Программирование в лингвистике

Объектно-ориентированное программирование

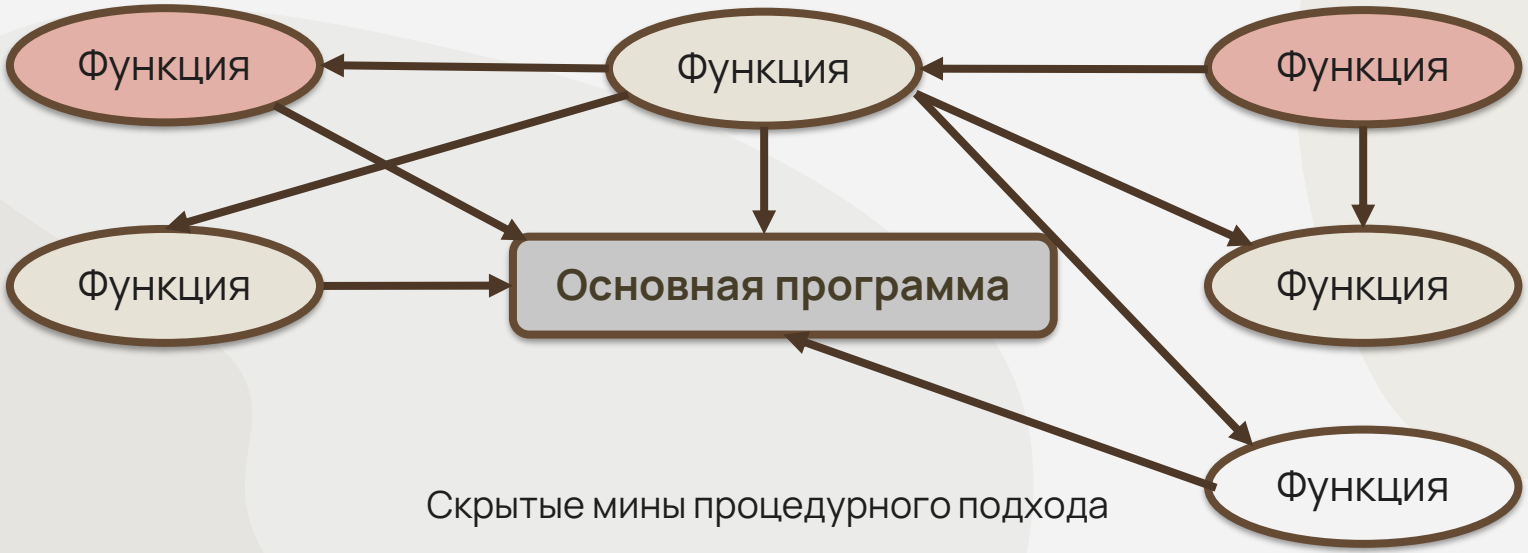
ООП VS. процедурное программирование

В то время как в объектно-ориентированном программировании центральным понятием является **объект (класс)**, процедурное программирование опирается на понятия **программы и подпрограммы (функции)**. Функции эти могут использоваться как программой, так и другими функциями:

Схема взаимодействия функций с основной программой и между собой

ООП VS. процедурное программирование

У процедурного программирования есть существенный недостаток — **части кода сильно зависят друг от друга**. Например, основная программа вызывает функцию, та вызывает вторую, та, в свою очередь, — третью. При этом, допустим, вторую функцию могут параллельно вызывать ещё несколько других, а также основная программа:



Скрытые мины процедурного подхода

ООП VS. процедурное программирование

Таким образом, при изменении одной функции **посыплется вся программа**. Проще будет переписать ее с нуля.

А что же в объектно-ориентированном программировании?

Схема ООП. Объекты независимы друг от друга и самодостаточны.

ООП

Итак, центральное понятие ООП - это **класс**. Весь мир в ООП - это объекты, которые можно объединить в какие-то классы, а классы выстроить в иерархию. Здесь можно вспомнить любую семантическую онтологию, которая пытается выстроить познания человека о вселенной с помощью иерархического дерева, или биологическую классификацию.

```
class Cat:
  def __init__(self, name):
    self.name = name
  def showname(self):
    print(f'Кота зовут {self.name}.')

cat = Cat('Барсик')
cat.showname()
```

Атрибут

Метод

Экземпляр

```
Кота зовут Барсик.
```

Класс - это абстрактная категория объектов, которая выделяется на основании каких-то **характеристик (атрибутов и методов)**, которые присущи именно этим объектам.

Если класс – подлежащее, то атрибут – определение (что у него есть?), а метод – сказуемое (что он умеет?).

