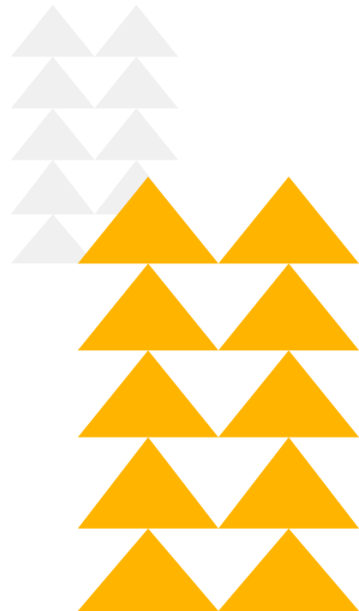
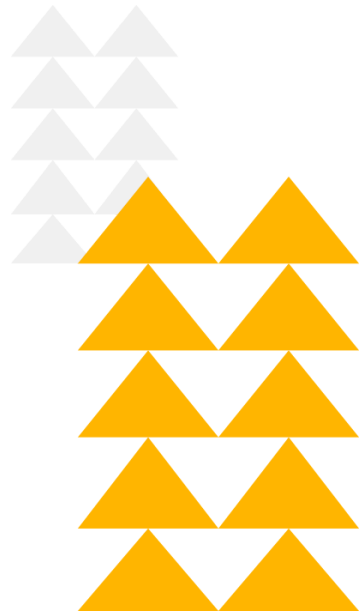
Классы

- Синтаксис объявления классов
- Атрибуты, связанные и несвязанные методы, `__dict__`, `__slots__`
- Статические методы и методы класса
- Свойства, декоратор `@property`
- Наследование, перегрузка методов и функция `super`
- Декораторы классов
- Магические методы



Синтаксис объявления классов

```
01.  class Counter:
02.      def __init__(self, initial=0):  # <- конструктор
03.          self.value = initial  # <- запись атрибута
04.      def increment(self):
05.          self.value += 1
06.      def get(self):
07.          return self.value  # <- чтение атрибута
08.
09.  c = Counter(42)
10.  c.increment()
11.  c.get()  # -> 43
```

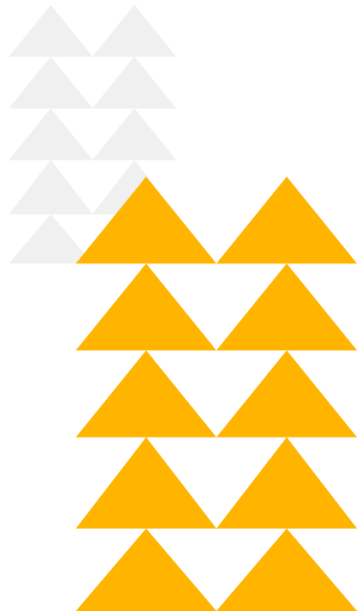


Классы и self

В отличие от Java и C++ в Python нет “магического” ключевого слова `this`. Первый аргумент конструктора `__init__` и всех остальных методов — экземпляр класса, который принято называть `self`.

Синтаксис языка не запрещает называть его по-другому, но так делать не рекомендуется

```
01. class Noop:
02.     def __init__(whatever):
03.         pass
04. noop = Noop()
```

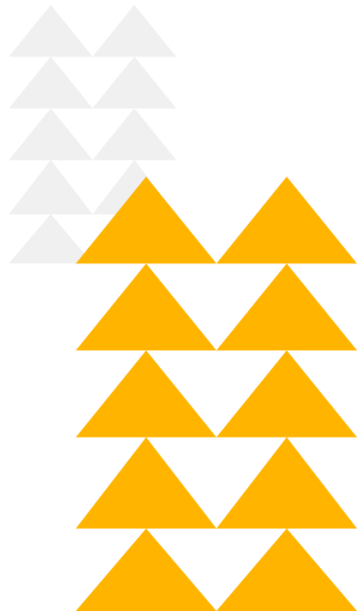


Классы, экземпляры и атрибуты

Аналогично другим ООП языкам Python разделяет атрибуты экземпляра и атрибуты класса.

Атрибуты добавляются к экземпляру посредством присваивания к `self` конструкцией вида:

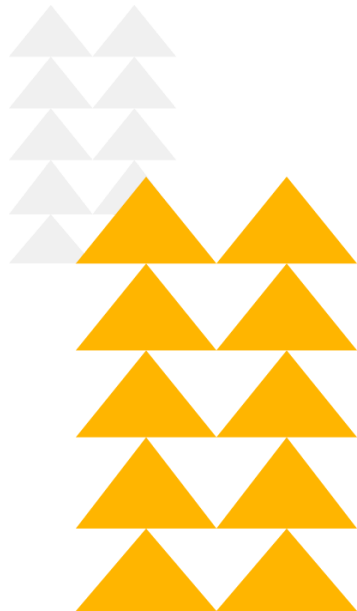
```
self.some_attribute = value
```



Классы, экземпляры и атрибуты (2)

Атрибуты класса объявляются в теле класса или прямым присваиванием к объекту класса.

```
>>> class Counter:
        all_counters = []

        def __init__(self, initial=0):
            Counter.all_counters.append(self)

>>> Counter.another_class_attribute = 42
```

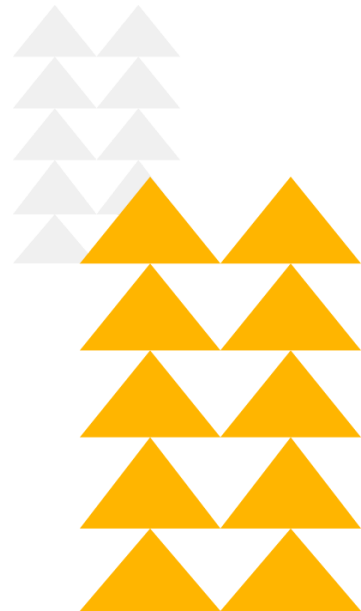


Соглашения об именовании атрибутов и методов

В Python нет модификаторов доступа к атрибутам и методам: почти всё можно читать и присваивать.

Для того чтобы различать публичные и внутренние атрибуты визуально, к внутренним атрибутам добавляют в начало символ подчеркивания:

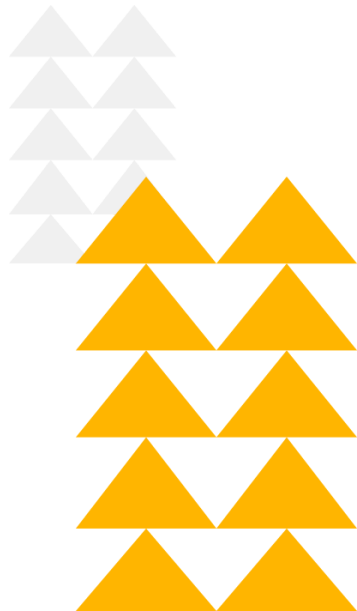
```
01. class Noop:
02.     some_attribute = 42
03.     _internal_attribute = []
```



Соглашения об именовании атрибутов и методов (2)

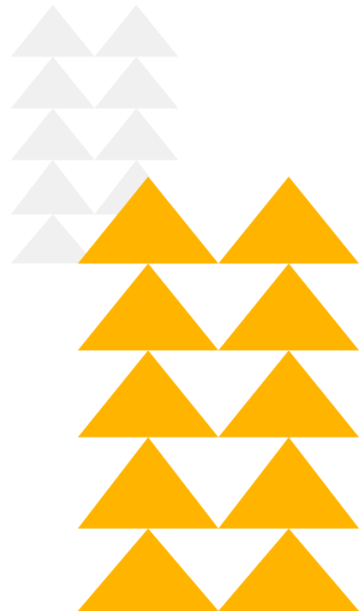
Особо ярые любители контроля используют два подчёркивания (name mangling)

```
>>> class Noop:
        __very_internal_attribute = 42

>>> Noop.__very_internal_attribute
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: type object 'Noop' has no attribute [...]

>>> Noop._Noop__very_internal_attribute
42
```



