивания (snake case).

Для объявления класса служит ключевое слово `class`.

Минимальное объявление класса выглядит так:

```
class Dog:  
    pass
```

Чтобы создать объект класса нужно написать справа круглые скобки (аналогично вызову функции):

```
sharik = Dog()
```

В классе могут объявлены атрибуты (общие для всех объектов класса):

```
class Dog:  
    legs = 4
```

Для каждого объекта класса могут быть свои атрибуты (индивидуальные для объекта):

```
sharik = Dog()  
sharik.name = 'Шарик'  
lassie = Dog()  
lassie.name = 'Лесси'  
print(sharik.name)  # Шарик
```

Каждому методу класса неявно передается аргумент `self` — указатель на объект класса:

```
class Dog():  
    def gav(self):  
        print(f'Привет, меня зовут {self.name}')  
sharik = Dog()  
sharik.name = 'Шарик'  
sharik.gav() # Привет, меня зовут Шарик
```

Атрибуты объектов могут быть созданы динамически внутри методов:

```
class Dog():  
    def set_name(self, name):  
        self.name = name  
  
sharik = Dog()  
sharik.set_name('Шарик')  
print(sharik.name) # Шарик
```

Инкапсуляция

Для того, чтобы показать, что атрибут или метод не должны использоваться за его пределами, следует начать его имя с символа "_" (подчеркивание).

Для того, чтобы запретить использование атрибута или метода следует начать его идентификатор с символов "__" (двойное подчеркивание).

```
class Batman:
    def _secret(self):
        print('Robin is Dick Grayson')

    def __supersecret(self):
        print('Batman is Bruce Wayne')

batman = Batman()
batman._secret() # 'Robin is Dick Grayson'
batman.__supersecret() # Ошибка
```

Инкапсуляция — это принцип ООП, согласно которому следует разделять внутреннюю и внешнюю стороны объектов. Внешняя сторона (интерфейс) описывает как объект взаимодействует с другими объектами. Внутренняя сторона (реализация) описывает как он устроен и как работает.

Принципа инкапсуляции приводит к тому что если в объекте измененились детали внутренней реализации без изменения внешнего интерфейса, то программа в целом продолжит работать как прежде.

Наследование

Наследование — это принцип ООП согласно которому класс может унаследовать от другого класса его атрибуты и методы.

Исходный класс называется **базовым** или **родительским** (иногда **суперклассом**).

Порожденный класс называется **производным** или **дочерним** (иногда **подклассом**).

```
class Animal:
    def test(self):
        print('Меня зовут', self.name)

class Dog(Animal): # базовый класс в скобках
    def bark(self):
        self.test()

class Cat(Animal): # базовый класс в скобках
    def meow(self):
        self.test()

gav = Cat()
gav.name = 'Гав'
gav.meow() # Меня зовут Гав
gav.test() # Меня зовут Гав
gav.bark() # Ошибка
```


Python поддерживает множественное наследование:

```
class Father:
    chromosome_Y = True

class Mother:
    chromosome_X = True

class Child(Father, Mother):
    pass

child = Child()
dir(child)  # chromosome_Y, chromosome_X
```

Производный класс может модифицировать поведение базового класса. То есть, в производном классе можно переопределить атрибуты и методы базового класса.

При поиске конкретного атрибута или метода сначала просматривается производный класс, потом его базовый класс и т.д.

Если использовано множественное наследование, то классы на одном уровне просматриваются слева направо.

Полиморфизм — это принцип ООП согласно которому если функция умеет работать с объектами базового класса, то она без изменений может работать с объектами производных классов.