Термины, связанные с ООП

Интерфейс	**набор методов класса**, доступных для использования **извне**.
Инкапсуляция	**свойство системы**, позволяющее **объединить** данные и методы, работающие с ними, в классе и **скрыть детали реализации** от пользователя.
Абстракция	**способ выделить набор значимых** характеристик объекта, **исключая** из рассмотрения **незначимые**. Соответственно, абстракция - это набор всех таких характеристик.
Полиморфизм	**свойство системы одинаково** использовать объекты **с одинаковым интерфейсом** без информации о типе и внутренней структуре объекта.

Рассмотрим пример класса

```
class Human:
  eyes = 2
  ears = 2

  def stats(self):
    print(f'''У меня {self.eyes} глаза
    и {self.ears} уха.''')

vasya = Human()
vasya.ears += 1
vasya.stats()
```

```
У меня 2 глаза
    и 3 уха.
```

› **class Human** – элемент синтаксиса Python, предполагающий начало класса. Human – название класса, пишется с большой буквы.

› **eyes и ears** – атрибуты класса с изначальным значением 2.

› **def stats** – метод экземпляра класса. Всегда требует self.

› **print(….)** – тело метода. Печатает текущие значения атрибутов ears и eyes.

› **vasya = Human()** – создание экземпляра класса. Этот экземпляр будет обладать всеми методами и атрибутами класса Human.

› **vasya.ears += 1** – изменение значения атрибута экземпляра класса. Теперь у этого экземпляра класса будет 3 уха.

› **vasya.stats()** – вызов метода stats.

Магический метод __init__

Init означает **initialize** (инициализировать), этот метод служит **конструктором класса**. Так как метод init **явным образом не вызывается**, он называется **магическим**.

Метод init срабатывает **в момент создания** экземпляра класса – тогда, когда мы пишем **petya = Human()**. Благодаря ему мы можем передавать в наш класс аргументы, строить структуру класса, вызывать методы и все это произойдет прямо в момент создания экземпляра класса. Изменение метода init в классе называется **перегрузкой метода**.

Все методы с двумя подчеркиваниями слева и справа являются **магическими**.

```
class Human:
  def __init__(self, name):
    self.name = name
    print(f'Человека зовут {self.name}.')

petya = Human('Петя')
```

```
Человека зовут Петя.
```

Что такое self?

Мы желаем, чтобы наши методы и атрибуты были применимы **к любому экземпляру** класса, поэтому нам нужно к чему-то их применять. Но к чему?

Тут на помощь придет **абстракция**: мы придумаем **переменную** для нашего будущего экземпляра, в нашем определении класса она займет место **любого абстрактного экземпляра класса** и будет выполнять его роль. Когда появится экземпляр класса, он будет автоматически подставлен на место этой переменной.

У этой переменной есть общепринятое имя: **self**.

```
class Human:
    eyes = 2
    def __init__(self, name, age):
        self.name = name
        self.age = age
vasya = Human('Vasya', 21)
```

Магический метод __init__

```
class Human:
    eyes = 2                    Статический атрибут
    def __init__(self, name, age):
        self.name = name        Динамические
        self.age = age          атрибуты

vasya = Human('Vasya', 21)
petya = Human('Petya', 25)
vasya.name, petya.name
```

```
('Vasya', 'Petya')
```

Init здесь выступает как **анкета** – она принимает имя и возраст, а затем присваивает их экземпляру класса **как атрибуты**. Так как init – это метод (по сути та же функция), name и age – это просто **параметры** метода, а передаваемые 'Vasya' и 21 – **аргументы** метода.

Соответственно, эти параметры мы можем назвать, как мы хотим, и аргументы в них можем передавать произвольные.

self в init важно указывать, поскольку без нее мы создадим **локальную переменную метода**, а не атрибут, поэтому мы **не сможем** обращаться к этим переменным **извне** или **из другого метода** этого класса.

Магические методы __str__ и __repr__

```
print(vasya)

<__main__.Human object at 0x78ac11bff0d0>
```

То, что сейчас выводит Python, когда мы пытаемся напечатать наш объект – неявно **унаследованный** от **дефолтного класса Object** шаблон.

Наследование – заимствование всех методов и атрибутов родительского класса дочерним классом с возможностью их перегрузки и дополнения новыми.

Чтобы наш класс выглядел более симпатично, мы можем перегрузить магические методы ___str___ и ___repr___.

Метод ___str___ неявным образом вызывается функцией **print** (которая все должна превратить в строки, чтобы напечатать) и некоторыми другими.

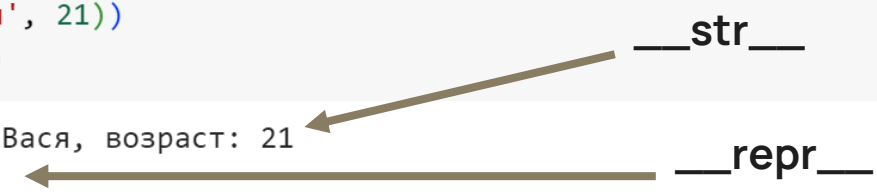
Метод ___repr___ неявно вызывается **в интерактивной среде разработки**, когда мы хотим просто вывести свой объект на экран, а также **в списке**.