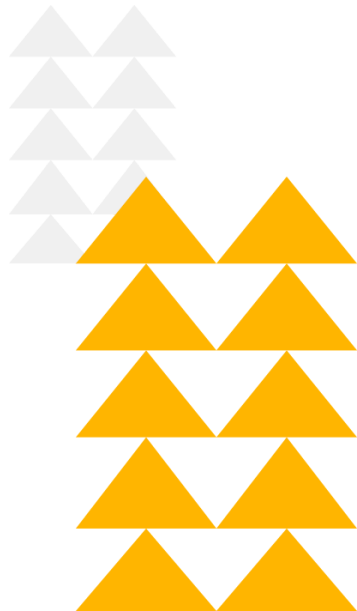
Name mangling

```
01.  class Mapping:
02.      def __init__(self, iterable):
03.          self.items_list = []
04.          self.__update(iterable)
05.      def update(self, iterable):
06.          for item in iterable:
07.              self.items_list.append(item)
08.  __update = update
09.
10.  class MappingSubclass(Mapping):
11.      def update(self, keys, values):
12.          # provides new signature for update()
13.          # but does not break __init__()
```



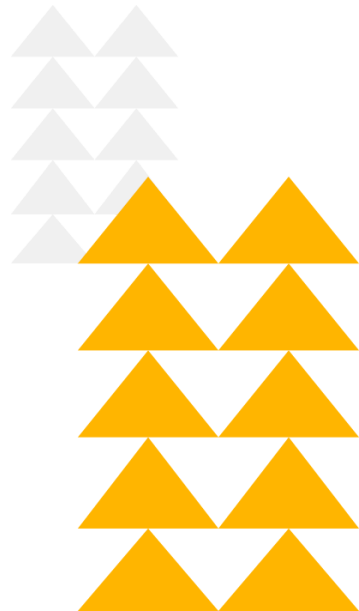
Внутренние атрибуты классов

```
01.  >>> class Noop:
02.          """I do nothing at all."""
03.  >>> Noop.__doc__
04.  'I do nothing at all.'
05.  >>> Noop.__name__
06.  'Noop'
07.  >>> Noop.__module__
08.  '__main__'
09.  >>> Noop.__bases__
10.  (<class 'object'>,)
```



Внутренние атрибуты экземпляров

```
01.  >>> noop = Noop()
02.  >>> noop.__class__
03.  <class '__main__.Noop'>
04.  >>> noop.__dict__  # <- словарь атрибутов объекта
05.  {}
```

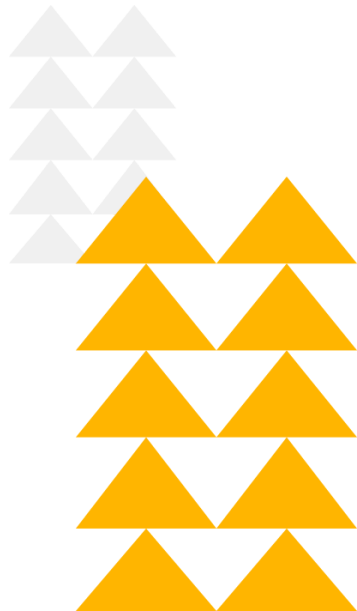


Подробнее о __dict__

Все атрибуты (и методы) объекта доступны в виде словаря:

```
01. >>> noop.some_attribute = 42
02. >>> noop.__dict__
03. {'some_attribute': 42}
```

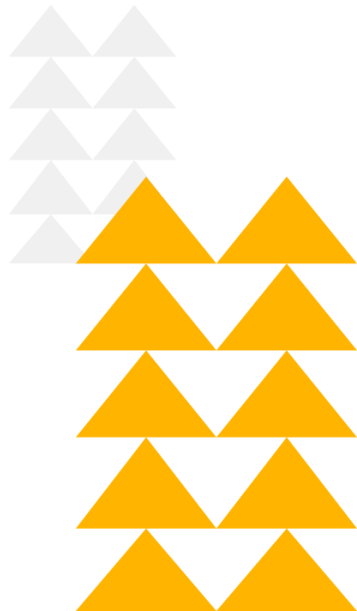
- Добавление, изменение и удаление атрибутов — это фактически операции со словарём.
- Поиск значения атрибута происходит динамически в момент выполнения программы.



Классы и __slots__

С помощью специального аттрибута класса `__slots__` можно зафиксировать множество возможных атрибутов экземпляра:

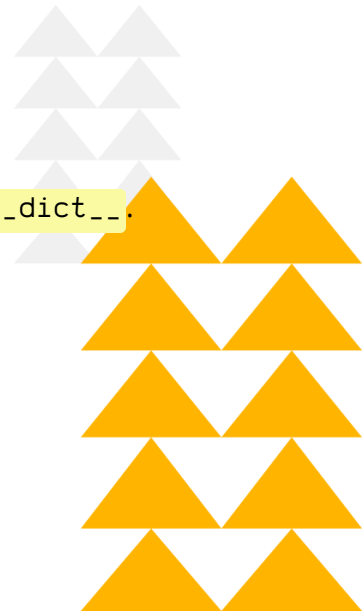
```
01.  >>> class Noop:
02.          __slots__ = ["some_attribute"]
03.  >>> noop = Noop()
04.  >>> noop.some_attribute = 42
05.  >>> noop.some_attribute
06.  42
```



Классы и __slots__

```
01.  >>> noop.some_other_attribute = 100500
02.  Traceback (most recent call last):
03.  File "<stdin>", line 1, in <module>
04.  AttributeError: 'Noop' object has no attribute [...]
05.
06.  >>> noop.__dict__
07.  Traceback (most recent call last):
08.    File "<stdin>", line 1, in <module>
09.  AttributeError: 'Noop' object has no attribute '__dict__'
```

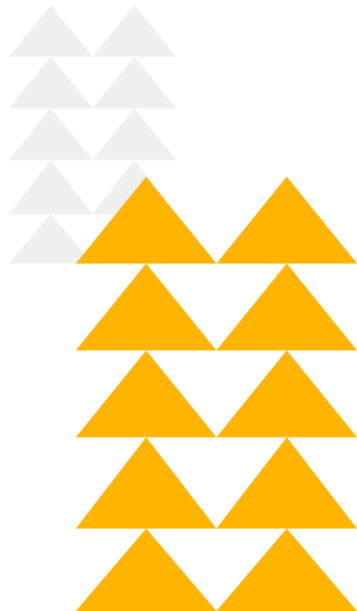
Экземпляры класса с указанным `__slots__` требуют меньше памяти, потому что у них отсутствует `__dict__`.



Связанные и несвязанные методы

У связанного метода первый аргумент уже зафиксирован и равен соответствующему экземпляру:

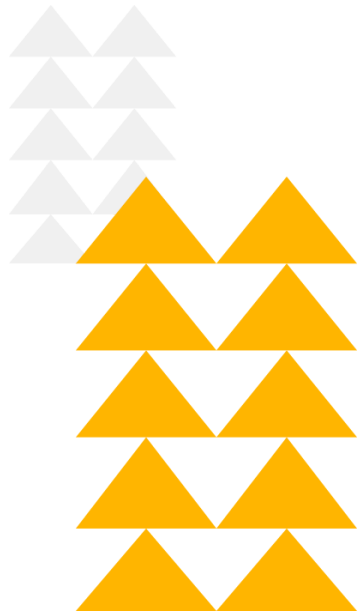
```
>>> class SomeClass:
        def do_something(self):
            print("doing something")
>>> instance = SomeClass()
>>> instance.do_something
<bound method SomeClass.do_something of [...]>
>>> instance.do_something()
doing something
```



Связанные и несвязанные методы

Несвязанному методу необходимо явно передать экземпляр первым аргументом в момент вызова:

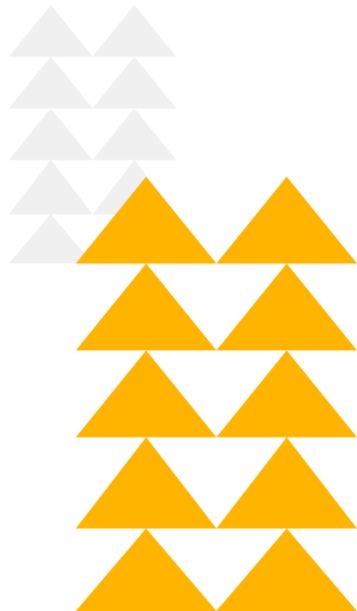
```
>>> SomeClass.do_something
<function SomeClass.do_something at 0x105466a60>
>>> instance = SomeClass()
>>> SomeClass.do_something(instance)
doing something
```



