ChatGPT

Dunder-методы и связанные концепции в Python

Dunder-методы (double-underscore methods) – специальные методы, описывающие низкоуровневое поведение объектов и классов. Они используются для создания нестандартного поведения при создании, уничтожении объектов, доступе к атрибутам, итерации, контекстному управлению и т.д. В этом документе мы подробно рассмотрим основные dunder-методы и связанные с ними концепции.

Метод `__new__` и `super().__new__(cls, ...)`

Метод `__new__()` отвечает за **создание экземпляра класса** и вызывается перед `__init__()`. Он статический (хотя мы не помечаем его `@staticmethod`) и принимает первым аргументом класс `cls`. По умолчанию он наследуется от `object` и выделяет память под новый объект. Типичная реализация `__new__()` вызывает `super().__new__(cls, ...)` для создания базового экземпляра, а затем при необходимости модифицирует его. В официальной документации сказано:

> *Typical implementations create a new instance of the class by invoking the superclass's* `__new__()` *method using* `super().__new__(cls, ...)` *with appropriate arguments and then modifying the newly created instance as necessary before returning it* [1].
> Если `__new__()` возвращает экземпляр `cls`, то затем вызывается `__init__()`. Если возвращается что-то иное, `__init__()` не вызывается [2]. Это позволяет, например, переопределять поведение при наследовании от неизменяемых типов (например, `int`, `tuple`) или в метаклассах.

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        # Создаем новый объект через базовый __new__
        instance = super().__new__(cls)
        # Дополнительная логика (например, кеширование, логирование)
        print(f"Создаем экземпляр {cls.__name__}")
        return instance

    def __init__(self, value):
        self.value = value

obj = MyClass(10)  # При создании сначала вызовется __new__, затем __init__.
```

Здесь `super().__new__(cls)` создаёт объект стандартным способом. Если объект должен быть создан с определёнными аргументами (например, для неизменяемых типов), их можно передать дальше в `super().__new__`.

Метод `__del__`

Метод `__del__()` (деструктор/финализатор) вызывается **перед уничтожением экземпляра** (когда ссылающихся на него объектов не остаётся). В документации:

> *Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor.* [3]
> Если базовый класс имеет `__del__()`, то в производном классе его следует явно вызывать для корректной очистки.

Важно помнить, что `__del__` не гарантированно будет вызван при завершении работы интерпретатора [4], и что внутри `__del__` может возникнуть нежелательное поведение (например, **объект может «воскреснуть»**, сохранив новую ссылку в себе [5]). Кроме того, `del x` не сразу вызывает `x.__del__()`: он лишь уменьшает счётчик ссылок, и `__del__()` вызовется только когда объект будет действительно собран [6].

```
class Obj:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} создан.")
    def __del__(self):
        print(f"{self.name} уничтожается.")

o = Obj("Пример")
del o  # Явный вызов: может привести к __del__
```

При разработке `__del__` рекомендуется избегать «сложных» операций (особенно блокировок), так как они могут выполняться при завершении работы интерпретатора, когда глобальные переменные уже очищены [7].

Паттерн Singleton (синглтон)

Singleton – паттерн, гарантирующий, что у класса есть **только один экземпляр**. В Python часто реализуется через переопределение `__new__()`:

```
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            # При первом создании вызываем super().__new__
            cls._instance = super().__new__(cls)
        return cls._instance

s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True: обе переменные указывают на один экземпляр
```

Здесь при первом создании объекта `cls._instance` устанавливается в созданный объект, а при последующих вызовах `__new__` возвращается уже существующий экземпляр, вместо создания нового.

Другой подход – хранить одиночку в глобальной области модуля (модульные переменные всегда создают по одному объекту), однако через `__new__` паттерн Singleton легко интегрируется с классической ООП.

Обобщения (Generics) через `__class_getitem__`

Начиная с Python 3.7 (PEP 560) в ядре появился метод `__class_getitem__()`, который позволяет **параметризовать классы при помощи синтаксиса с квадратными скобками** для целей типизации. Например, `list[int]` вызывает именно `list.__class_getitem__(int)` [8] и возвращает объект типа `GenericAlias`.

Если ваш класс задаёт метод `__class_getitem__`, он автоматически является метод-@classmethod (декоратор `@classmethod` не нужен) [9]. Цель его – поддержка runtime-параметризации generic-классов (типизации) [10]. В общем случае, при выражении `Класс[args]`, интерпретатор действует так (упрощённо) [11]:

```
from inspect import isclass

def subscribe(obj, x):
    class_of_obj = type(obj)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)
    else:
        raise TypeError(f"'{class_of_obj.__name__}' object is not
subscriptable")
```

В частности, стандартный тип `type` **не** определяет `__getitem__`, поэтому `list[int]`, `dict[str, float]` и т.д. вызывают `list.__class_getitem__` [12]. Например:

```
>>> list[int]
list[int]              # появляется GenericAlias
>>> type(list[int])
<class 'types.GenericAlias'>
```

При наличии пользовательского метакласса с `__getitem__()` (например, у `enum.EnumMeta`), класс может вести себя иначе: например, `EnumClass['MEMBER']` возвращает константу enum, а не GenericAlias [13].

Обратите внимание, что `__class_getitem__` в основном предназначен для статической типизации. Использование его «вдруг так» может не поддерживаться сторонними анализаторами типа `mypy` [14].

Индексируемые объекты (`__getitem__`, `__setitem__`, `__delitem__`) и подписываемость

- **Подписываемый** объект (например, список, словарь) – это тот, у которого определён метод `__getitem__()`, отвечающий за доступ `obj[key]`. Если класса объекта нет `__getitem__`, при попытке `obj[...]` будет **TypeError**: `"'X' object is not subscriptable"` [15].
- `__class_getitem__` мы рассмотрели выше – он вызывается, когда подписывается сам **класс** (не экземпляр) и нет `__getitem__` у метакласса.

Методы для индексирования:

- `object.__getitem__(self, key)`: реализует `self[key]`.
- Для **последовательностей** (`list`, `tuple` и своих аналогов) обычно `key` – целое (или `slice`). Если индекс выходит за границу, должен быть брошен `IndexError` [16]. Это критично: цикл `for x in obj` при отсутствии `__iter__()` будет перебирать `obj[0], obj[1], ...` до `IndexError`, считая это концом последовательности [17] [18]. В примере ниже цикл остановится на `IndexError`.

```
class Seq:
    def __getitem__(self, index):
        if index < 5:
            return index*10
        else:
            raise IndexError
for x in Seq():
    print(x, end=' ')  # Выведет 0 10 20 30 40, потом IndexError -> конец.
```

- Для **отображений** (`dict`-подобных) `__getitem__` по несуществующему ключу должен бросать `KeyError`.

- `object.__setitem__(self, key, value)`: реализует присваивание `self[key] = value`. Рекомендуется определять для изменяемых контейнеров (словари, списки). Требует тем же исключениям при недопустимом `key` [19].

- `object.__delitem__(self, key)`: реализует удаление элемента `del self[key]`. Тоже аналогичные примечания [20].

Важно: если у класса есть `__getitem__`, то выражение `obj[...]` вызывает его сразу, даже если `obj` – класс. Однако для **классов** сначала проверяется `__class_getitem__` (если класс вызывает `[]`) [11]. Если ни `__getitem__`, ни `__class_getitem__` не определены, будет `TypeError`.

Пример: объект с `__getitem__`, без `__iter__`, поддерживает старый протокол итерации:

```
class MySeq:
    def __getitem__(self, i):
        if i < 3:
```

```
            return chr(ord('A') + i)
        raise IndexError

for ch in MySeq():
    print(ch)  # A B C
```

Цикл автоматически обрабатывает `IndexError` как конец.

Если нужно явное прерывание итерации в новом стиле, то надо реализовать `__iter__` и `__next__` (см. ниже). В противном случае итератором по умолчанию будет объект `Seq().__iter__()` (при наличии `__iter__`), либо серия вызовов `__getitem__` до `IndexError`.

Итераторы: `__iter__`, `__next__`, `StopIteration`

Итератором называют объект, который возвращает следующий элемент коллекции при вызове метода `__next__()`. Сам протокол итерации описывается так:

1. При запуске цикла `for x in iterable` выполняется `iterator = iter(iterable)`.
2. Если у `iterable` есть метод `__iter__`, он должен вернуть объект-итератор. Если `__iter__` отсутствует, Python попробует старый протокол: создать индекс 0, и на каждом шаге вызывать `iterable[0]`, `iterable[1]` и т.д. до `IndexError` [21] [18].
3. Итератор должен иметь метод `__next__(self)`, возвращающий следующее значение. Когда элементов больше нет, `__next__()` должен выбросить `StopIteration` (не `return None`). Это сигнализирует циклу о завершении.

Пример простой итерации нового стиля:

```
class CountToN:
    def __init__(self, n):
        self.n = n
        self.i = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            val = self.i
            self.i += 1
            return val
        else:
            raise StopIteration

for num in CountToN(5):
    print(num, end=' ')  # 0 1 2 3 4
```

- `__iter__(self)`: возвращает сам итератор (обычно `return self` для самих себя) [22].

- `__next__(self)`: возвращает очередное значение или бросает `StopIteration`. После этого цикла `for` прерывается (конец).

При использовании *генераторов* (функции с `yield`) Python сам создаёт объект-итератор: каждый `yield` даёт следующее значение, а когда функция завершает выполнение (`return` или конец тела), автоматически выбрасывается `StopIteration` [23].

В сумме: - Если реализован `__iter__`/`__next__`, `for` использует их.
- Если нет, но есть `__getitem__`, то пытается индексную последовательность (см. выше).
- Каждый раз при отработке итератора возвращается `StopIteration` на выходе.

Контекстные менеджеры: `__enter__`, `__exit__`

Конструкция `with` позволяет управлять ресурсами (файлы, блокировки и др.) в стиле RAII: гарантирует вызов методов при входе и выходе из блока. Класс контекстного менеджера должен определять методы `__enter__` и `__exit__`. Семантика `with` (PEP 343, см. мануал) равносильна примерно такому коду [24]:

```
manager = EXPRESSION           # создаём объект менеджера
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)         # вызов __enter__
try:
    TARGET = value             # если указан as TARGET, сохраняем результат
    SUITE                      # выполняем тело with
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise                  # передаём исключение в __exit__
finally:
    if not hit_except:
        exit(manager, None, None, None)  # завершающий вызов __exit__
```

- `__enter__(self)`: вызывается при входе в блок. Может возвращать объект, привязываемый к ключевому слову `as`.
- `__exit__(self, exc_type, exc_val, exc_tb)`: вызывается **всегда** при выходе из блока (даже при исключении) [24]. Если при выходе было исключение, его тип/значение/traceback передаются в `__exit__`.
    - Если `__exit__` вернёт `True`, исключение подавляется; если `False` или `None`, то исключение повторно пробрасывается после выхода.

```
class MyContext:
    def __enter__(self):
        print("Вход в with")
        return "ресурс"  # вернём что-нибудь
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type:
            print("Исключение:", exc_type, exc_val)
            return False  # не подавляем
```

