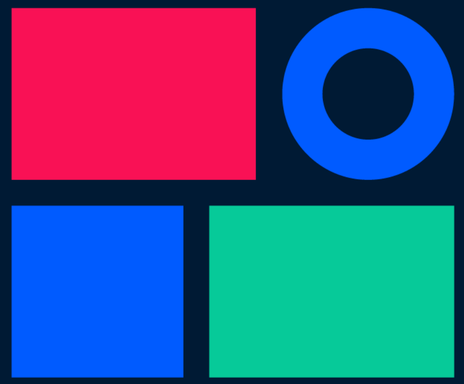


Ozon
New Skills

Ozon
New Skills

# Python 18+

## Классы/ Теория и практика

# ООП

ООП — одна из ведущих парадигм программирования в современном мире. Она чаще востребована в больших компаниях, которые стремятся сэкономить огромное количество человеко-часов за счет постоянного реиспользования классов.

# ООП

Объектно-ориентированная методология(ООП) — методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы могут образовывать иерархию наследования.

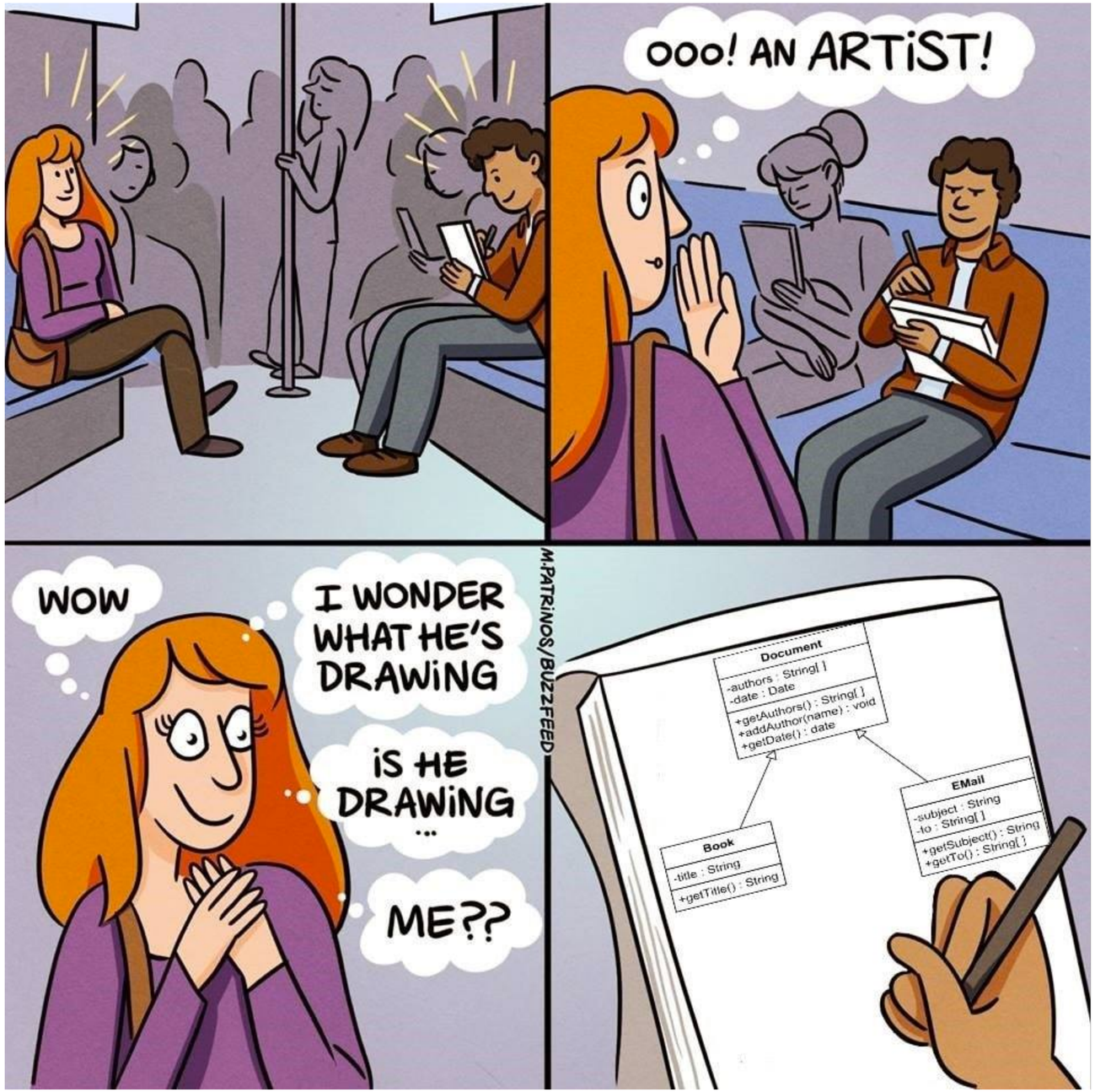
# ООП

На самом деле каждый язык программирования предоставляет свои возможности к реализации ООП парадигмы. В JS и прочих скриптовых языках их возможности весьма ограничены, в языках C и Go класса как понятия нет в принципе.