Статические методы

Декоратор `staticmethod` позволяет объявить статический метод, то есть просто функцию, внутри класса:

```
>>> class SomeClass:
        @staticmethod
        def do_something():
            pass

>>> SomeClass.do_something()
```

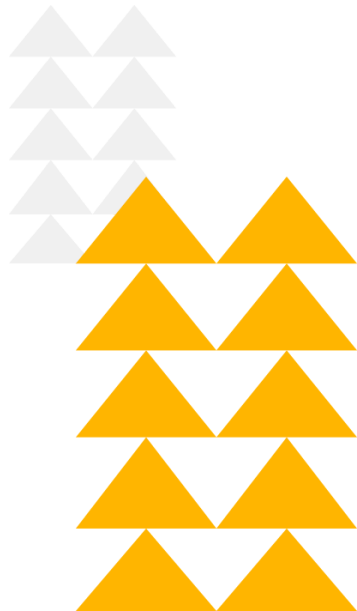


Методы класса

Для объявления методов класса используется похожий декоратор `classmethod`.

Первый аргумент метода класса — непосредственно сам класс, а не его экземпляр.

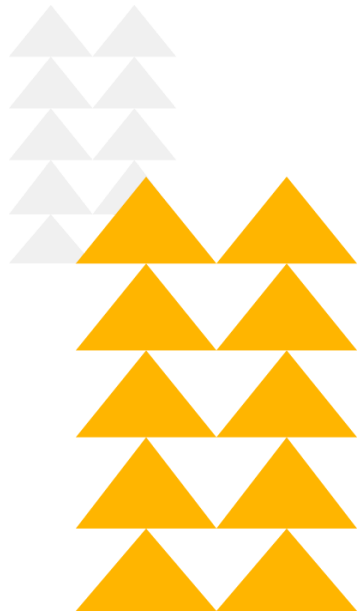
```
>>> class Config:
        @classmethod
        def load_from(cls, path):
            pass

>>> Config.load_from("./application.ini")
<__main__.Config at 0x10f558208>
```



Свойства

Механизм свойств позволяет объявлять атрибуты, значение которых вычисляется в момент обращения:

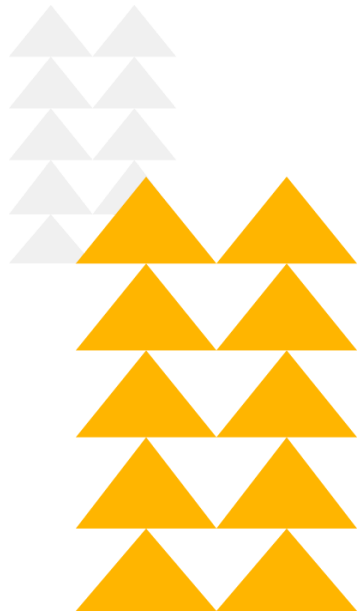
```
>>> class Path:
        def __init__(self, current):
            self.current = current
        @property
        def parent(self):
            return Path(dirname(self.current))
>>> p = Path("./examples/some_file.txt")
>>> p.parent
'./examples'
```



Наследование

Синтаксис оператора `class` позволяет унаследовать объявляемый класс от произвольного количества других классов:

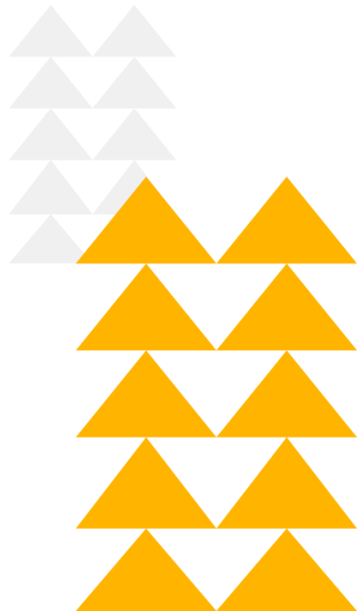
```
>>> class Counter:
        def __init__(self, initial=0):
            self.value = initial

>>> class NeverCountCounter(Counter):
        def get(self):
            return 42
```



Поиск имени при наследовании

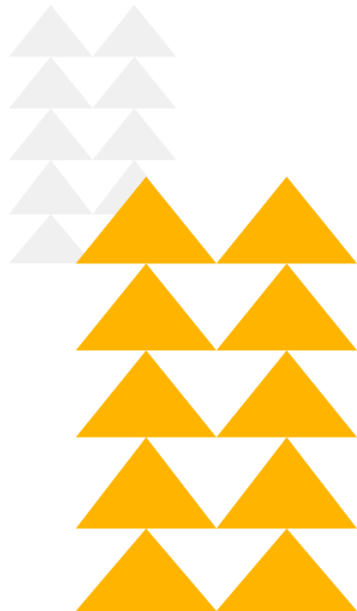
Поиск имени при обращении к атрибуту или методу ведётся сначала в `__dict__` экземпляра. Если там имя не найдено, оно ищется в классе, а затем рекурсивно во всей иерархии наследования.

```
>>> c = NeverCountCounter()  # <- вызывает Counter.__init__
>>> c.get()  # <- вызывает NeverCountCounter.get
42
>>> c.value # <- c.__dict__["value"]
```



Перегрузка методов и функция super

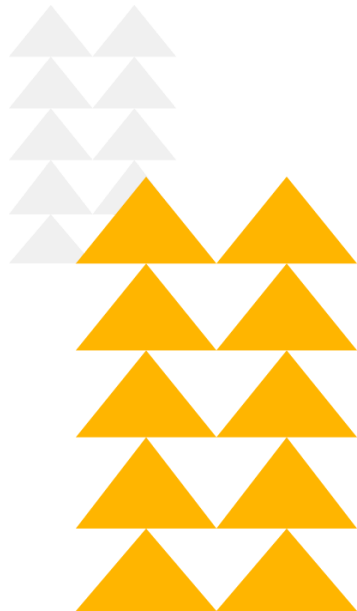
```
01.  >>> class Counter:
02.          def __init__(self, initial=0):
03.              self.value = initial
04.
05.  >>> class MyCounter(Counter):
06.          def __init__(self, initial=0):
07.              super().__init__(initial)  # <- Counter.__init__(initial)
08.              self.initial = initial
09.
10.  >>> vars(MyCounter())
11.  {'initial': 0, 'value': 0}
```



Предикат isinstance

Предикат `isinstance` принимает объект и класс и проверяет, что объект является экземпляром класса:

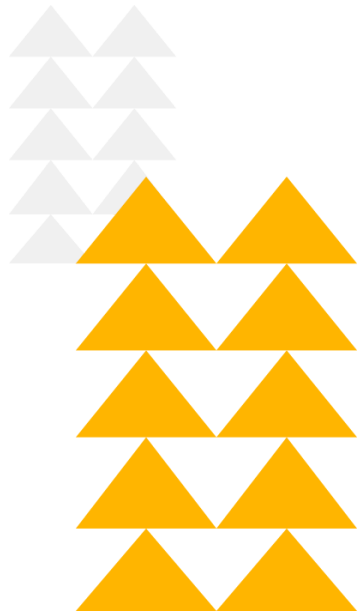
```
>>> class A:
        pass

>>> class B(A):
        pass

>>> isinstance(B(), A)
True
```



Предикат issubclass

Предикат `issubclass` принимает два класса и проверяет, что первый класс является потомком второго:

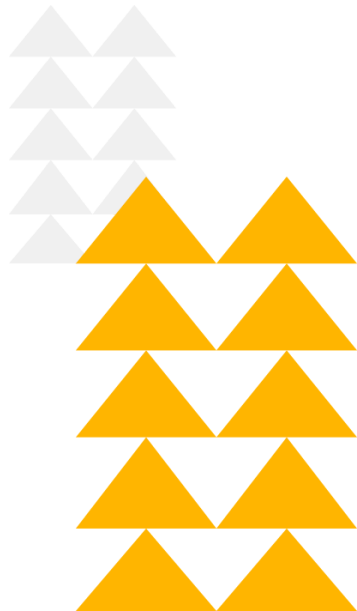
```
>>> class A:
        pass

>>> class B(A):
        pass

>>> issubclass(B, A)
True
```



MRO

```
class A:
    def f(self):
        print('A.f')

class B:
    def f(self):
        print('B.f')

class C(A, B):
    pass
```

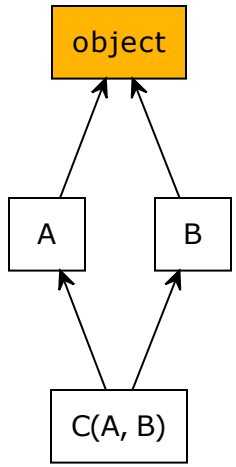
В случае множественного наследования Python использует алгоритм линеаризации C3 для определения метода, который нужно вызвать.



