

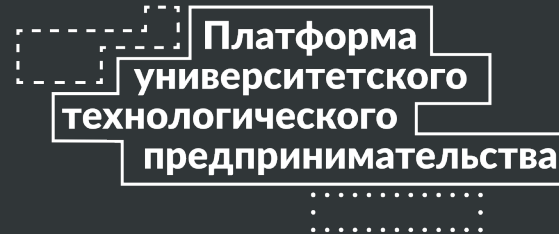


Мастер-класс



Python и
примеры реализации
бизнес-процессов с помощью ИИ

31.10.2022



ООП в Python

Python поддерживает ООП на сто процентов: все данные в нём являются объектами.

Числа всех типов, строки, списки, словари, даже функции, модули, и наконец, сами типы данных — всё это объекты!

Все вычисления в Python можно представить как взаимодействия между объектами.

Основные понятия

Как узнать класс объекта?

```
type(123)
```

```
# => '<class 'int'>'
```

```
type([1, 2, 3])
```

```
# => '<class 'list'>'
```

Создание классов

```
# Простейший класс
```

```
class Fruit:  
    pass
```



PEP 8

Имена классов по стандарту именования **PEP-8** должны начинаться с большой буквы.

Встроенные классы (**int**, **float**, **str**, **list** и другие) этому правилу не следуют, однако в вашем коде его лучше придерживаться. Так делает большинство программистов на Python.

```
# Создаём экземпляры класса  
# Теперь создадим два конкретных фрукта — экземпляра  
# класса Fruit:
```

```
a = Fruit()  
b = Fruit()
```



```
# Создаём атрибуты
# Переменные a и b содержат ссылки на два разных объекта –
# экземпляра класса Fruit, которые можно наделить разными
# атрибутами:
```

```
a.name = 'apple'
a.weight = 120
# теперь a - это яблоко весом 120 грамм
```

```
b.name = 'orange'
b.weight = 150
# b - это апельсин весом 150 грамм
```

Атрибуты

Атрибуты можно не только устанавливать, но и читать.

При чтении ещё не созданного атрибута будет появляться ошибка **AttributeError**. Вы её часто увидите, допуская неточности в именах атрибутов и методов.

```
# Атрибуты
```

```
print(a.name, a.weight) # apple 120  
print(b.name, b.weight) # orange 150  
b.weight -= 10 # Апельсин долго лежал на складе и усох  
print(b.name, b.weight) # orange 140
```


```
c = Fruit()  
c.name = 'lemon'  
c.color = 'yellow'
```

```
# Атрибут color появился только в объекте c.  
# Забыли добавить свойство weight и обращаемся к нему  
print(c.name, c.weight)  
# Ошибка AttributeError, нет атрибута weight
```

Методы классов

Создаём метод класса

```
class Greeter:  
    def hello_world(self):  
        print("Привет, Мир!")
```



```
greet = Greeter()  
greet.hello_world()
```

Привет, Мир!

```
class Greeter:
    def hello_world(self):
        print("Привет, Мир!")

    def greeting(self, name):
        """Поприветствовать человека с именем name."""
        print("Привет, {}".format(name))

    def start_talking(self, name, weather_is_good):
        """Поприветствовать и начать разговор с вопроса о погоде."""
        print("Привет, {}".format(name))
        if weather_is_good:
            print("Хорошая погода, не так ли?")
        else:
            print("Отвратительная погода, не так ли?")

greet = Greeter()
greet.hello_world()      # Привет, Мир!
greet.greeting("Петя")   # Привет, Петя!

greet.start_talking("Саша", True)
# Привет, Петя!
# Хорошая погода, не так ли?
```

```
class Greeter:
    def hello_world(self):
        print("Привет, Мир!")

    def greeting(self, name):
        """Поприветствовать человека с именем name."""
        print("Привет, {}".format(name))

    def start_talking(self, name, weather_is_good):
        """Поприветствовать и начать разговор с вопроса о погоде."""
        print("Привет, {}".format(name))
        if weather_is_good:
            print("Хорошая погода, не так ли?")
        else:
            print("Отвратительная погода, не так ли?")

greet = Greeter()
greet.hello_world()      # Привет, Мир!
greet.greeting("Петя")   # Привет, Петя!

greet.start_talking("Саша", True)
# Привет, Петя!
# Хорошая погода, не так ли?
```