# ООП

Объектно-ориентированное программирование берет концепции из реальной жизни. Вы легко можете перенести в ООП любую реальную вещь – например, автомобиль.

# Картинка для привлечения внимания



# Реализация ООП

Непосредственно в Python

# ООП

Объявление класса начинается с ключевого слова class, после которого следует указанное программистом имя класса (при выборе имени придерживаются обычных правил наименований переменных и функций, но имя класса традиционно начинается с Заглавной буквы):

# Вариант создания класса

```
class ClassName:
    ''' строка документации '''
    # объявление переменных класса
    # объявление методов класса
```

# ООП

Объявление класса, в котором определяются его атрибуты и методы, является образцом, из которого могут быть произведены рабочие копии( экземпляры) класса.

Все переменные, которые объявлены внутри определения методов, известны как переменные экземпляра и доступны только локально в том методе, в котором они были объявлены — к ним нельзя обратиться извне структуры класса.

# ООП

Как правило, переменные экземпляра содержат данные, которые передаются вызывающим оператором во время создания экземпляра класса

# ООП

Поскольку эти данные доступны только локально (для внутреннего пользования) они фактически скрыты от остальной части программы. Такой прием называется инкапсуляцией данных, и он гарантирует, что данные надежно спрятаны только внутри класса (первый столп ООП)

# Простой класс

```
class SimpleClass:
      '''самый простой класс на свете'''
      def __init__(self, x, y):
            self.x = x
            self.y = y
      def sum_num(self):
            return self.x + self.y

s = SimpleClass(1, 3)
print(s.sum_num())
```

# ООП

Ко всем свойствам класса можно обратиться локально, используя точечную запись с префиксом self, например, атрибут с именем name можно записать следующим образом self.name. Кроме того, все определения методов должны содержать self в качестве первого аргумента.

# ООП

Так же при создании экземпляра класса автоматически вызывается специальный метод __init__(self). Если необходимо передать еще значения для инициализации его атрибутов, то в скобки могут быть добавлены необходимые аргументы.

Давайте попробуем создать класс, к примеру, автомобиля:

# Класс с конструктором

```
class Car:
        """Базовый класс автомобиля"""
        def __init__(self, marka, speed):
        """ инициализирует атрибуты марки и скорости автомобиля"""
                self.marka = marka
                self.speed = speed
        def car_ride(self):
                return "машинка " + self.marka + " едет со скоростью ” +
        str(self.speed)
bmw = Car("bmw", 90)
print(bmw.car_ride())
```

# Экземпляры и атрибуты

После того, как вы создали свой экземпляр, вы можете

обращаться к нему через «точечную» нотацию. К примеру, можно узнать свойства нашего экземпляра:

```
print(bmw.marka)
print(bmw.speed)
```

# удаление свойства

чтобы удалить свойство, можно воспользоваться следующим оператором del:

```
del bmw.speed
```

# Альтернативный синтаксис

| Команды | Их назначение |
| --- | --- |
| `getattr(bmw, 'speed')` | возвращает значение атрибута экземпляра класса. |
| `hasattr(bmw, 'speed')` | возвращает логическое True, если у объекта есть это свойство |
| `setattr(bmw, 'speed', 180)` | модифицирует текущее значение, либо создает новый атрибут для экземпляра |
| `delattr(bmw, 'speed')` | удаляет атрибут из экземпляра |

# Встроенные атрибуты

## Встроенные атрибуты

В языке Python каждый класс автоматически создается с определенным набором встроенных «частных» атрибутов. Доступ к их значениям можно получить, использую точечную запись. Например, чтобы получить значение атрибута строки документации определенного класса, вам нужно записать

**`имя_класса.__doc__`**

# Встроенные атрибуты

Встроенный атрибут **`__dict__`** является словарем, который содержит пары ключей и связанных с ними значений. Ключами здесь являются имена атрибутов, а значениями — соответствующие значения атрибута.