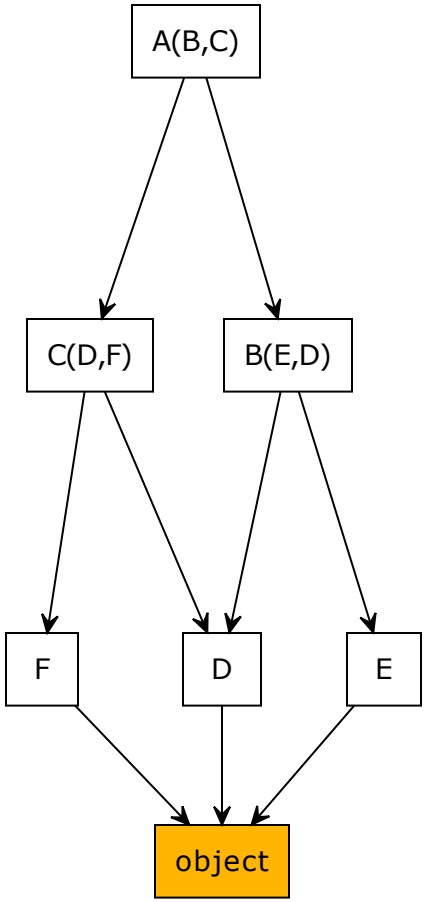
MRO

Получить линеаризацию иерархии наследования можно с помощью метода `mro`:

```
>>> C.mro()  # C.__mro__
[<class '__main__.C'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>]
>>> C().f()
A.f
```



MRO



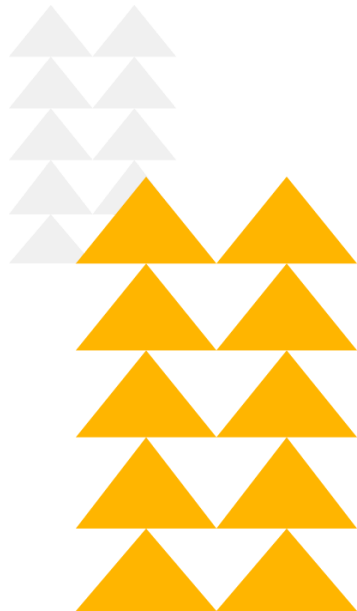
Результат работы алгоритма C3 далеко не всегда тривиален, поэтому использовать сложные иерархии множественого наследования не рекомендуется.

Классы-примеси

Классы-примеси позволяют выборочно модифицировать поведение класса в предположении, что класс реализует некоторый интерфейс.

```
class ThreadSafeMixin:
    def get_lock(self):
        pass

    def increment(self):
        with self.get_lock():
            super().increment()

    def get(self):
        with self.get_lock():
            super().get()
```

```
class ThreadSafeCounter(
    ThreadSafeMixin, Counter
):
    pass
```



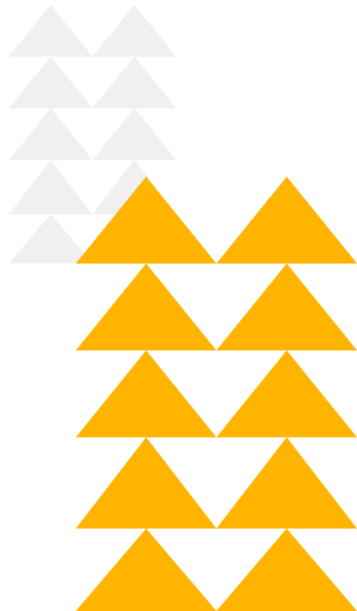
Декораторы классов

Синтаксис декораторов работает не только для функций, но и для классов.

```
01. @decorator
02. class Noop:
03.     pass
```

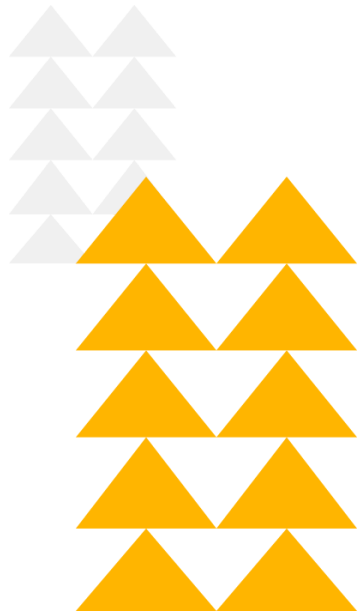
В этом случае декоратор — это функция, которая принимает класс и возвращает другой, возможно, преобразованный, класс.

Декораторы классов можно также использовать вместо чуть более магических классов-примесей.



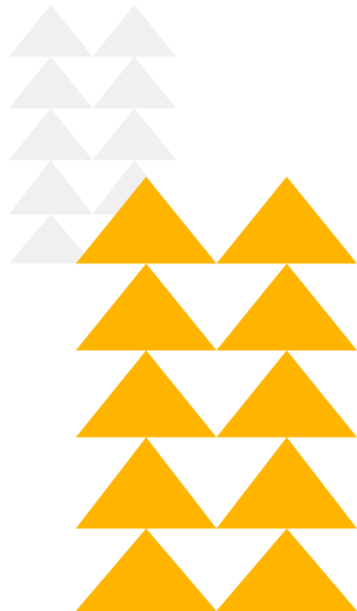
Декораторы классов: thread_safe

```
01.  def thread_safe(cls):
02.      orig_get = cls.get
03.      orig_increment = cls.increment
04.
05.      def increment(self):
06.          with self.get_lock():
07.              return orig_increment()
08.
09.      def get(self):
10.          with self.get_lock():
11.              return orig_get()
12.
13.      cls.get = get
14.      cls.increment = increment
15.
16.      return cls
```



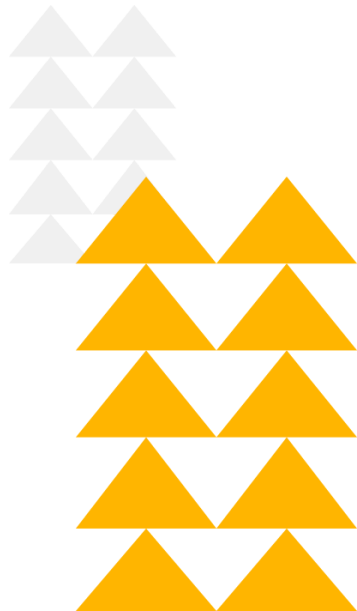
Декораторы классов: deprecated

```
>>> def deprecated(cls):
        orig_init = cls.__init__

        @functools.wraps(cls.__init__)
        def new_init(self, *args, **kwargs):
            warnings.warn(
                cls.__name__ + 'is deprecated', category=DeprecationWarning)
            return orig_init(self, *args, **kwargs)
        cls.__init__ = new_init
        return cls

>>> @deprecated
    class Counter: pass

>>> c = Counter()
__main__:6: DeprecationWarning: Counter is deprecated.
```



Take away: классы

- В отличие от большинства объектно-ориентированных языков Python:
  - делает передачу ссылки на экземпляр явной, `self` — первый аргумент каждого метода,
  - позволяет объявлять не только методы класса, но и статические методы, то есть просто функции в пространстве имён класса,
  - реализует механим свойств — динамически вычисляемых атрибутов,
  - поддерживает изменение классов с помощью декораторов.



