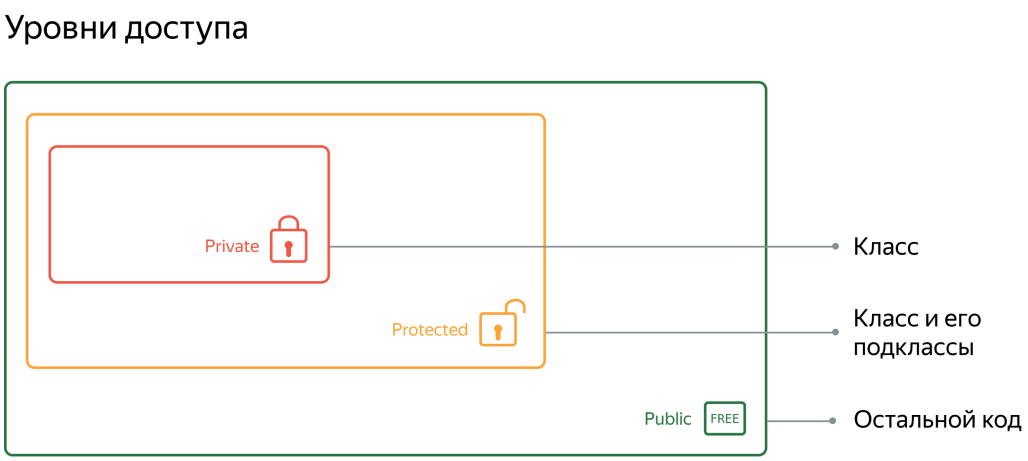
# Шпаргалка: инкапсуляция

📌 Модификаторы доступа — это такие ограничители. Они контролируют доступ к атрибутам и методам класса: говорят, из какой части программы к ним можно обратиться, а из какой — нельзя.

В Python три модификатора доступа:

- **публичный** (public) — атрибуты и методы доступны отовсюду;
- **защищённый** (protected) — доступны в классе и его подклассах;
- **приватный** (private) — доступны только внутри класса.




В Python невозможно ограничить доступ на 100%. С любыми атрибутами и методами можно работать, как с публичными.

📌 Модификаторы доступа пишут для пользователей кода — разработчиков, которые будут с ним работать. Это такое соглашение: программисты договорились, что переменные и методы с модификаторами трогать не нужно.

### Модификатор public

По умолчанию все атрибуты и методы классов — публичные. Специального обозначения нет.

К публичным переменным можно обратиться **из любой части программы**.

**Пример.** Есть класс `Robot`. У него два атрибута:

- `name` — имя робота;
- `energy_level` — заряд батареи.

Их использует метод `say_hello`. Он проверяет заряд батареи робота и печатает сообщение.

```
class Robot:
    def __init__(self, name, energy_level):
        self.name = name
        self.energy_level = energy_level

    def say_hello(self):
        if self.energy_level > 0:            # проверяем уровень заряда робота
            self.energy_level -= 10          # если он больше нуля, вычитаем десять единиц
            return f"Привет, я {self.name}!" # и выводим приветствие
        else:
            return "Энергия закончилась :("  # если меньше нуля - выводим предупреждение
```

Тут все атрибуты публичные. Их можно прочитать и изменить вне класса. Это полезно, если нужно управлять значениями атрибутов в объектах.

**Пример.** Изменили уровень энергии робота и переименовали его:

```
# создали объект и установили значения
robot = Robot("R2-D2", 100)
print(robot.say_hello())      # Привет, я R2-D2!

# выставили другой уровень энергии, чтобы вывод изменился
robot.energy_level = 0
print(robot.say_hello())      # Энергия закончилась :(

# получили имя робота
print(robot.name)             # R2-D2

# изменили имя
robot.name = "C-3PO"
print(robot.name)             # C-3PO
```

## Модификатор protected

Защищённые атрибуты и методы доступны **внутри класса и его подклассов**. Их можно наследовать и переопределять.