```
        print("Выход из with")
        # возвращать True подавит исключение

with MyContext() as r:
    print("Используем", r)
    # В блоке можно генерировать исключения
```

Если `__enter__` завершился без ошибок, `__exit__` гарантированно будет вызван [24]. При наличии нескольких контекстов в `with A() as a, B() as b:`, они обрабатываются как вложенные `with` [25].

Дескрипторы: data-descriptor, non-data-descriptor, `__get__`, `__set__`, `__delete__`, `__set_name__`, `@property`

Дескриптор – это объект (обычно атрибут класса) с одним из методов `__get__`, `__set__` или `__delete__`. Он позволяет управлять тем, что происходит при доступе, присваивании или удалении атрибута.

- `__get__(self, obj, objtype)`: вызывается при чтении атрибута. `obj` – экземпляр, из которого читаем (`None`, если чтение у класса), `objtype` – класс.
- `__set__(self, obj, value)`: вызывается при `obj.attr = value`.
- `__delete__(self, obj)`: вызывается при `del obj.attr`.

Data-descriptor vs non-data-descriptor:
- **Data-descriptor** – реализует `__set__` или `__delete__` (или оба).
- **Non-data-descriptor** – только `__get__`.

Главное различие – при разрешении имени атрибута: data-descriptor имеет **приоритет** над атрибутом экземпляра, а non-data – нет. То есть, если у объекта-экземпляра есть атрибут с тем же именем, то запись в `obj.__dict__` перекрывает non-data-дескриптор, но data-дескриптор всё равно будет вызываться [26]:

> *If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If it has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.* [26].

Пример data-дескриптора (с `__get__` и `__set__`):

```
class EvenValue:
    def __get__(self, obj, objtype=None):
        return obj._value
    def __set__(self, obj, val):
        if val % 2 != 0:
            raise ValueError("Только четные значения")
        obj._value = val

class My:
    value = EvenValue()
```

```
    def __init__(self, x):
        self.value = x  # пойдёт через EvenValue.__set__

m = My(4)
print(m.value)   # через EvenValue.__get__ вернёт 4
# m.value = 3    # ValueError: Только четные значения
```

`__delete__` аналогично может перехватывать `del obj.attr`. **(Не путать с `__del__`!)** – `__delete__` удаляет атрибут класса/экземпляра, тогда как `__del__` – деструктор объекта (см. выше).

Метод `__set_name__`: вызывается при создании класса (в Python 3.6+) и сообщает дескриптору имя атрибута, под которым он сохранился в классе [27]. Сигнатура `__set_name__(self, owner, name)`, где `owner` – класс, `name` – имя атрибута. Это удобно, если дескриптору нужно знать своё имя или класс. Пример:

```
class Field:
    def __set_name__(self, owner, name):
        self.name = name
    def __get__(self, obj, objtype=None):
        return obj.__dict__[self.name]

class My:
    x = Field()  # при создании класса вызовется Field.__set_name__(My, 'x')
```

Декоратор `@property`: на деле `property` – это готовый data-дескриптор. Он позволяет объявить геттер/сеттер в виде методов:

```
class Person:
    def __init__(self, age):
        self._age = age
    @property
    def age(self):         # __get__
        return self._age
    @age.setter
    def age(self, value):  # __set__
        if value < 0:
            raise ValueError
        self._age = value