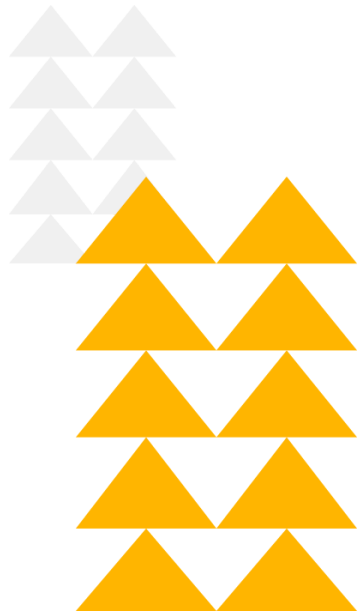
“Магические” методы

- “Магическими” называются внутренние методы классов, например, метод `__init__`.
- С помощью “магических” методов можно:
  - управлять доступом к атрибутам экземпляра,
  - перегрузить операторы, например, операторы сравнения или арифметические операторы,
  - определить строковое представление экземпляра или изменить способ его хеширования.
- Мы рассмотрим только часть наиболее используемых методов.

“Магический” метод __getattr__

Метод `__getattr__` вызывается при попытке прочитать значение несуществующего атрибута:

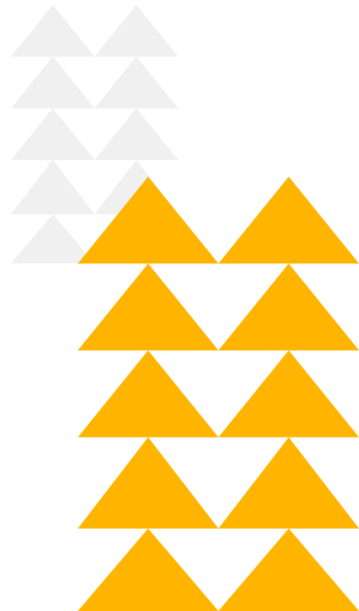
```
01. >>> class Noop:
02.         pass
03. >>> Noop().foobar
04. Traceback (most recent call last):
05. File "<stdin>", line 1, in <module>
06. AttributeError: 'Noop' object has no attribute 'foobar'
```



“Магический” метод __getattr__

Определим метод `__getattr__` для класса Noop:

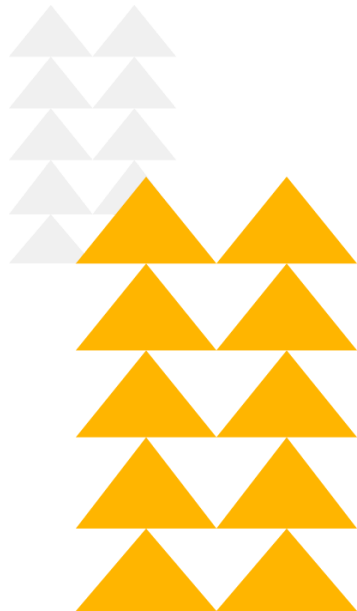
```
>>> class Noop:
        def __getattr__(self, name):
            return name

>>> Noop().foobar
'foobar'
```



“Магические” методы __setattr__ и __delattr__

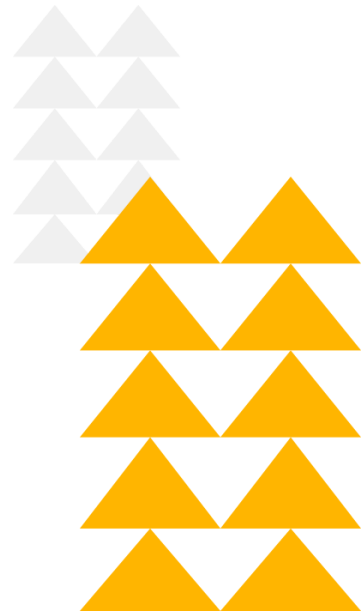
Методы `__setattr__` и `__delattr__` позволяют управлять изменением значения и удалением атрибутов.

В отличие от `__getattr__` они вызываются для всех атрибутов, а не только для несуществующих.



“Магические” методы __setattr__ и __delattr__

```
01.  class Guarded:
02.      guarded = ()
03.
04.      def __setattr__(self, name, value):
05.          assert name not in self.guarded
06.          super().__setattr__(name, value)
07.
08.  class Noop(Guarded):
09.      guarded = (”read_only_attr”,)
10.
11.      def __init__(self):
12.          self.__dict__[”read_only_attr”] = 42  # Зачем это?
```



Функции getattr, setattr и delattr

Функция `getattr` позволяет безопасно получить значение атрибута экземпляра класса по его имени:

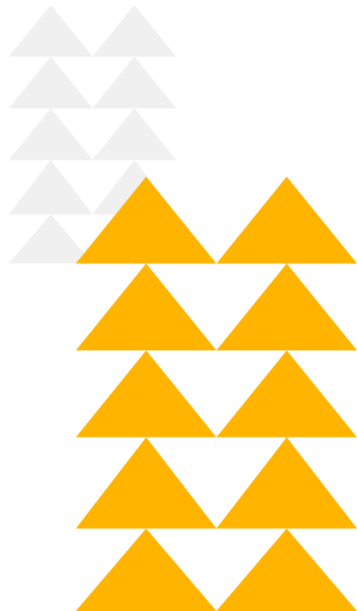
```
>>> class Noop:
        some_attribute = 42
>>> noop = Noop()
>>> getattr(noop, "some_attribute")
42
>>> getattr(noop, "some_other_attribute", 100500)  # <- нет ошибки
100500
```

Комплементарные функции `setattr` и `delattr` добавляют и удаляют атрибут.

“Магические” методы: операторы сравнения

Чтобы экземпляры класса поддерживали все операторы сравнения, нужно реализовать внушительное количество “магических” методов:

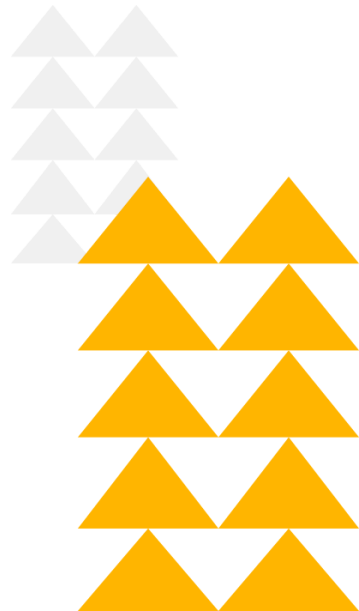
```
01. instance.__eq__(other)  # instance == other
02. instance.__ne__(other)  # instance != other
03. instance.__lt__(other)  # instance < other
04. instance.__le__(other)  # instance <= other
05. instance.__gt__(other)  # instance > other
06. instance.__ge__(other)  # instance >= other
```



“Магические” методы: операторы сравнения

В модуле `functools` есть декоратор, облегчающий реализацию операторов сравнения `total_ordering`.

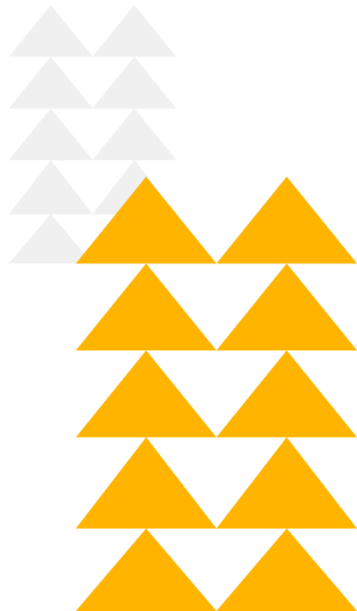
```
>>> import functools
>>> @functools.total_ordering
    class Counter:
        def __eq__(self, other):
            return self.value == other.value

        def __lt__(self, other):  # или <=, >, >=
            return self.value < other.value
```