Модификатор protected обозначают одним подчеркиванием перед именем — `_`. Вот так:

```
class Example:
     def __init__(self, attribute):
     self._attribute = attribute     # одно подчёркивание - это модификатор protected
                                     # к attribute можно обращаться только в классе или
                                     # в подклассах
```

**Пример.** Есть два класса: суперкласс «машина» — `Car` и подкласс «спортивная машина» — `SportsCar`:

```
class Car:
    def __init__(self, speed_limit):
        self._speed_limit = speed_limit    # защищённый атрибут суперкласса

    def drive(self, speed):
        if speed <= self._speed_limit:
            print(f"Скорость - {speed} км/ч")
        else:
            self._speed_warning()

    def _speed_warning(self):              # защищённый метод суперкласса
        print("Слишком высокая скорость!")


class SportsCar(Car):
    def __init__(self, speed_limit, turbo_speed):
        super().__init__(speed_limit)      # подкласс унаследовал защищённый атрибут
        self._turbo_speed = turbo_speed    # и у него есть свой такой же

    def turbo_drive(self, speed):
        if speed <= self._turbo_speed:
            print(f"Скорость - {speed} км/ч в турбо-режиме")
        else:
```

```
            self._speed_warning()          # защищённый метод тоже унаследован


my_car = SportsCar(200, 300)
my_car.turbo_drive(250)  # Скорость - 250 км/ч в турбо-режиме
my_car.turbo_drive(350)  # Слишком высокая скорость!
```

Подкласс `SportsCar` унаследовал все атрибуты и методы суперкласса `Car`. Ещё у него есть собственный защищённый атрибут — `_turbo_speed`.

### Обращение к защищённым атрибутам и методам

📌 Интерпретатор не выдаст ошибку, если обратиться к защищённому атрибуту вне класса. Но делать так не стоит.

**Пример.** Можно вызвать метод `_speed_warning` без вызова `drive`:

```
my_car = SportsCar(200, 300)
my_car._speed_warning()     # Слишком высокая скорость!
```

**Ещё пример.** Можно напрямую обратиться к `_speed_limit` и `_turbo_speed`:

```
print(my_car._speed_limit)  # Выведет: 200
print(my_car._turbo_speed)  # Выведет: 300
```

❌ **Так делать не нужно: это пример плохого кода.**

Если обратишься к защищённым переменным класса в объекте, то нарушишь соглашение. Их скрыли специально — значит, прямое обращение не предусмотрено. Оно может сломать программу.

### Модификатор private

Приватный модификатор похож на защищённый. Его обозначают двумя подчёркиваниями. Например, `__attribute`.

С ним атрибуты и методы доступны только **внутри класса**. Их нельзя вызывать в объектах и наследовать.

**Пример.** Можно переписать класс `BankAccount`:

```
class BankAccount:
    def __init__(self):
        self.__balance = 0

    def deposit(self, amount):            # пополнить счёт
        if amount > 0:
            self.__balance += amount
            self.__display_balance()
        else:
            self.__invalid_operation()

    def withdraw(self, amount):           # снять деньги со счёта
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            self.__display_balance()
        else:
            self.__invalid_operation()

    def __invalid_operation(self):       # вывести сообщение об ошибке
        print("Некорректная операция")

    def __display_balance(self):         # вывести, сколько денег на счёте
        print(f"Текущий баланс: {self.__balance}")
```

```
# создали объект
my_account = BankAccount()
my_account.deposit(200)   # Текущий баланс: 200
my_account.withdraw(100)  # Текущий баланс: 100
```

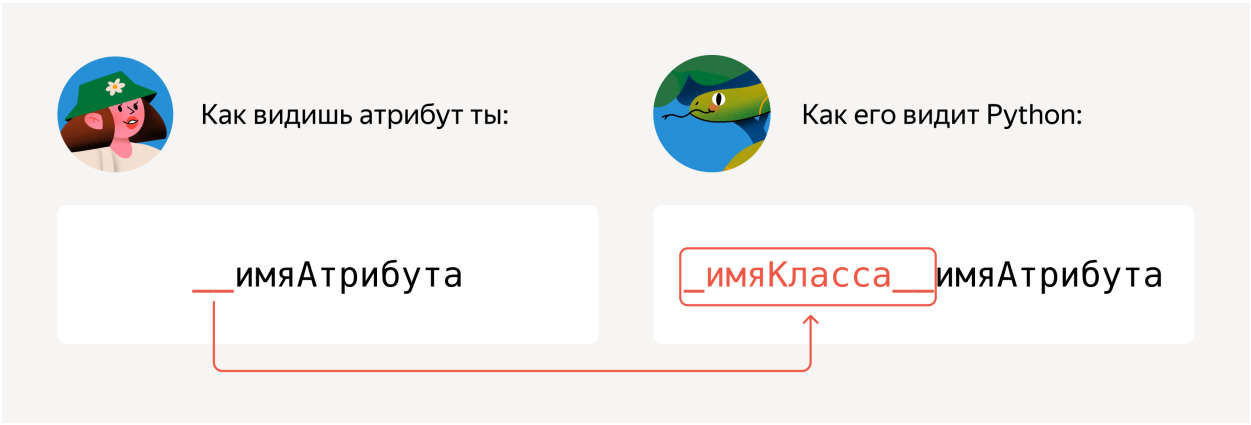
Атрибут `__balance` приватный. Значит, пользователь не сможет поменять его значение в обход методов `deposit` и `withdraw`. Теперь нельзя вывести баланс в минус. Или добавить к сумме пару нулей, не пополняя счёт.