Инициализация
экземпляров класса


```
# Класс «Машина»
```

```
class Car:
    def start_engine(self):
        engine_on = True # К сожалению, не сработает

    def drive_to(self, city):
        if engine_on: # Ошибка NameError
            print("Едем в город {}".format(city))
        else:
            print("Машина не заведена, никуда не едем")
```

```
c = Car()
c.start_engine()
c.drive_to('Владивосток')
```

```
NameError: name 'engine_on' is not defined
```

Класс «Машина»

```
class Car:
    def start_engine(self):
        self.engine_on = True # К сожалению, не сработает

    def drive_to(self, city):
        if self.engine_on:
            print("Едем в город {}".format(city))
        else:
            print("Машина не заведена, никуда не едем")

c = Car()
c.start_engine()
c.drive_to('Владивосток')
```

Зависит от вызова
start_engine

Едем в город Владивосток.

Специальные методы

Нет ли способа задать значение атрибута **engine_on** по умолчанию?

Да. Есть метод **`__init__`**, который относится к группе так называемых специальных методов, которые имеют особое значение для интерпретатора Python.

Особое значение метода **`__init__`** заключается в том, что если такой метод в классе определён, то интерпретатор автоматически вызывает его при создании каждого экземпляра этого класса для инициализации экземпляра.

Класс «Машина»

```
class Car:
    def __init__(self):
        self.engine_on = False

    def start_engine(self):
        self.engine_on = True

    def drive_to(self, city):
        if self.engine_on:
            print("Едем в город {}".format(city))
        else:
            print("Машина не заведена, никуда не едем")
```

```
car1 = Car()
car1.start_engine()
car1.drive_to('Владивосток')
```

```
car2 = Car()
car2.drive_to('Лиссабон')
```

```
Едем в город Владивосток.
Машина не заведена, никуда не едем
```

Инкапсуляция

Технология сокрытия информации о внутреннем устройстве объекта за внешним интерфейсом из методов называется инкапсуляцией.

Надо стараться делать интерфейс методов достаточно полным. Тогда вы, как и другие программисты, будете пользоваться этими методами, а изменения в атрибутах не будут расползаться по коду, использующему ваш класс. Кроме того, инкапсуляция позволяет шире использовать такое понятие, как полиморфизм.

Соглашения об
именовании, вызов
методов атрибутов

```
class RoboticMailDelivery:
    def __init__(self):
        self.house_flat_pairs = []

    def add_mail(self, house_number, flat_number):
        """Добавить информацию о доставке письма по номеру дома
        house_number, квартира flat_number."""
        self.house_flat_pairs.append((house_number, flat_number))

    def flat_numbers_for_house(self, house_number):
        """Вернуть список квартир в доме house_number,
        в которые нужно доставить письма."""
        flat_numbers = []
        for h, f in self.house_flat_pairs:
            if h == house_number:
                flat_numbers.append(f)
        return flat_numbers
```

Специальные методы

Специальные методы

Остальные специальные методы также вызываются в строго определённых ситуациях.

Так, например, всякий раз, когда интерпретатор встречает запись вида

`x + y,`

он заменяет её на

`x.__add__(y),`

и для реализации сложения нам достаточно определить в классе экземпляра `x` метод **`__add__`**.

Пример

```
class Time:
```

```
    def __init__(self, minutes, seconds):  
        self.minutes = minutes  
        self.seconds = seconds
```

```
    def __add__(self, other):  
        m = self.minutes + other.minutes  
        s = self.seconds + other.seconds  
        m += s // 60  
        s = s % 60  
        return Time(m, s)
```

```
    def info(self):  
        return '{}:{}'.format(self.minutes, self.seconds)
```

```
t1 = Time(5, 50)
print(t1.info()) # 5:50

t2 = Time(3, 20)
print(t2.info()) # 3:20

t3 = t1 + t2
print(t3.info()) # 9:10
```

Важно

Обратите внимание, что в методе `__add__` мы создаём новый экземпляр с результатом сложения, а не изменяем уже существующий.

Для арифметических операторов мы будем поступать так почти всегда, ведь при выполнении `z = x + y` ни `x`, ни `y` изменяться не должны.

Должен создаваться новый объект `z` с результатом операции.