p = Person(30)
print(p.age)  # 30 (__get__)
p.age = -5    # ValueError (__set__)
```

`@property` задействует `__set_name__` неявно, присваивая свои методы (обратимся к документации для деталей).

Доступ к атрибутам: `__getattribute__` vs `__getattr__`, `__setattr__`, `__delattr__`

Python позволяет перехватывать **любой доступ** к атрибутам, присваивание и удаление:

- `object.__getattribute__(self, name)` – вызывается **всегда** при попытке получить `self.name` (за исключением оптимизаций для «специальных» методов). Если не переопределён, реализует стандартный поиск: сначала в `self.__dict__`, затем по классам (с учётом дескрипторов). Если `__getattribute__` не найдёт атрибут, должно выбросить `AttributeError`. Чтобы избежать бесконечной рекурсии внутри своего `__getattribute__`, **надо** явно вызывать базовый метод: например, `object.__getattribute__(self, name)` [28].

- `object.__getattr__(self, name)` – вызывается **только если** обычный поиск атрибута не нашёл ничего и выбросил `AttributeError` [29]. То есть это резервный метод для вычисления «пропущенных» атрибутов. Если атрибут всё же найден (или `__getattribute__` возвращает значение), `__getattr__` **не** вызывается [30]. Его реализация должна вернуть значение или снова бросить `AttributeError`, чтобы сигнализировать об ошибке.

- `object.__setattr__(self, name, value)` – вызывается при `self.name = value` вместо обычного `self.__dict__[name] = value`. Для нормального сохранения в атрибут используйте `object.__setattr__(self, name, value)` [31] внутри своего метода, иначе возникнет бесконечный рекурсивный вызов.

- `object.__delattr__(self, name)` – аналогично для `del self.name` (вызывает удаление атрибута, или бросает `AttributeError`, если не возможно) [32]. Внутри можно вызывать `object.__delattr__(self, name)` для стандартного удаления.

Как избежать бесконечной рекурсии:
При переопределении `__getattribute__` или `__setattr__`, если вы обратитесь к `self.attr` внутри них, снова вызовется тот же метод. Поэтому всегда внутри них нужно обращаться к атрибутам через базовый класс, например:

```
class C:
    def __getattribute__(self, name):
        print(f"Доступ к {name}")
        return object.__getattribute__(self, name)

    def __setattr__(self, name, value):
        print(f"Установка {name} = {value}")
        object.__setattr__(self, name, value)

c = C()
c.x = 5  # вызывает __setattr__, который вызывает базовый object.__setattr__
print(c.x)  # __getattribute__
```

Если внутри `__getattr__`, можно спокойно использовать `self.__dict__` или `object.__getattribute__`, потому что `__getattribute__` уже завершился без результатов и не будет рекурсии.

Функция `super()`: прокси-объект, атрибуты `__self__`, `__self_class__`, `__thisclass__`, `mappingproxy`, ошибки `TypeError`

`super()` создаёт объект-прокси, через который можно вызывать методы родительских классов по методу поиска в MRO. Синтаксис: `super(type, obj)` или в методе без аргументов `super()`.

- `super(type, obj)` возвращает прокси, настроенный на поиск в MRO начиная после `type`. Второй аргумент (`obj`) может быть либо экземпляром (должен удовлетворять `isinstance(obj, type)`), либо вторым аргументом может быть **класс** (в случае `classmethod`, тогда `issubclass(type2, type)` должно быть True). Если переданы неверные аргументы, бросается `TypeError` [33].
- Нуль-аргументный `super()` (обычно внутри метода класса) автоматически подбирает `type` как текущий класс, а `obj` как первый аргумент функции (`self`) [34].

Объект `super` имеет атрибуты:
- `__self__` – это либо экземпляр (если вызов внутри метода экземпляра), либо класс (для методов класса), либо `None` (для unbound-super).
- `__self_class__` – фактический класс экземпляра (аналог `type(__self__)`).
- `__thisclass__` – класс, от которого начнётся поиск методов (тот, который мы указали первым в `super`).

Пример:

```
class A:
    def hello(self): print("A")
class B(A):
    def hello(self):
        print("B")
        super().hello()  # вызывает A.hello
```

У прокси `super().__dict__` есть тип `mappingproxy` – это неизменяемый словарь атрибутов класса [35]. Это означает, что через `super` нельзя напрямую присвоить что-то в классовый словарь родителя.

Важный момент: если вы попытаетесь вызвать `super()` с некорректными аргументами, возникнет `TypeError`. Например, `super(int, some_obj)` бросит, если `some_obj` не является экземпляром `int` или его подкласса, и наоборот.

Подробнее о `super()` см. официальную документацию: она подчёркивает, что `super()` – это прокси для вызова родительских методов в соответствии с MRO, и поддерживает *кооперативный множественный наследование* [36].

Метод разрешения порядка (MRO): ромбовидное наследование и C3-линеаризация

В случае множественного наследования классов может возникнуть ситуация “ромба”, когда класс наследует от двух классов, у которых общий предок. Python использует алгоритм **C3-линеаризации**, гарантирующий детерминированный и «моно́тонный» порядок обхода базовых классов.

Например:

```
class A:
    def method(self): print("A")
class B(A):
    def method(self): print("B"); super().method()
class C(A):
    def method(self): print("C"); super().method()
class D(B, C):
    def method(self): print("D"); super().method()

d = D()
d.method()
# Выведет:
# D
# B
# C
# A
```

Здесь MRO для `D` будет `D -> B -> C -> A -> object`. Алгоритм C3 учитывает порядок наследования и сохраняет последовательность, совместимую со всеми цепочками наследования. Он гарантирует, что каждый класс появляется **после всех своих базовых классов** и при этом порядок наследования везде соблюдается. Официальная документация отмечает:

> *Поиск по базовым классам использует C3-порядок разрешения методов, который правильно работает даже в случаях «ромбовидного» наследования* [37] .

Функция `type.mro()` возвращает кортеж (`__mro__`), описывающий этот порядок [38] . При обращении к методу Python будет искать его последовательно в классах из `__mro__`.

`type(name, bases, dict)`: создание динамических классов и метаклассы

В Python классы сами являются объектами типа `type`. Вы можете динамически создать класс с помощью встроенной функции `type`. Есть два вызова:

1. **Одно-аргументный:** `type(obj)` – просто возвращает тип объекта.
2. **Три-аргументный:** `type(name, bases, dict)` – создаёт новый класс. Аргументы:
3. `name` – имя класса (строка), станет его `__name__`.

4. `bases` – кортеж базовых классов. Если пуст, базовый класс по умолчанию – `object`.
5. `dict` – словарь атрибутов/методов (имитация тела класса).

   With three arguments, return a new type object. This is essentially a dynamic form of the class statement. [39] .
   Например, эти два определения эквивалентны:

```
class X:
    a = 1

# то же самое, динамически
X = type('X', (), {'a': 1})
```

При создании класса через `type`, исходный `dict` может быть скопирован или обёрнут специальным объектом, чтобы стать `__dict__` класса. Как указано в документации, классы обычно используют `types.MappingProxyType` – **отображение-прокси** – чтобы сделать собственный `__dict__` только для чтения [40] . Поэтому атрибуты класса нельзя менять напрямую через `.__dict__`, а только через присваивание `Class.attr = value`.

Метаклассы: по умолчанию метаклассом всех классов является `type`. Если нужно контролировать процесс создания класса, можно определить свой класс-метакласс (наследник `type`) и передать его в метапараметре: `class C(metaclass=Meta): ...`. Тогда вызов `type(name, ...)` пойдёт через `Meta.__new__` и/или `Meta.__init__`. PEP 487 (метод `__init_subclass__`) и `__prepare__` влияют на детали создания классов, но это более продвинутые темы.

Миксины (Mixins)

Миксин – это класс, который предоставляет дополнительный функционал через множественное наследование, но сам по себе обычно не предназначен для самостоятельного использования. Идея в том, что миксины «подмешиваются» к основным классам, расширяя их.

Характерные черты миксина: - Миксин **не должен** хранить своё состояние (поля), а скорее дополнять класс методами. Обычно у миксина нет собственного `__init__` (или он очень простой), чтобы не мешать инициализации основного класса.
- Миксин наследует от `object` (или другого базового), но **не переопределяет** критичные методы основного класса.
- Миксины обычно пишут так, чтобы их методы вызывали `super()`, позволяя цепочку MRO объединять несколько миксинов.

Пример миксина для валидации:

```
class ValidateMixin:
    def set_age(self, value):
        if not (0 <= value <= 150):
            raise ValueError("Возраст вне диапазона")
        self.age = value
```