При этом функция не знает объект какого конкретно класса её передадут, базового или одного из производных.

```
class Animal:
    def test(self):
        print('I am an animal')

class Dog(Animal):
    pass

class Cat(Animal):
    def test(self):
        print('I am a cat')

def test_animal(animal):
    # Функция не знает какой конкретно метод будет вызван
    animal.test()

sharik = Dog()
test_animal(sharik) # I am an animal

gav = Cat()
test_animal(gav) # I am a cat
```

Конструктор

Метод с именем `__init__()` вызывается при создании объекта и называется конструктором:

```
class Dog:
    name = ''
    def __init__(self, name):
        self.name = name

sharik = Dog('Шарик')
print(sharik.name)  # Шарик
```

Функция super()

Функция super() позволяет обратиться к родительскому классу:

```
class Animal:
    name = ''
    def __init__(self, name):
        name = self.name

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

lassie = Dog('Лесси', 'Колли')
print(lassie.name)  # Лесси
print(lassie.breed) # Колли
```

Методы класса

В классе можно создать метод который привязан к классу, а не к объекту:

```
class Car:
    tax_rate = 0.5

    @classmethod
    def set_tax_rate(cls, new_tax_rate):
        # данный метод изменяет общий для всех автомобилей процент налога
        cls.tax_rate = new_tax_rate

    def __init__(self, model, price):
        self.model = model
        self.price = price

    def get_tax(self):
        return self.price * self.tax_rate / 100.0

camry = Car('Toyota Camry', 2_000_000)
largus = Car('Lada Largus', 1_000_000)
print(camry.get_tax()) # 10000
print(largus.get_tax()) # 5000
Car.set_tax_rate(0.2)
print(camry.get_tax()) # 4000
print(largus.get_tax()) # 2000
```



Свойства (динамические атрибуты)

В классе можно создать **свойства** — методы, к которым можно обращаться как к атрибутам класса:

```
class Person:
    def __init__(self, last_name, first_name):
        self.last_name = last_name
        self.first_name = first_name

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

# обращаемся к full_name как будто это атрибут, а не метод
p = Person('Smith', 'John')
print(p.full_name) # John Smith
```

Статические методы

В классе можно создать метод, который привязан к классу только формально.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def hello(self):
        return f"Привет, я {self.breed} по кличке {self.name}!"

    @staticmethod
    def new_bulldog(name):
        return Dog(name, 'бульдог')

    @staticmethod
    def new_kolli(name):
        return Dog(name, 'колли')

lassie = Dog.new_kolli('Лесси')
print(lassie.hello()) # Привет, я колли по кличке Лесси!
```

Тема 7. Модули и пакеты

Введение

Модуль — это скрипт на языке Python (файл с расширением `*.py`).

Пакет — это каталог с файлом `__init__.py`. Файл `__init__.py` является заголовком пакета и содержит перечень идентификаторов, которые становятся доступными при импорте пакета.

Пакет состоит из некоторого количества вложенных модулей и пакетов.

Подключение модуля или пакета происходит с помощью ключевых слов `import` и `from`. При этом нужно указать имя каталога (если подключается пакет) или имя файла без расширения `*.py` (если подключается модуль).

При подключении модуля или пакета происходит выполнение импортируемого модуля или заголовка пакета. В ходе выполнения Python регистрирует имена созданных объектов:

- модулей/пакетов
- переменных
- функций
- классов

```
# при импорте Python выполняет модуль/пакет и дает доступ к его именам
import os
```

```
# обращаемся к переменной по имени path из модуля sys
print(os.getcwd())
```


При импорте Python просматривает определенный перечень каталогов которые хранятся в списке `path` в модуле `sys`. В каждом каталоге он ищет нужный модуль/пакет. Как только он найден, поиск прекращается. Текущий каталог также входит в этот перечень и просматривается в первую очередь.

Если модуль/пакет импортируется повторно, то повторного выполнения не происходит. Вместо этого все необходимые имена берутся из кеша созданного при первом импорте.

При импорте программист может указать имя пакета/модуля в целом, либо указать определенные имена из него:

```
# Импорт модулей/пакетов в целом
import os

# Импортирование может перемежаться со строками обычного кода,
# однако рекомендуется по возможности расположить импорты в начале модуля
print(os.getcwd())