# Перечисление свойств класса

```
for attr in dir(bmw): if attr[0] == '_':
     print(attr)
for item in bmw.__dict__:
     print(item, bmw.__dict__[item])
```

# Импорт класса

Достаточно популярна стратегия размещения отдельного класса в отдельном файле. Если он вам нужен - вы его импортируете. Если не нужен — можете не импортировать. Зато если сложность программы возрастет, и экземпляров и классов станет очень много, вы легко сможете с ними разобраться, потому каждый класс будет в отдельном файле (название класса желательно должно совпадать с названием файла).

# Изменение атрибута

Значение атрибута можно изменить одним из трех способов: изменить его в экземпляре, задать значение при помощи метода или изменить его с приращением (то есть прибавлением определенной величины) при помощи метода.

# Назначение свойства через метод

```
class Car:
...
    def read_odometr(self):
        """Выведет нам пробег автомобиля """
        print("У этого автомобиля пробег " +
        str(self.odometr_reading) + " на счетчике")
    def update_odometr(self, mileage):
        self.odometr_reading = mileage
    # теперь мы сможем обновлять
    # значения одометра с помощью вызова метода
bmw = Car('bmw', 90)
bmw.update_odometr(290)
bmw.read_odometr()
```

# Наследование

Работа на новый классом не обязана начинаться с нуля. Если класс, который вы пишите, представляет собой специализированную версию ранее написанного класса, вы можете воспользоваться наследованием.

# Наследование

Один класс, наследующий от другого, автоматически получает все атрибуты и методы первого класса. Исходный класс тогда называется родителем, а новый класс — потомком. Класс - потомок наследует атрибуты и методы родителя, но при этом может определять свои собственные атрибуты и методы.

# metod __init__

Первое, что делает Python при создании экземпляра класса- потомка — присваивает значения всем атрибутам класса-родителя. Для этого методу __init__ класса потомка необходима помощь со стороны родителя.

# Наследование

```
class ElectricCar(Car):
        """ Представляет класс автомобиля с электродвигателем"""
        def __init__(self, marka, speed):
        """ Инициализирует атрибуты класса - родителя"""
                super().__init__(marka, speed)
myTesla = ElectricCar('tesla', 300)
print(myTesla.car_ride())
```

# metod super()

Функция super() — специальная функция, которая помогает Python связать потомка с родителем. Эта строка приказывает Python вызвать метод **__init__.** класса, который является родителем ElectricCar, в результате чего экземпляр ElectricCar получает все атрибуты класса родителя. Имя super происходит из распространённой терминологии: класс родитель называется суперклассом, а класс-потомок — субклассом.

# Экземпляры как атрибуты

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся слишком длинными и громоздкими. В такой ситуации часть одного класса нередко стоит записать в виде отдельного класса. Один большой класс разбивается на несколько небольших, которые куда удобнее тестировать

# Экземпляры как атрибуты

Например, при дальнейшей доработке класса ElecticalCar может оказаться, что в нем появилось слишком много новых атрибутов и методов, которые относятся только к аккумулятору. В таком случае куда логичнее создать отдельный класс Battery

# Статичные методы

Иногда вы не хотите создавать экземпляр класса, а хотите воспользоваться каким-то методом класса как шаблоном. Для этого вам может понадобиться статичные методы **`@staticmethod:`**

# Статичные методы

```
@staticmethod
def get_class_details():
     print("Это класс Car")
Car.get_class_details()
```

# Оформление класса

Имена классов должны записываться в «верблюжной» схеме.

Имена экземпляров и модулей записываются в нижнем регистре с разделением слов символами подчеркивания.

# Оформление класса

Каждый класс должен иметь строку документации, которая следует сразу за определением класса. Строка документации должна содержать краткое описание того, что делает класс, и в ней должны соблюдаться те соглашения по форматированию, которые вы использовали для написания строк документации в функциях.

# Модификаторы доступа

В таком виде, в котором она есть в Python

# Ограничение доступа

По умолчанию все свойства классов открыты для доступа извне, благодаря чему их можно в любой момент изменить по своему усмотрению при помощи оператора точки. Это не всегда хорошо, так как существуют некие риски потери информации либо введения неправильных данных, приводящих к сбоям в работе программы.

# Ограничение доступа

В такой ситуации помогает еще одна особенность ООП под названием инкапсуляция. Она предписывает применение приватных свойств класса, к которым отсутствует доступ за его пределами.

# Ограничение доступа

Чтобы ограничить видимость полей, следует задать для них имя, начинающееся с двойного подчеркивания.

# Приватные методы

```
class Cat:
      __name = "Kitty”
      def get_name(self):
            return self.__name
      def set_name(self, name):
            self.__name = name
cat = Cat()
print(cat.get_name())
cat.set_name("Misty")
print(cat.get_name())
```

```
Kitty
Misty
```