Переопределение
функции `print()`

Метод `__str__`

Перед выводом аргументов на печать функция **print** преобразует их в строки с помощью функции **str**.

Но функция **str** делает это не сама, а вызывает метод **__str__** своего аргумента. Так что вызов **str(x)** эквивалентен **x.__str__()**

Печать объекта

Если мы сейчас попытаемся распечатать экземпляры **Time** просто с помощью **print(t1)**, то получим что-то вроде:

```
<__main__.Time object at 0x7fa021586f98>
```

Это сработала реализация метода **__str__** по умолчанию из класса **object**.

Если мы определим в своём классе собственный метод **__str__**, он заменит тот, что был унаследован от **object**.

```
# Пример
class Time:
    def __init__(self, minutes, seconds):
        self.minutes = minutes
        self.seconds = seconds

    def __add__(self, other):
        m = self.minutes + other.minutes
        s = self.seconds + other.seconds
        m += s // 60
        s = s % 60
        return Time(m, s)

    def __str__(self):
        return '{}:{}'.format(self.minutes, self.seconds)
```



```
t1 = Time(5, 50)
```

```
print(t1) # 5:50
```

```
t2 = Time(3, 20)
```

```
print(t2) # 3:20
```

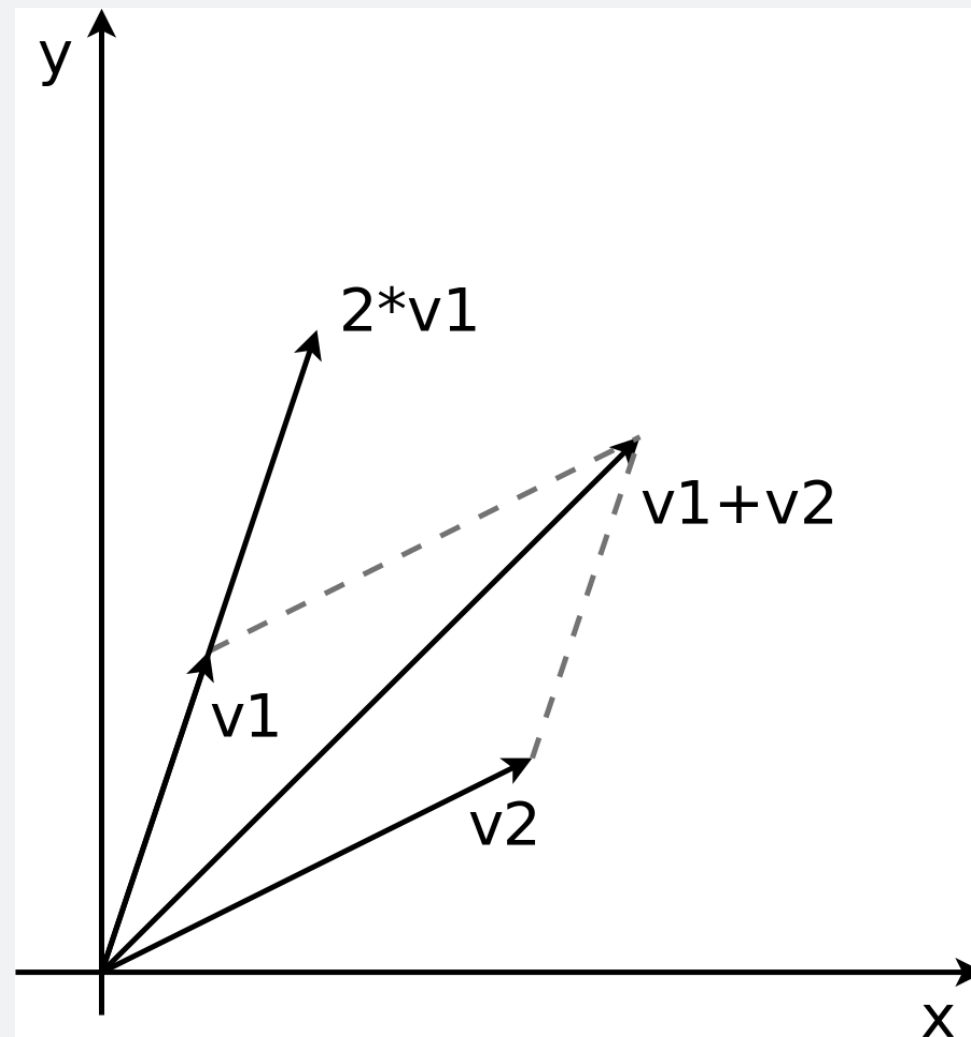
```
t3 = t1 + t2
```

```
print(t3) # 9:10
```

Двумерные векторы

Двумерные векторы — очень полезный и важный геометрический объект.