# Импорт определенных имен из модуля/пакета
from datetime import date
from math import cos, sin, PI

# Если из модуля импортируется очень много имен, то рекомендуется заключить их
# в скобки и разбить на несколько строк
from collections import (namedtuple, deque, ChainMap, Counter, OrderedDict,
                          defaultdict, UserDict, UserList, UserString)

print(PI)
```



Псевдонимы

При импорте можно назначить имени исходного объекта псевдоним (для удобства использования, или чтобы избежать конфликта имен):

```
import math as m
from math import sin as s

print(s(m.PI))
```

Вложенные пакеты

Внутри пакета можно создать вложенные пакеты. Глубина вложенности не ограничивается, но на практике достаточно 2-3 уровней вложенности.

Чтобы импортировать имена из вложенного модуля/пакета нужно последовательно указать цепочку пакетов соединённых точкой:

```
from os.path import getsize

print(f"This file is {getsize(__file__)} bytes long.")
```

🔗 Относительная адресация

Представим ситуацию, когда внутри некоторого проекта создан пакет `package`. Внутри него находятся два модуля `module1.py` и `module2.py`. Тогда импортировать имена из одного в соседний возможно двумя разными способами:

```
# обычный способ
from package.module1 import func1

# относительная адресация, обратите внимание на точку перед именем модуля
from .module1 import func2

print(func2())
```

Относительная адресация использует точку перед именем модуля/пакета, чтобы указать что он берется из текущего пакета.

Относительная адресация использует две точки перед именем модуля/пакета, чтобы указать что он берется из родительского пакета.

Возможно и большее количество точек перед именем модуля/пакета, но обычно этого не требуется.



Пакеты сторонних разработчиков

На сайте <http://pypi.org/> перечислены пакеты опубликованные сторонними разработчиками.

Скрипт `pip` позволяет устанавливать такие пакеты:

```
pip install flask
```

Часто для корректной работы одному пакету нужно наличие какого-то другого пакета.

Скрипт `pip` автоматически находит необходимые пакеты и скачивает их из Интернета по ссылкам с сайта <http://pypi.org/>.

Бинарные пакеты

Python написан на языках C/C++ и имеет возможность вызывать определенным образом подготовленные функции из этих языков. Поэтому в Python часто применяются модули и пакеты написанные на C/C++. Такие модули называются бинарными.

Т.к. C/C++ — это компилируемые языки, то бинарные модули перед использованием должны быть скомпилированы (при этом их исходный код превращается в машинный код). При компиляции они привязываются:

- к определенной операционной системе;
- к архитектуре процессора;
- к версии интерпретатора Python (учитываются только первые две цифры в версии, третья — незначительна).

Например, если бинарный пакет скомпилирован для Windows, то его нельзя использовать на macOS или Linux. Если он скомпилирован для Python 3.9, то его нельзя использовать на Python 3.10.

При публикации бинарных пакетов авторы компилируют его для наиболее популярных комбинаций ОС / процессор / интерпретатор. Если подходящего для вас бинарного пакета нет (например если вы установили свежую версию Python которая вышла меньше месяца назад), то pip попытается скомпилировать его на вашем компьютере. Для этого нужен компилятор C/C++, однако если он не найден или не подходит, то pip выдаст ошибку.

Использование бинарных пакетов позволяет значительно эффективнее использовать ресурсы компьютера, чем обычный пакет на Python.

Задание на тему «ООП: Классы и объекты»

- Создать классы и соответствующие объекты «игрового мира», которые будут иметь атрибуты, характеризующие «здоровье», «силу» (защиту, критический урон, шанс попадания) и методы позволяющие производить воздействия на другие объекты («атака», «лечение», «воскрешение»...).
- Идеи для сюжета: “Dogs & Cats”, “StarWars”, Танки, Корабли, Вампиры vs оборотни...