Методы `__invalid_operation` и `__display_balance` — служебные. Они нужны для работы `deposit` и `withdraw`. Пользователю ни к чему вызывать их напрямую.

### Искажение имени: name mangling

📌 Ограничения доступа private строже, чем protected.

Если перед атрибутом стоят два подчёркивания, Python искажает его имя. Делает он это незаметно: у тебя в IDE имя не изменится, а для себя интерпретатор добавит к нему `_ИмяКласса`.




Это называется **name mangling** — искажение имени. Обращение к атрибуту с двумя подчёркиваниями выведет ошибку `AttributeError`.

**Пример**:

```
class BankAccount:
    def __init__(self):
        self.__balance = 0    # balance - приватный атрибут

my_account = BankAccount()
print(my_account.__balance)   # попробовали его напечатать

# получили ошибку:
# AttributeError: 'BankAccount' object has no attribute '__balance'
```

Python знает атрибут `__balance` как `_BankAccount__balance`. Имя `__balance` ему ни о чём не говорит. Поэтому программа пишет: «У объекта нет такого атрибута».

Кажется, что name mangling — серьёзная защита. Но и её можно обойти. Достаточно обратиться к атрибуту по новому имени.

```
class BankAccount:
    def __init__(self):
        self.__balance = 0
```

```
my_account = BankAccount()
print(my_account._BankAccount__balance)  # обратились к атрибуту так, как его зовёт интерпретатор
```

📌 **Не применяй это на практике:** к приватным переменным не нужно обращаться напрямую. Это нарушает соглашение. Просто знай, что это возможно.

### Модификатор private и наследование

Если вызвать приватный метод или использовать приватный атрибут в классе-наследнике, тоже вылезет ошибка.

**Пример.** Класс `PremiumAccount` — это подкласс `BankAccount`. В нём есть метод `purchase`. Он использует приватные элементы суперкласса:

```
class PremiumAccount(BankAccount):      # PremiumAccount - подкласс BankAccount
    def purchase(self, cost):
        if cost > 0 and cost <= self.__balance:
            self.__balance -= cost      # приватный атрибут суперкласса
            print("Purchase successful")
            self.__display_balance()    # приватные методы суперкласса
        else:
            self.__invalid_operation()


my_account = PremiumAccount()
my_account.purchase(100)  # попытались совершить покупку

# получили ошибку:
# AttributeError: 'PremiumAccount' object has no attribute '_PremiumAccount__balance'
```

Метод `purchase` не работает. Всё потому, что у `PremiumAccount` нет доступа к приватным атрибутам и методам суперкласса.

## Как выбрать модификатор

✅ **Public =** доступ из любой части кода.

Оставляй публичный доступ, если хочешь, чтобы с объектами класса могли работать другие части программы.

**Атрибуты.** Используй публичные атрибуты, когда нужно дать прямой доступ к данным класса извне. Например, как атрибут `name` в классе `Robot`.

**Методы.** Метод должен быть публичным, если он представляет основной функционал класса. Например, как методы `deposit` и `withdraw` в классе `BankAccount`.

✅ **Protected** = доступ внутри класса и его подклассах.

Этот модификатор используют, когда пишут суперкласс.

**Атрибуты.** Делай атрибуты защищёнными, если они должны быть доступны только для класса и его подклассов. Например, как `_speed_limit` в классе `Car`.

**Методы.** Делай защищёнными служебные методы, которые используешь в других методах суперкласса и подклассов. Например, как `_speed_warning`.

✅ **Private** = доступ только внутри класса.

**Атрибуты.** Приватными делают атрибуты, которые важны для работы класса. Например, их значение постоянно или меняется по определённой логике. Если пользователь сможет свободно менять их значение, он сломает программу. Например, как `balance` в `BankAccount`.

