ООП

С точки зрения ООП, класс представляет собой коллекцию данных. Фактически, классы – это абстракция.

Объектно-ориентированный подход строится на следующих принципах:

- Полиморфизм: в разных объектах одна и та же операция может выполнять различные функции. Слово «полиморфизм» имеет греческую природу и означает «имеющий многие формы». Простым примером полиморфизма может служить функция count(), выполняющая одинаковое действие для различных типов обьектов: 'abc'.count('a') и [1, 2, 'a'].count('a'). Оператор плюс полиморфичен при сложении чисел и при сложении строк.
- Инкапсуляция: можно скрыть ненужные внутренние подробности работы объекта от окружающего мира. Это второй основной принцип абстракции. Он основан на использовании атрибутов внутри класса. Атрибуты могут иметь различные состояния в промежутках между вызовами методов класса, вследствие чего сам объект данного класса также получает различные состояния — state.
- Наследование: можно создавать специализированные классы на основе базовых. Это позволяет нам избегать написания повторного кода.
- Композиция: объект может быть составным и включать в себя другие объекты.

Объектно-ориентированный подход в программировании подразумевает следующий алгоритм действий.

- На основе понятий формулируются классы.
- На основе действий проектируются методы.
- Реализуются методы и атрибуты.
- Мы получили скелет — объектную модель. На основе этой модели реализуется наследование.

Объектно-ориентированный подход хорош там, где проект подразумевает долгосрочное развитие, состоит из большого количества библиотек и внутренних связей.

Наиболее важные особенности классов в питоне:

- множественное наследование;
- производный класс может переопределить любые методы базовых классов;
- в любом месте можно вызвать метод с тем же именем базового класса;
- все атрибуты класса в питоне по умолчанию являются public, т.е. доступны отовсюду; все методы — виртуальные, т.е. перегружают базовые.

Фактически, класс — это пользовательский тип данных. Простейшая модель определения класса выглядит следующим образом:

```
class имя:
    инструкция1
    инструкция…
```

Класс состоит из объявления (инструкция class), имени класса и тела класса, которое содержит атрибуты и методы.

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

```
имя_объекта = имя_класса()
```

Класс может содержать атрибуты и методы. Ниже представлен класс, содержащий атрибуты color (цвет), width (ширина), height (высота).

```
class Figure:
    color = “green”
    width = 100
    height = 100
```

Доступ к атрибуту класса можно получить следующим образом.

имя_объекта.атрибут

```
fig1 = Figure()
print(fig1.color)
```

Добавим к нашему классу метод. Метод – это функция внутри класса. Например, нашему классу Figure, можно добавить метод, считающий площадь прямоугольника. Для того, чтобы метод в классе «знал», с каким объектом он работает (это нужно для того, чтобы получить доступ к атрибутам: ширина (width) и высота (height)), первым аргументом ему следует передать параметр self, через который он может получить доступ к своим данным.

```
class Figure
    color = "green"
    width = 100
    height = 100
    def square(self):
        return self.width * self.height
```

Конструктор класса

Конструктор класса позволяет задать определенные параметры объекта при его создании. Конструктором класса является метод:

```
__init__(self)
```

Для того, чтобы иметь возможность задать цвет, длину и ширину прямоугольника при его создании, добавим к классу Figure следующий конструктор:

```
class Figure:
    def __init__(self, color="green", width=100, height=100):
        self.color = color
        self.width = width
        self.height = height
    def square(self):
        return self.width * self.height
fig1 = Figure()
print(fig1.color)
```

```
print(fig1.square())
fig1 = Figure("yellow", 23, 34)
print(fig1.color)
print(fig1.square())
```

Наследование

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка есть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в Python можно его использовать.

Синтаксически создание класса с указанием его родителя/ей выглядит так:

```
class имя_класса(имя_родителя1, [имя_родителя2,…, имя_родителя_n])
```

```
class Figure:
    def __init__(self, color):
        self.color = color
    def get_color(self):
        return self.color
class Rectangle(Figure):
    def __init__(self, color, width=100, height=100):
        super().__init__(color)
        self.width = width
        self.height = height

    def square(self):
        return self.width*self.height
fig1 = Rectangle("blue")
print(fig1.get_color())
print(fig1.square())
fig2 = Rectangle("red", 25, 70)
print(fig2.get_color())
print(fig2.square())
```

Функции

Именные функции

Функция в python - объект, принимающий аргументы и возвращающий значение. Функция определяется с помощью инструкции def.

Определим простейшую функцию:

```
def plus(x, y):
    return x + y
```

Инструкция return говорит, что нужно вернуть значение (return). В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

```
>>>plus(1, 9)
10
>>> plus('abc', 'defg')
'abcdefg'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
def newfunc(n):
     def myfunc(x):
         return x + n
     return myfunc
>>> new = newfunc(100)
>>> new(200)
300
```

Здесь, new – это функция.

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

Аргументы функции

Функция может принимать произвольное количество аргументов или не принимать их вовсе.

```
>>> def func(a, b, c=2):
     return a + b + c
>>> func(1, 2)
5
>>> func(1, 2, 4)
7
```

```
>>> func(a=1, b=3)
6
>>> func(a=3, c=6)
```

выдаст ошибку

Здесь c – необязательный аргумент.

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится *:

```
>>> def func(*args):
...     return args
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
(1,)
```

Как видно из примера, args - это кортеж из всех переданных аргументов функции.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
>>> def func(**kwargs):
...     return kwargs
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

В переменной kwargs хранится словарь.

Анонимные функции

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции lambda. Кроме этого, их не обязательно присваивать переменной:

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
```

```
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

lambda функции, в отличие от обычной, не требуется инструкция return:

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```