Магические методы __str__ и __repr__

```
class Human:
    eyes = 2
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f'Человек по имени {self.name}, возраст: {self.age}'
    def __repr__(self):
        '''Метод repr принято писать таким образом, чтобы он возвращал строчку,
        полностью совпадающую с той, которую вам нужно написать, чтобы
        завести такой экземпляр класса'''
        return f"Human('{self.name}', {self.age})"

print(Human('Вася', 21))
Human('Вася', 21)
```

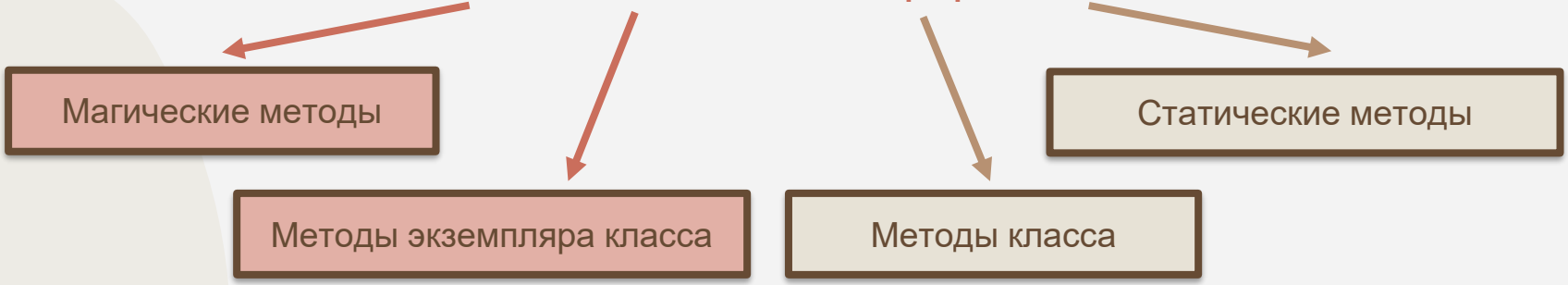
```
Человек по имени Вася, возраст: 21
Human('Вася', 21)
```



Методы в ООП Python

Итак, из магических методов, как правило, мы перегружаем метод ___init___ (его не нужно перегружать только в каких-то очень редких случаях) и иногда ___str___ и ___repr___ - эти для красоты и программистской вежливости.

В целом, методы делятся на четыре разновидности:



В рамках базового курса мы рассмотрели **магические методы** и **методы экземпляра класса**.

Любопытным: метод класса принимает вместо экземпляра self сам класс, а статический метод вообще ничего не принимает и не привязан ни к классу, ни к экземпляру.

Пример кода

```python
class Human:
    eyes = 2
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def showage(self):
        x, xx = self.age % 10, self.age % 100
        years = 'лет'
        if not 11 < xx < 15:
            if x == 1:
                years = 'год'
            elif x in {2, 3, 4}:
                years = 'года'
        print(f'Человек {self.name}: {self.age} {years}')

    def compare(self, other):
        return self.age > other.age
```

› **init** – магический метод, в котором мы принимаем аргументы name и age и превращаем их в **атрибуты** экземпляра класса.

› **showage** – метод экземпляра класса. Используя атрибуты name и age, он выводит **грамматичную информацию** об имени и возрасте человека.

› **compare** – метод экземпляра класса, который предполагает передачу в него **второго** экземпляра класса Human. Он сравнивает возрасты двух людей и выводит bool, если **этот** человек старше второго.

Вызов методов

```
vasya = Human('Вася', 21)
petya = Human('Петя', 25)

vasya.showage()
print(petya.compare(vasya))
print(Human.compare(petya, vasya))
```

```
Человек Вася: 21 год
True
True
```

В __init__ при создании экземпляра класса **vasya** передались аргументы **‘Вася’ и 21**, которые соответственно положились в атрибуты **name и age**.

Метод **showage** сработал верно, используя эти два атрибута.

Также, как мы видим, метод **compare** мы можем вызывать как от **экземпляра класса**, так и от самого класса **Human**.

Результат один и тот же – **Петя старше Васи**.

Для чего это все нам?

Почти все библиотеки для лингвистов написаны **с использованием классов**. Важно уметь распознавать классы, а также их атрибуты и методы.

К примеру, всем знакомый счетчик **Counter** и дефолтный словарь **defaultdict** - не что иное, как **классы**, унаследовавшие методы и атрибуты от встроенного типа данных **dict**.

Методы экземпляра класса есть почти у всех встроенных типов данных в Python (кроме None), а атрибуты – редкость.

Для любопытных существует функция **dir()**, которая возвращает все атрибуты и методы объекта-аргумента.