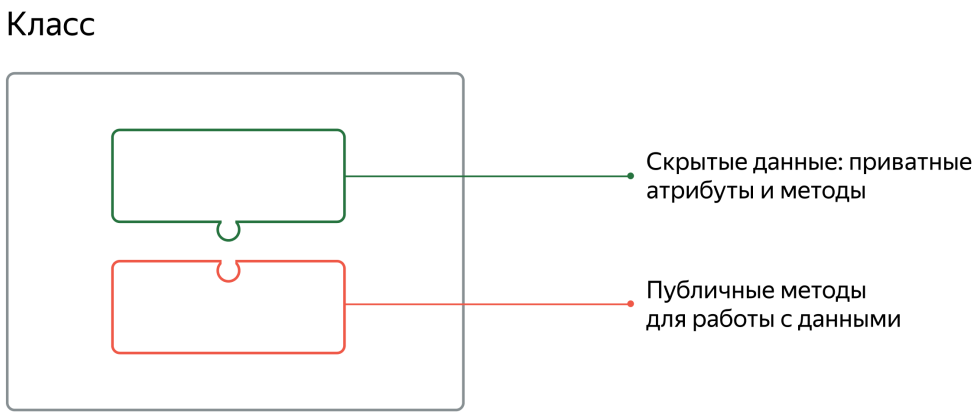
**Методы.** Если используешь методы только внутри класса, делай их приватными. Например, как `__display_balance()` и `__invalid_operation()`. Они нужны только для работы `deposit` и `withdraw`.

## Инкапсуляция

В Python модификаторы доступа используют, чтобы писать код с **инкапсуляцией**.

> 📌 Инкапсуляция — это третий принцип ООП. В ней два аспекта: сокрытие данных и объединение данных с методами.

Идея в том, чтобы контролировать доступ к данным класса. Скрыть их недостаточно: они всё ещё могут понадобиться пользователю. Нужно предложить альтернативу — написать методы, которые смогут получать и менять значения приватных атрибутов.




Так пользователю не придётся разбираться в коде класса и что-то напрямую менять в объекте. Вместо этого он будет пользоваться готовыми удобными методами.

### Класс без инкапсуляции

**Пример.** Есть класс «автомобиля» — `Car`. Он написан без инкапсуляции: атрибуты не скрыты, методов для работы с ними нет:

```
class Car:
    def __init__(self):
        self.fuel = 0           # уровень топлива
        self.speed = 0          # скорость

# создали объект класса
my_car = Car()
my_car.fuel = -50   # Недопустимая операция, но код не выдаст ошибку
my_car.speed = 150  # Опасно, но снова ничто не остановит нас
```

Уровень топлива и скорость машины можно менять напрямую — значения ничем не ограничены. Значит, пользователь может ввести всё, что угодно. Это противоречит логике программы и может её сломать.

Класс `Car` можно переписать с инкапсуляцией в два шага.

### Шаг 1. Сделать атрибуты приватными

Первым делом нужно ограничить доступ к атрибутам — `fuel` и `speed`:

```
class Car:
    def __init__(self):
        self.__fuel = 0            # сделали атрибуты приватными
        self.__speed = 0
```

Атрибуты доступны только внутри класса. Напрямую обратиться к ним в объектах не получится.

### Шаг 2. Написать методы

Теперь нужно методы, которые будут контролировать доступ к атрибутам. Через них пользователь сможет менять значения: заливать топливо и регулировать скорость машины.

```
class Car:
    def __init__(self):
        self.__fuel = 0            # сделали атрибуты приватными
        self.__speed = 0

    def add_fuel(self, amount):    # добавили метод, который заливает топливо
        if amount > 0:
            self.__fuel += amount

    def set_speed(self, speed):    # добавили метод, который устанавливает скорость
        if 0 <= speed <= 120:
            self.__speed = speed
        else:
            print("Скорость должна быть в пределах от 0 до 120.")
```

Обрати внимание на условия в методах:

- `add_fuel` проверяет, что в бак заливают положительное значение топлива;
- `set_speed` следит, чтобы скорость машины не превышала 120 км/ч и не опускалась ниже нуля.

> 👉 Условия не позволят пользователю выставить неправильные значения. Используй их, если хочешь ограничить значения атрибутов.