“Магический” метод __call__

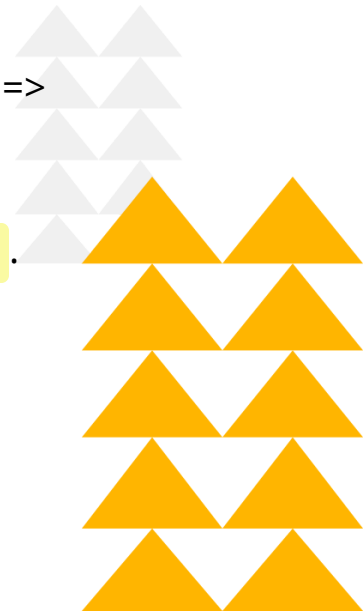
Метод `__call__` позволяет “вызывать” экземпляры классов, имитируя интерфейс функции:

```
>>> class Identity:
        def __call__(self, x):
            return x

>>> Identity()(42)
42
```



“Магический” метод __hash__

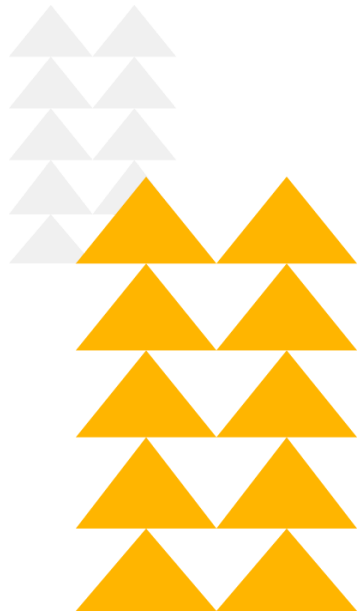
- Метод `__hash__` используется для вычисления значения хеш-функции.
- Реализация по умолчанию гарантирует, что одинаковое значение хеш функции будет только у физически одинаковых объектов, то есть: x is y <=> hash(x) == hash(y).
- Несколько очевидных рекомендаций:
  - Метод `__hash__` имеет смысл реализовывать только вместе с методом `__eq__`. При этом реализация `__hash__` должна удовлетворять: x == y => hash(x) == hash(y)
  - Для изменяемых объекто можно ограничиться только методом `__eq__`.



“Магический” метод __bool__

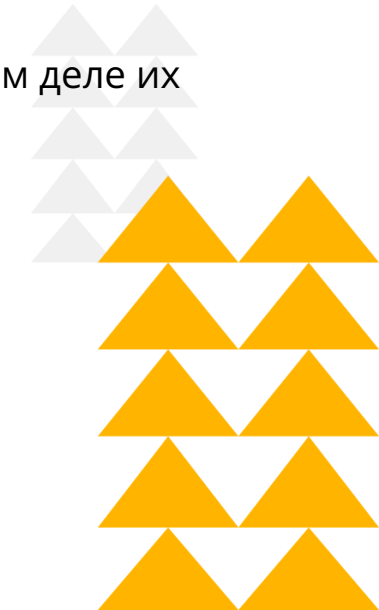
Метод `__bool__` для проверки значения на истинность, например в условии оператора `if`.

```
>>> class Counter:
        def __init__(self, initial=0):
            self.value = initial
        def __bool__(self):
            return bool(self.value)
>>> c = Counter()
>>> if not c: print(“no counts yet”)
no counts yet
```



Take away: магические методы

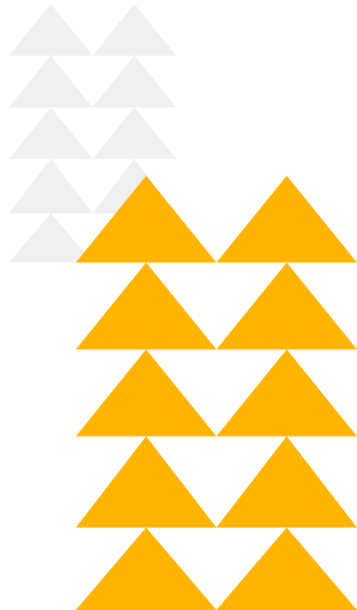
- “Магические” методы позволяют уточнить поведение экземпляров класса в различных конструкциях языка.
- Например, с помощью магического метода `__str__` можно указать способ приведения экземпляра класса, а с помощью метода `__hash__` — алгоритм хеширования состояния класса.
- Мы рассмотрели небольшое подмножество “магических” методов, на самом деле их много больше: практически любое действие с экземпляром можно специализировать для конкретного класса.



SOLID

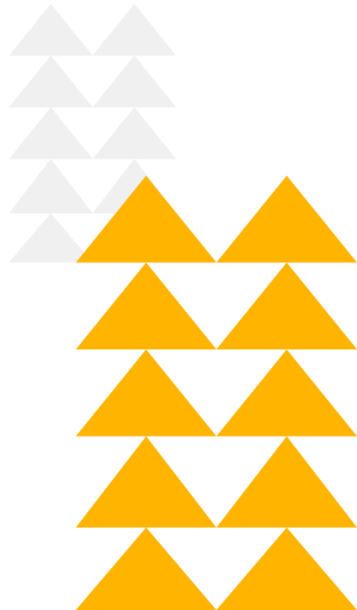
При создании программ использование принципов SOLID способствует созданию такой системы, которую будет легко поддерживать и расширять в течение долгого времени.

- Принцип единственной ответственности
- Принцип открытости/закрытости
- Принцип подстановки Барбары Лисков
- Принцип разделения интерфейса
- Принцип инверсии зависимостей



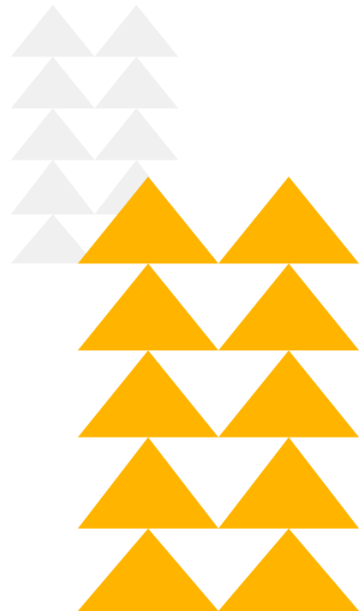
Принцип единственной ответственности (SPR)

Должна быть одна и только одна причина для изменения класса.



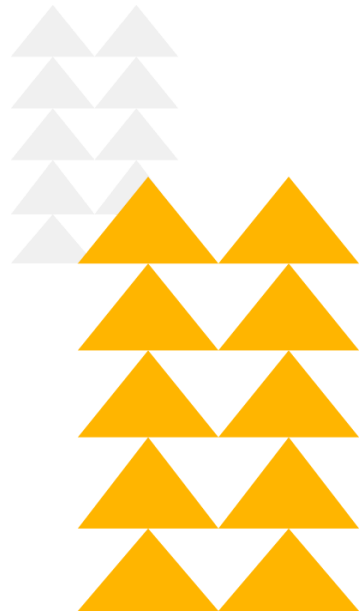
Пример: SRP

```
01.  class Animal:
02.      def __init__(self, name: str):
03.          self.name = name
04.      def get_name(self) -> str:
05.          pass
06.      def save(self, animal: Animal):
07.          pass
```



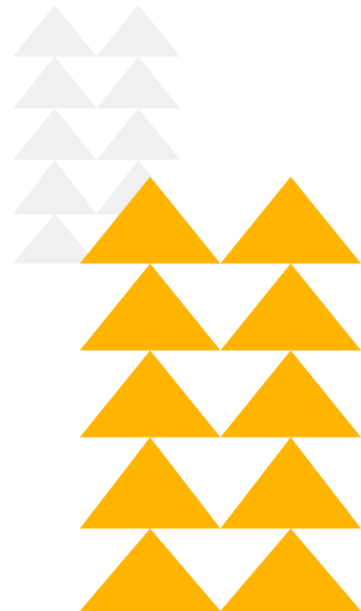
Пример: SRP

```
01.  class Animal:
02.      def __init__(self, name: str):
03.          self.name = name
04.      def get_name(self):
05.          pass
06.
07.  class AnimalDB:
08.      def get_animal(self, id) -> Animal:
09.          pass
10.      def save(self, animal: Animal):
11.          pass
```