Векторы любой нужной размерности уже есть в библиотеке NumPy.



```
# Реализуем двумерный вектор
```

```
class MyVector:
```

```
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
    def __add__(self, other):  
        return MyVector(self.x + other.x, self.y + other.y)
```

```
    def __sub__(self, other):  
        return MyVector(self.x - other.x, self.y - other.y)
```

```
    def __mul__(self, other):  
        return MyVector(self.x * other, self.y * other)
```

```
    def __rmul__(self, other):  
        return MyVector(self.x * other, self.y * other)
```

```
    def __str__(self):  
        return 'MyVector({}, {})'.format(self.x, self.y)
```

```
# Реализуем двумерный вектор
```

```
v1 = MyVector(-2, 5)
```

```
v2 = MyVector(3, -4)
```

```
v_sum = v1 + v2
```

```
print(v_sum)    # MyVector(1, 1)
```

```
v_mul = v1 * 1.5
```

```
print(v_mul)    # MyVector(-3.0, 7.5)
```

```
v_rmul = -2 * v1
```

```
print(v_rmul)   # MyVector(4, -10)
```

Другие специальные методы

Печать объекта

Метод	Описание
<code>__add__(self, other)</code>	Сложение ($x + y$). Будет вызвано: <code>x.__add__(y)</code>
<code>__sub__(self, other)</code>	Вычитание ($x - y$)
<code>__mul__(self, other)</code>	Умножение ($x * y$)
<code>__truediv__(self, other)</code>	Деление (x / y)
<code>__floordiv__(self, other)</code>	Целочисленное деление ($x // y$)
<code>__mod__(self, other)</code>	Остаток от деления ($x \% y$)
<code>__divmod__(self, other)</code>	Частное и остаток (<code>divmod(x, y)</code>)
<code>__radd__(self, other)</code>	Сложение ($y + x$). Будет вызвано: <code>y.__radd__(x)</code>
<code>__rsub__(self, other)</code>	Вычитание ($y - x$)
<code>__lt__(self, other)</code>	Сравнение ($x < y$). Будет вызвано: <code>x.__lt__(y)</code>
<code>__eq__(self, other)</code>	Сравнение ($x == y$). Будет вызвано: <code>x.__eq__(y)</code>
<code>__len__(self)</code>	Возвращение длины объекта
<code>__getitem__(self, key)</code>	Доступ по индексу (или ключу)
<code>__call__(self[, args...])</code>	Вызов экземпляра класса как функции

Декораторы

Переопределение функций


```
def main_answer_in_the_universe():  
    return 42
```

```
input = main_answer_in_the_universe  
x = input()  
print(x)
```

42

ИНСТРУКЦИЯ pass.
Согласованность
аргументов

```
def nop(*rest, **kwargs):  
    pass
```

```
print = nop  
print("(шепотом) Потіше, будь ласка!", end='')
```

Инструкция def

```
language = 'fr'  
if language == 'ru':  
    def hello(name):  
        print('Привет, ', name)  
else:  
    def hello(name):  
        print('Hi, ', name)  
  
hello('Joe')
```

Hi, Joe

Важно

Стоит понимать, что **изменять существующие функции очень опасно.**

Почти всегда есть способ обойтись без этого, и этот способ стоит предпочесть. Переопределение функций делает программу плохо предсказуемой.