Теперь работать с данными можно только через методы:

```
# создали объект
my_car = Car()

# залили бензин
my_car.add_fuel(50)

# разогнали машину
my_car.set_speed(90)
my_car.set_speed(180)              # Скорость должна быть в пределах от 0 до 120.
```

### Чем полезна инкапсуляция

Инкапсуляция помогает:

- **Упростить код.** Пользователи получают понятные методы для работы с данными.
- **Повысить безопасность кода.** Данные защищены от неправильного использования. Если с атрибутом нельзя что-то делать, метод этого делать не будет.

- **Повысить гибкость и расширяемость кода.** Класс можно дорабатывать: дописывать новые методы и редактировать старые, добавлять атрибуты. При этом код, который использует класс, менять не придётся. Там будет только вызов методов.

### Инкапсуляция в автотестах

Инкапсуляция пригодится тебе в автотестах для веб-приложений. Она поможет упростить и структурировать код.

В автотестах будет много действий: нужно открыть страницу, войти в аккаунт, заполнить форму, прокрутить страницу... Каждое действие — это крупный кусочек кода с кучей деталей.

Если описывать в каждое действие, программа будет длинной и сложной:

```
# код автотеста

# найти поле для ввода логина и ввести логин
...
# найти поле для ввода пароля и ввести пароль
...
# нажать кнопку входа
...
# открыть форму
...
# заполнить первое поле формы
...
# заполнить второе поле формы
...
# заполнить третье поле формы
...
# нажать кнопку «Отправить форму»
...
# перейти на другую страницу
...
```

Можно создать класс `WebPage` и вынести код действий туда:

```
# класс с методами для работы с веб-страницей
class WebPage:
    ...

    def login(self, username, password):
        ...  # Авторизуемся: вводим логин и пароль, нажимаем кнопку входа...

    def fill_form(self, form_data):
        ...  # Заполняем форму...

    def navigate_to_page(self, page_url):
        ...  # Переходим на страницу...


# а вот код автотеста
# здесь всё красиво и аккуратно
page = WebPage()
page.login('user', 'pass')  # авторизуемся
page.fill_form(...)         # заполняем форму
page.navigate_to_page(...)  # переходим на новую страницу
```

Действия со страницей тут инкапсулированы в методы класса `WebPage`. Это значит, что детали их реализации скрыты внутри класса. В автотесте не нужно писать код действий — только вызвать готовые методы.

Эта программа написана по правилам Page Object Model. Это подход к написанию автотестов, в основе которого — инкапсуляция.

### Геттеры и сеттеры

Инкапсуляция предполагает, что к приватным атрибутам класса нельзя обращаться напрямую. Для работы с ними пишут специальные методы, которые получают и меняют их значения.

У этих методов есть названия. Их называют геттеры и сеттеры.


### Геттеры

📌 Геттер — это метод, который возвращает значение приватного атрибута.

Название происходит от слова get — получить.

Чтобы его написать, нужно:

1. Создать в классе метод;
2. В теле метода написать `return` и имя атрибута вместе с `self`;

Вот так:

```
    def get_attribute(self):
        return self.__attribute
```

Геттеры называют по одной схеме: `get_` + `имя_атрибута`. **Пример:** `get_balance` — это метод, который получает значение баланса.

**Ещё пример.** Есть приложение, которое управляет роботом-пылесосом. Нужно следить за его состоянием: работает он, заряжается или находится в режиме ожидания.

Робота описывает класс `RobotVacuum`. Так он выглядит с геттером:

```
class RobotVacuum:
    def __init__(self, state):
        self.__state = state      # атрибут состояния

    # геттер, который получает состояния робота-пылесоса
    def get_state(self):
        return self.__state
```

### Сеттеры

📌 Сеттер — это метод, который устанавливает значение приватного атрибута.

Название происходит от слова set — установить.

Чтобы его написать, нужно:

1. Создать в классе метод;

2. В теле метода ещё раз присвоить атрибуту новое значение.

Вот так:

```
def set_attribute(self, attribute):
    self.__attribute = attribute
```

Сеттеры называют так же, как геттеры. Только к имени атрибута добавляют `set_`, а не `get_`. **Пример:** метод `set_balance` устанавливает значение баланса.

Часто значение атрибута нужно ограничить. **Пример.** Атрибут отвечает за имя пользователя. Его значением может быть только строка.

> 📌 Чтобы ограничить неподходящие значения, в сеттере пишут условие.

**Пример.** У робота-пылесоса три состояния: `"работает"`, `"заряжается"`, `"в ожидании"`. Атрибут `state` может принимать одно из этих значений.