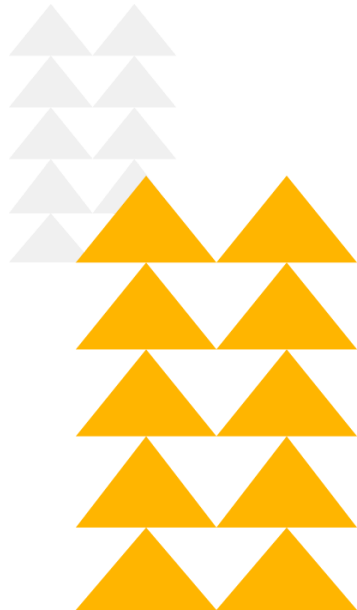
Пример: SRP

```
01.  class Animal:
02.      def __init__(self, name: str):
03.          self.name = name
04.          self.db = AnimalDB()
05.      def get_name(self):
06.          return self.name
07.      def get(self, id):
08.          return self.db.get_animal(id)
09.      def save(self):
10.          self.db.save(animal=self)
```



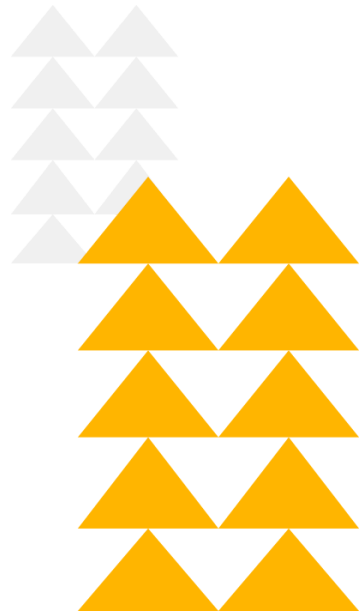
Принцип открытости/закрытости (OCP)

Программные сущности (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменения.



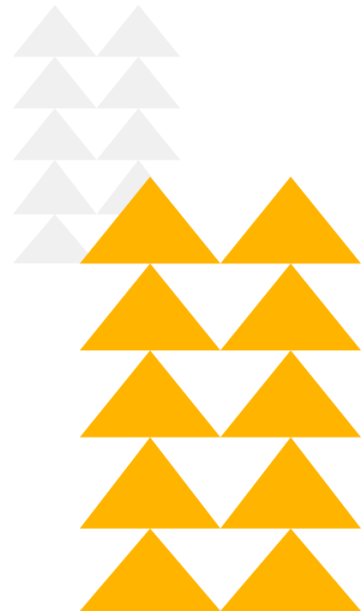
Пример: OCP

```
01.  animals = [
02.      Animal('lion'),
03.      Animal('mouse')
04.  ]
05.
06.  def animal_sound(animals: list):
07.      for animal in animals:
08.          if animal.name == 'lion':
09.              print('roar')
10.          elif animal.name == 'mouse':
11.              print('squeak')
12.
13.  animal_sound(animals)
```



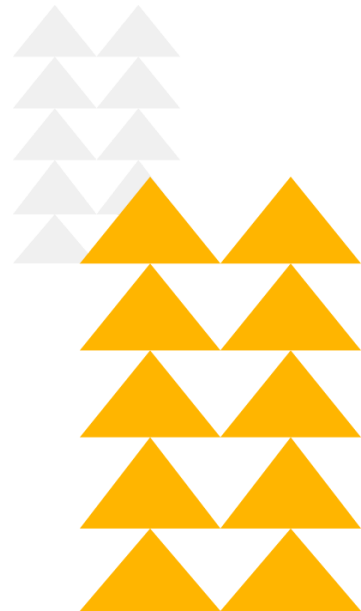
Пример: OCP

```
01.  class Animal:
02.      def make_sound(self):  # <- определим интерфейс
03.          raise NotImplementedError
04.
05.  class Lion(Animal):
06.      def make_sound(self):  # <- конкретная реализация
07.          return 'roar'
08.
09.  class Mouse(Animal):
10.      def make_sound(self):  # <- конкретная реализация
11.          return 'squeak'
```



Пример: OCP

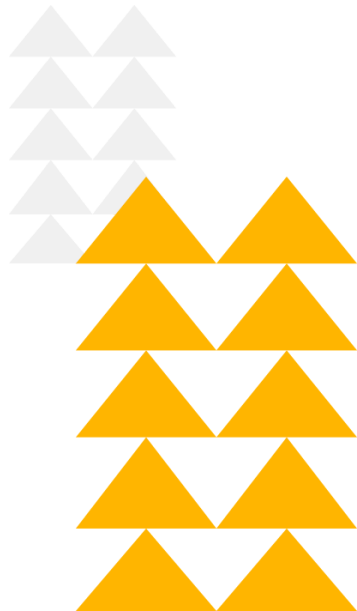
```
01.  def animal_sound(animals: list):
02.      for animal in animals:
03.          print(animal.make_sound())
04.
05.  animal_sound(animals)
```



Принцип подстановки (LSP)

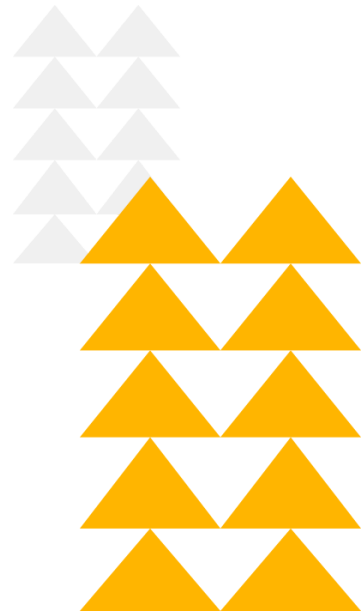
Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.



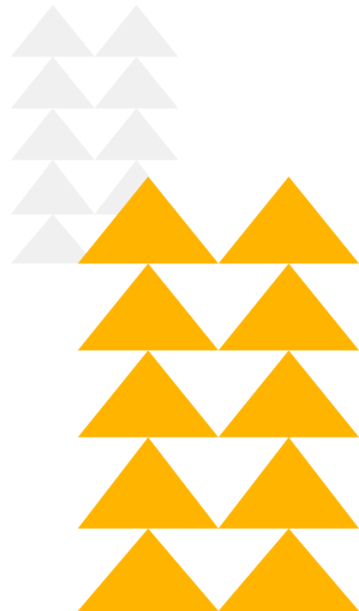
Пример: LSP

```
01.  def animal_leg_count(animals: list):
02.      for animal in animals:
03.          if isinstance(animal, Lion):
04.              print(lion_leg_count(animal))
05.          elif isinstance(animal, Mouse):
06.              print(mouse_leg_count(animal))
07.          elif isinstance(animal, Pigeon):
08.              print(pigeon_leg_count(animal))
09.
10.  animal_leg_count(animals)
```



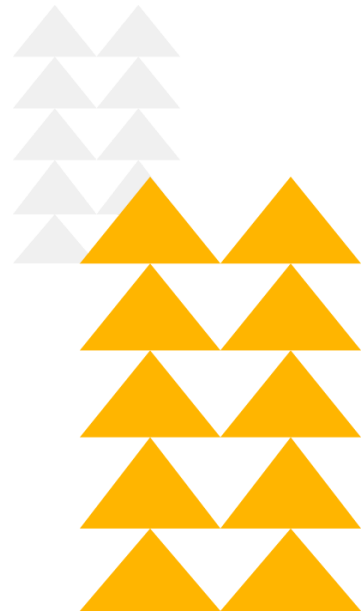
Пример: LSP

```
01.  def animal_leg_count(animals: list):
02.      for animal in animals:
03.          print(animal.leg_count())
04.
05.  animal_leg_count(animals)
```



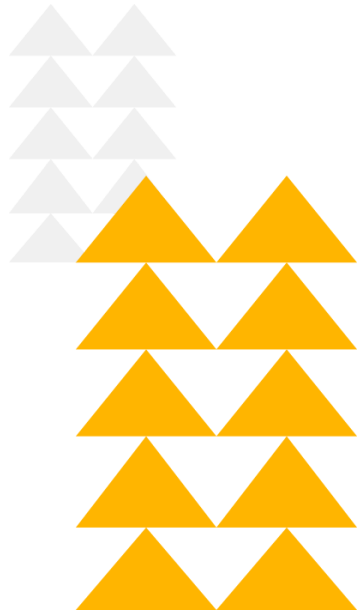
Пример: LSP

```
01.  class Animal:
02.      def leg_count(self) -> int:
03.          raise NotImplementedError
04.
05.  class Lion(Animal):
06.      def leg_count(self) -> int:
07.          return 4
```



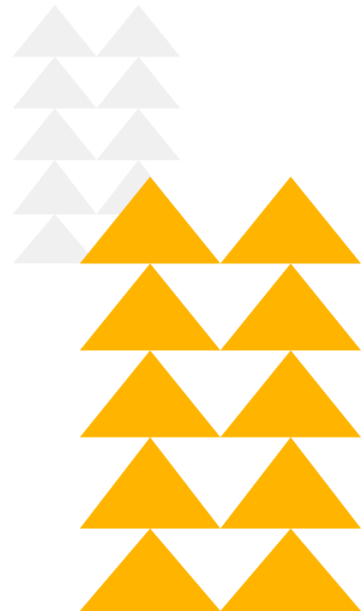
Принцип разделения интерфейса (ISP)

Клиенты не должны зависеть от методов, которые они не используют.