ФУНКЦИЯ ВНУТРИ ФУНКЦИИ

```
def answer(question):  
    return 'В разработке'  
  
def dialog():  
    def answer(question):  
        if question.lower().startswith('когда'):  
            return 'Никогда!'  
        else:  
            return 'Разоблачили.'  
  
    question = input()  
    while question != '':  
        print(answer(question))  
        question = input()
```

dialog()

```
# <= Когда станет тепло?  
# => Никогда!  
# <= Когда смогу найти богатство?  
# => Никогда!  
# <= Какие в Чили существуют города?  
# => Разоблачили
```


Декораторы

```
old_print = print
```

```
def print_upper_case(*args):  
    args_upcased = [str(arg).upper() for arg in args]  
    old_print(*args_upcased)
```

```
print = print_upper_case  
print('Нельзя ли потише?')
```

НЕЛЬЗЯ ЛИ ПОТИШЕ?

```
def use_uppercased_arguments(old_func):  
    def new_func(*args, **kwargs):  
        args_upcased = [str(arg).upper() for arg in args]  
        old_func(*args_upcased, **kwargs)  
  
    return new_func
```

```
print = use_uppercased_arguments(print)  
print('Нельзя ли потише?')
```

НЕЛЬЗЯ ЛИ ПОТИШЕ?

Декораторы

Декораторами такие функции как **`use_uppercased_arguments()`** называются потому, что обычно они добавляют какие-то штрихи (декор) к уже существующему поведению, не изменяя её код.

Примеры использования

- Когда каждый вызов функции нужно залоггировать, т.е. вывести при вызове сообщение о том, как и когда функция была вызвана.
- Когда результат функции должен быть закэширован (т.е. после вычисления сохранен на будущее, чтобы не считать его повторно).
- Чтобы функция использовалась для ответа на запросы к веб-серверу.
- Чтобы перед запуском проверялось какое-то условие (например, что пользователь имеет право доступа к выполнению функции).

```
def logged(func):  
    count = 0  
  
    def decorated_func(*args, **kwargs):  
        nonlocal count  
        count += 1  
        print(count, '>>', 'Arguments:', args,  
              'Named arguments:', kwargs)  
        result = func(*args, **kwargs)  
        print('--', 'Result:', result)  
        return result  
  
    return decorated_func
```

```
@logged  
def make_burger(typeOfMeat, withOnion=False, withTomato=True):  
    print('Булочка')  
    if withOnion:  
        print('Луковые колечки')  
    if withTomato:  
        print('Ломтик помидора')  
    print('Котлета из', typeOfMeat)  
    print('Булочка')
```

```
make_burger('Курица')  
make_burger('Рыба')
```

```
1 >> Arguments: ('Курица',) Named arguments: {}
```

```
Булочка
```

```
Ломтик помидора
```

```
Котлета из Курица
```

```
Булочка
```

```
-- Result: None
```

```
2 >> Arguments: ('Рыба',) Named arguments: {}
```

```
Булочка
```

```
Ломтик помидора
```

```
Котлета из Рыба
```

```
Булочка
```

```
-- Result: None
```

```
def logged(func):
    count = 0

    def decorated_func(*args, **kwargs):
        nonlocal count
        count += 1
        current_index = count
        print(current_index, '>>', 'Arguments:', args)
        result = func(*args, **kwargs)
        print(current_index, '--', 'Result:', result)
        return result

    return decorated_func

@logged
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

fib(5)
```

```
1 >> Arguments: (5,)
2 >> Arguments: (4,)
3 >> Arguments: (3,)
4 >> Arguments: (2,)
4 -- Result: 1
5 >> Arguments: (1,)
5 -- Result: 1
3 -- Result: 2
6 >> Arguments: (2,)
6 -- Result: 1
2 -- Result: 3
7 >> Arguments: (3,)
8 >> Arguments: (2,)
8 -- Result: 1
9 >> Arguments: (1,)
9 -- Result: 1
7 -- Result: 2
1 -- Result: 5
```



```
# Захват значения из аргумента
```

```
def power(degree):  
    def func(x):  
        return x ** degree  
  
    return func
```

```
square = power(2)  
cube = power(3)  
  
print(square(5))
```

```
25
```

```
# Захват значения из аргумента
```

```
def invitation_sender(city, date, text):  
    def sender(email, name):  
        return send_invitation(email, name, text, date, city)  
  
    return sender
```

```
sendmail = invitation_sender('Москва', '1 апреля 2022 г',  
                             'Приглашаем вас на встречу')  
sendmail('vasiliy-petrov@mail.ru', 'Василий Петров')  
sendmail('petr-alekseev@mail.ru', 'Петр Алексеев')  
sendmail('vasiliy-vasilyev@mail.ru', 'Василий Васильев')
```

TO BE CONTINUED...

PART 2: Computer Science – Машинное обучение