Сеттер должен сравнить предлагаемое значение и допустимое. Для этого в условии нужно использовать оператор `in` и список подходящих состояний. Вот так:

```
class RobotVacuum:
    def __init__(self, state):
        self.__state = state

    # геттер получает состояние робота-пылесоса
    def get_state(self):
        return self.__state

    # сеттер устанавливает состояние робота-пылесоса
    def set_state(self, state):
        if state in ["работает", "заряжается", "в ожидании"]:
            self.__state = state
        else:
            print(f"Недопустимое состояние: {state}. Состояние может быть 'работает', 'заряжается', или 'в ожидании'")
```

Вот и всё: класс `RobotVacuum` написан по всем правилам инкапсуляции. Теперь можно создать объект и проверить работу методов:

```
# создали объекта
robo_vacuum = RobotVacuum("в ожидании")

# вызвали геттер, чтобы получить состояние робота-пылесоса
print(robo_vacuum.get_state())  # Вывод: в ожидании

# вызвали сеттер, чтобы поменять состояние робота-пылесоса
robo_vacuum.set_state("работает")
print(robo_vacuum.get_state())  # Вывод: работает

# попытались использовать сеттер, чтобы установить недопустимое состояние
robo_vacuum.set_state("летает")  # Вывод: Недопустимое состояние: летает. Состояние может быть 'работает', 'заряжается', или 'в ожидании'
print(robo_vacuum.get_state())   # Вывод: работает
```

## Свойства

Работу с сеттерами и геттерами можно упростить. С этим помогают **свойства**.

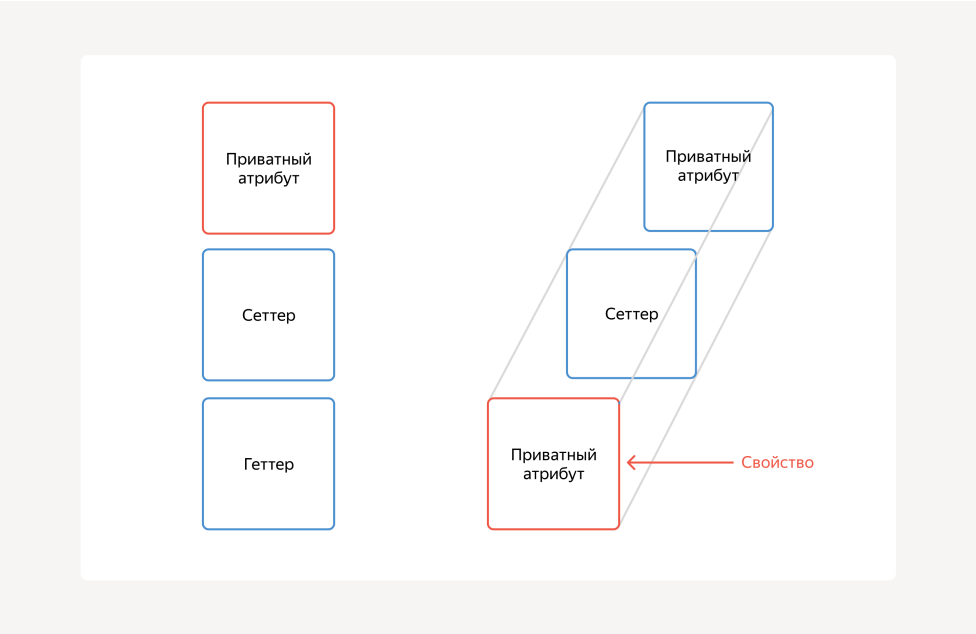
> 📌 Свойства (property) — это особый вид атрибутов. При чтении они вызывают геттер, а при присвоении значения — сеттер.

Проще говоря, вызов методов работает через обращение к атрибуту.

```
# пишем так                              # а работает так
robo_vacuum.state        --->            robo_vacuum.get_state()

robo_vacuum.state = 'заряжается'   --->  robo_vacuum.set_state('заряжается')
```

Геттеры и сеттеры словно прячутся за атрибутом. Они во всём его пародируют: откликаются на его имя и ведут себя не как методы. Но работу свою делать продолжают: если в них есть условия, они сработают. Программа не даст присвоить атрибуту неправильное значение.




## Как создать свойство

Чтобы сделать атрибут свойством, нужно навесить на его геттер декоратор `@property`.