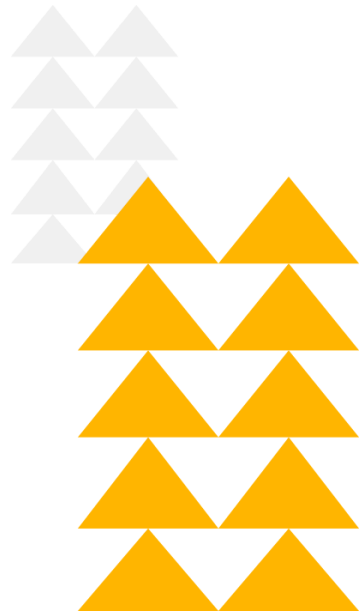
Пример: ISP

```
01.  class IShape:
02.      def draw_square(self):
03.          raise NotImplementedError
04.
05.      def draw_rectangle(self):
06.          raise NotImplementedError
07.
08.      def draw_circle(self):
09.          raise NotImplementedError
```



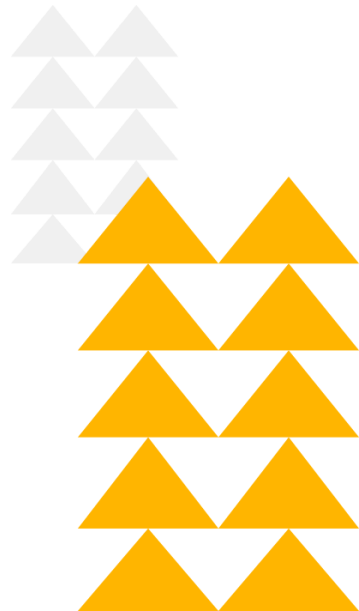
Пример: ISP

```
01.  class Circle(IShape):
02.      def draw_square(self):  # <- unused
03.          pass
04.
05.      def draw_rectangle(self):  # <- unused
06.          pass
07.
08.      def draw_circle(self):
09.          pass
```



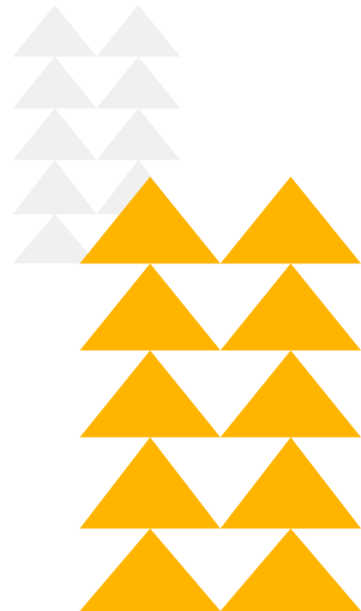
Пример: ISP

```
01.  class Square(IShape):
02.      def draw_square(self):
03.          pass
04.
05.      def draw_rectangle(self):  # <- unused
06.          pass
07.
08.      def draw_circle(self):  # <- unused
09.          pass
```



Пример: ISP

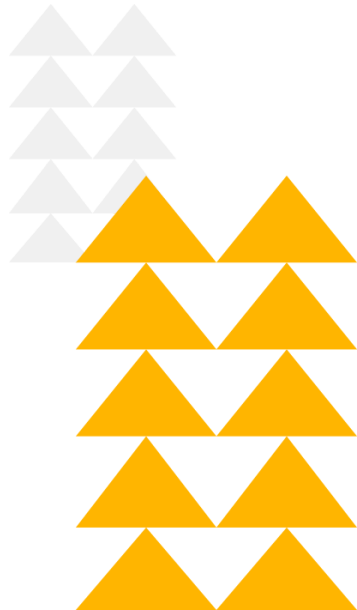
```
01.  class IShape:
02.      def draw(self):
03.          raise NotImplementedError
04.
05.  class Circle(IShape):
06.      def draw(self):
07.          pass
08.
09.  class Square(IShape):
10.      def draw(self):
11.          pass
```



Принцип инверсии зависимостей (DIP)

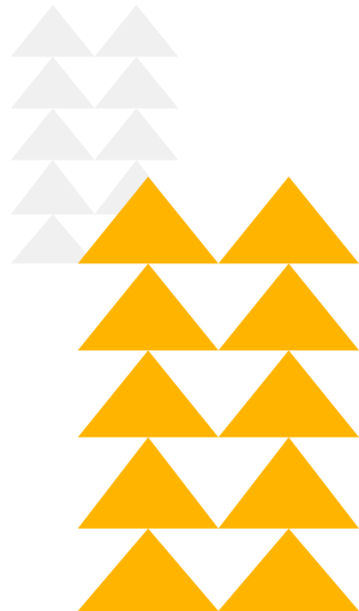
Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



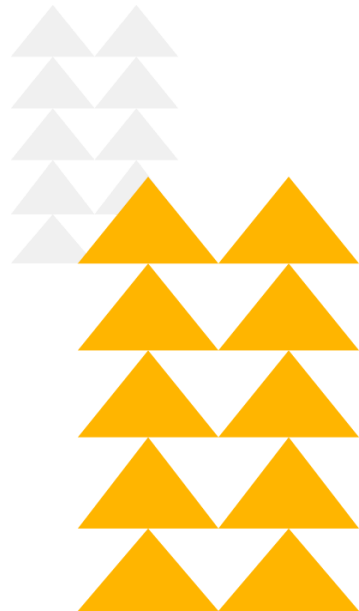
Пример: DIP

```
01. class XMLHttpService(XMLHttpRequestService):
02.     pass
03.
04. class Http:
05.     def __init__(self, xml_http_service: XMLHttpService):
06.         self.xml_http_service = xml_http_service
07.     def get(self, url: str, options: dict):
08.         self.xml_http_service.request(url, 'GET')
09.     def post(self, url, options: dict):
10.         self.xml_http_service.request(url, 'POST')
```



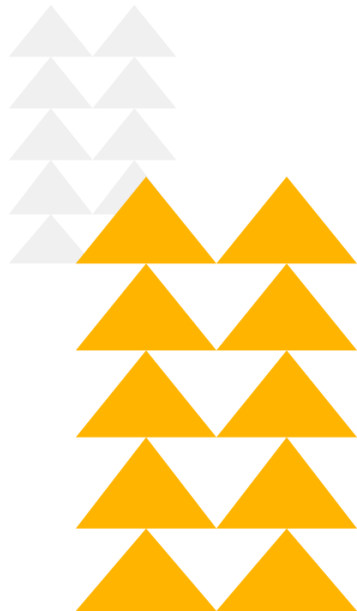
Пример: DIP

```
01.  class Connection:  # <- base interface
02.      def request(self, url: str, options: dict):
03.          raise NotImplementedError
04.
05.  class Http:
06.      def __init__(self, http_connection: Connection): # <- depends from interface
07.          self.http_connection = http_connection
08.      def get(self, url: str, options: dict):
09.          self.http_connection.request(url, 'GET')
10.      def post(self, url, options: dict):
11.          self.http_connection.request(url, 'POST')
```



Пример: DIP

```
01.  class MockHttpService(Connection):  # <- concrete implementation
02.      def request(self, url: str, options:dict):
03.          pass
04.
05.  http = Http(http_connection=MockHttpService())
06.  http.get("http://example.com")
```



Спасибо за внимание

Виталий Кудёлка

Разработчик программного обеспечения

vital.kudzelka@leverx.com

github.com/vitalk