📌 Декоратор `@property` называют декоратором свойств. Он заставляет метод работать как атрибут.

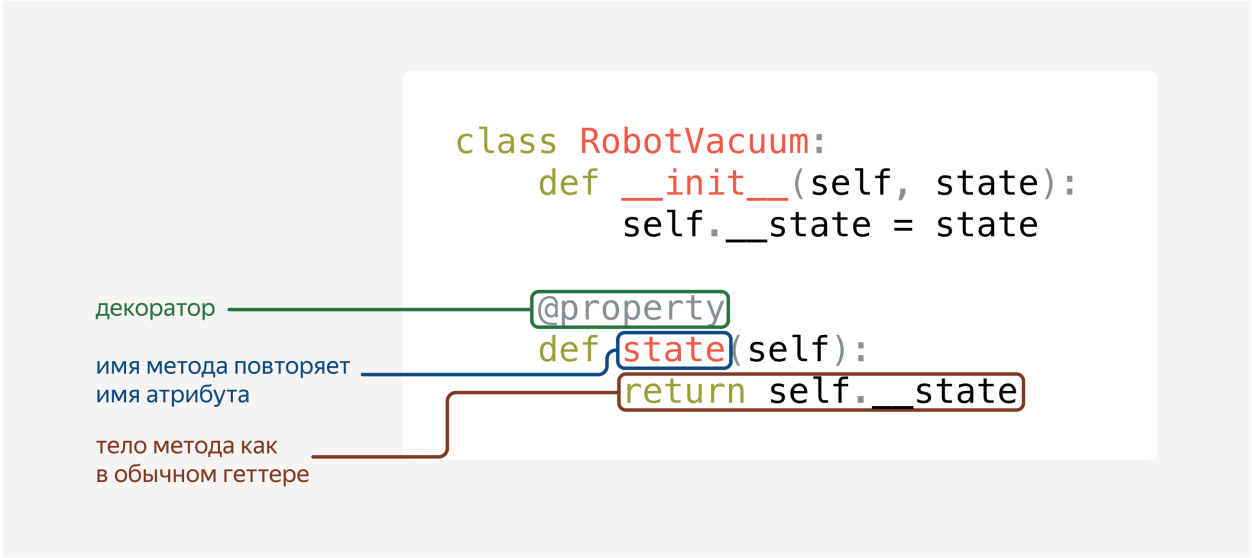
Выбери атрибут, который хочешь сделать свойством. Затем:

1. Создай метод с тем же именем, что и у атрибута;
2. Напиши над ним декоратор `@property`. Метод станет геттером свойства.

Например, сделаем атрибут `state` свойством:

```
class RobotVacuum:
    def __init__(self, state):
        self.__state = state    # атрибут стал свойством

    @property                   # теперь это - геттер свойства
    def state(self):
        return self.__state
```




Теперь можно обращаться к `state` как к обычной переменной. Это не будет обращением напрямую — в этот момент вызывается геттер:

```
# создали объект
robo_vacuum = RobotVacuum("в ожидании")

# свойство вызывают как атрибут, а не как метод
print(robo_vacuum.state)  # Вывод: в ожидании
```

Часто разработчики пишут свойство только с геттером, без сеттера. Такое свойство можно прочитать, но нельзя изменить.

Это полезно, когда значение атрибута вычисляется динамически. То есть оно меняется через другие методы. Например, как «баланс» в классе «банковский счёт». Он привязан к операциям: пополнению и снятию средств. Нельзя выставить значение баланса из головы.

### Как написать сеттер для свойства

Чтобы значение свойства можно было менять, для него пишут сеттер.

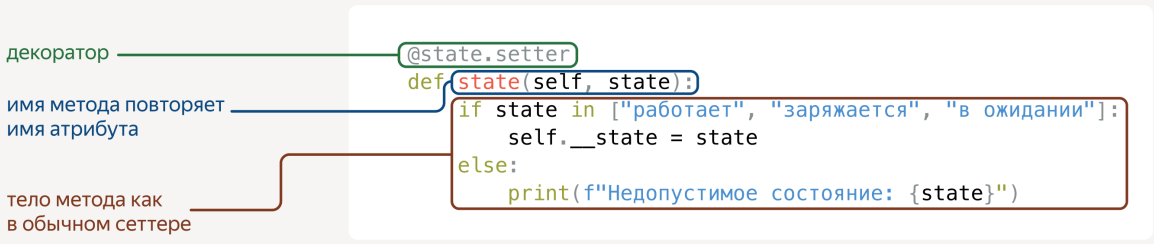
> 📌 Важно: сеттер пишут, когда уже есть геттер. Это требование языка: не бывает свойств с сеттером, но без геттера.

Сеттер для свойства выглядит почти так же, как сеттер для атрибута. Но есть пара отличий:

1. У свойства должен быть геттер с декоратором `@property`;
2. Метод называют так же, как свойство. Например, `state`;
3. Над методом ставят декоратор `@<имя_свойства>.setter`. Например, `@state.setter`.

**Пример целиком:**

```
@state.setter             # сеттер для свойства state
def state(self, state):
    if state in ["работает", "заряжается", "в ожидании"]:
        self.__state = state
    else:
        print(f"Недопустимое состояние: {state}. Состояние может быть 'работает', 'заряжается', или 'в ожидании'")
```




Оба метода для свойства — геттер и сеттер — носят одно имя. Их всегда называют так же, как свойство. Обращай на это внимание. Иначе программа не будет работать.

Так всё вместе:

```
class RobotVacuum:
    def __init__(self, state):
        self.__state = state

    @property                   # геттер для свойства state
    def state(self):
        return self.__state

    @state.setter              # сеттер для свойства state
    def state(self, state):
        if state in ["работает", "заряжается", "в ожидании"]:
            self.__state = state
        else:
            print(f"Недопустимое состояние: {state}. Состояние может быть 'работает', 'заряжается', или 'в ожидании'")
```

Атрибут `state` стал свойством. На это указывают декораторы `@property` и `@state.setter` над геттером и сеттером.

Со свойством `state` можно работать напрямую —как если бы это был обычный атрибут. При этом действия через геттер и сеттер всё ещё выполняются:

```
# создали объект
robo_vacuum = RobotVacuum("в ожидании")

# обратились к свойству, чтобы получить состояния
print(robo_vacuum.state)  # Вывод: в ожидании

# перезаписали свойство, чтобы установить состояние
robo_vacuum.state = "работает"
print(robo_vacuum.state)  # Вывод: работает

# попытались использовать свойство, чтобы установить недопустимое состояние
robo_vacuum.state = "летает"  # Вывод: Недопустимое состояние: летает. Состояние может быть 'работает', 'заряжается', или 'в ожидании'
print(robo_vacuum.state)   # Вывод: работает
```

### Несколько свойств в классе

Если в классе несколько приватных атрибутов, их все можно сделать свойствами. Здесь нет ограничений.

**Пример.** В классе `Student` два свойства: «возраст» — `age` и «оценка» — `grade`. У каждого из них есть сеттер и геттер:

```
class Student:
    def __init__(self, age=0, grade=0):
        self.__age = age
        self.__grade = grade

    @property                # геттер для свойства age
    def age(self):
        return self.__age
```

```
    @age.setter              # сеттер для свойства age
    def age(self, value):
        if 0 <= value <= 100:
            self.__age = value
        else:
            self.__age = 0

    @property                # геттер для свойства grade
    def grade(self):
        return self.__grade

    @grade.setter            # сеттер для свойства grade
    def grade(self, value):
        if 0 <= value <= 100:
            self.__grade = value
        else:
            self.__grade = 0
```

Значение для `age` и `grade` устанавливаются через свойства:

```
# создали объект
student = Student()
student.age = 20       # работает сеттер age
student.grade = 90     # работает сеттер grade
```

Получить их можно так же — обратившись напрямую к свойствам.

```
print(student.age)    # работает геттер age
# Выведет: 20

print(student.grade)  # работает геттер grade
# Выведет: 90
```

При этом методы всё ещё работают. Если установить недопустимые значения, они обновятся до нуля:

```
student.age = -1
student.grade = 101
print(student.age)      # Выведет: 0
print(student.grade)    # Выведет: 0
```

### Чем полезны свойства

> 💡 Геттеры и сеттеры пришли в Python из других языков. С ним код работает, но выглядит не питонически.

Свойства и декоратор `@property` — это питоническое решение. С ними код выглядит чище и аккуратнее. А ещё его удобнее использовать.

Так тебе не нужно:

- придумывать названия методам — они отзываются на имя атрибута;
- вызывать методы, если работаешь со значением атрибута.

Контроль доступа при этом сохраняется: геттеры и сеттеры работают так же, а вызывать их проще.