```
class Person(ValidateMixin):
    def __init__(self, age):
        self.age = age
    # наследуем метод set_age из миксина
```

Здесь `ValidateMixin` добавляет метод `set_age`, но не хранит своего состояния. Такой подход позволяет многократно переиспользовать `ValidateMixin` вместе с разными классами.

Миксины часто используют вместе с абстрактными базовыми классами или большим множеством функциональных классов, когда необходима композиция поведения через наследование.

Абстрактные базовые классы (ABC)

Модуль `abc` позволяет определить **абстрактные классы**, которые не могут быть инстанцированы до тех пор, пока не реализованы абстрактные методы. Это делается через `ABCMeta` и декораторы `@abstractmethod`.

Документация `abc` говорит следующее:

> *Модуль* `abc` *предоставляет инфраструктуру для определения абстрактных базовых классов (ABCs).* [41]
> Класс-наследник `ABC` (или прямое использование `metaclass=ABCMeta`) делает класс абстрактным. Методы, помеченные `@abstractmethod`, должны быть реализованы в подклассах, иначе при попытке создать объект будет `TypeError`.

Пример:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r): self.r = r
    def area(self): return 3.14*self.r**2

c = Circle(5)  # Всё хорошо
s = Shape()    # Ошибка TypeError: нельзя инстанцировать абстрактный класс
```

Из документации `abc` видно, что для свойства `@property` тоже предусмотрена поддержка абстрактности: `abstractmethod()` может комбинироваться с другими дескрипторами [42] [43]. Функция `abstractmethod` помечает метод (или `property`) как абстрактный. У абстрактных методов и свойств устанавливается признак `__isabstractmethod__ = True` [44].

После наследования от `ABC` или `ABCMeta` вызывается механизм обновления набора абстрактных методов. Если в дочернем классе все `abstractmethod` реализованы, класс перестаёт быть абстрактным.

Удаление/замена встроенных имен (`builtins`) и их восстановление

Модуль `builtins` в Python содержит все встроенные функции и исключения (например, `len`, `print`, `Exception` и т.д.) [45]. Он автоматически импортируется в каждый модуль (через имя `__builtins__` или напрямую).

Теоретически, можно удалить или заменить любой из этих имен. Например:

```
import builtins
print(len("abc"))  # 3
del builtins.len
# print(len("abc"))  # теперь ошибка: NameError или TypeError
```

Тут `builtins.len` удалён, поэтому глобальный `len` больше не существует (для нового кода). Для восстановления обычно поступают так:

- **Сохранить** исходную функцию перед удалением: `old_len = builtins.len`, а потом `builtins.len = old_len`.
- Или **переустановить модуль** (например, перезапустить интерпретатор или выполнить `import importlib; importlib.reload(builtins)` – хотя стандартного `reload` для builtins нет, можно переимпортировать модуль `builtins` из `sys.modules`).
- Можно также восстанавливать `__builtins__` словарь: в глобальных переменных модулей `__builtins__` обычно указывает на сам модуль `builtins` или на его `__dict__` [46]. Но полагаться на `__builtins__` нежелательно, это внутренняя деталь.

Например, чтобы временно подменить встроенную функцию в одном модуле, можно сделать:

```
import builtins
orig_print = builtins.print
builtins.print = lambda *args, **kwargs: orig_print("DEBUG:", *args,
**kwargs)
print("test")  # выводит "DEBUG: test"
builtins.print = orig_print  # восстановим оригинальную функцию
```

Важно: изменение `builtins` затрагивает **все** модули (во всём интерпретаторе). Будьте очень осторожны: удаление или изменение ключевых функций (например, `len`, `print` и т.д.) может сломать стандартную работу многих функций. Поэтому, если вы поменяли что-то в `builtins`, лучше сразу восстановить или ограничить область видимости (например, локальная переназначенная переменная).

Официальная документация упоминает `builtins` в контексте доступа к глобальным идентификаторам [45], но не содержит рекомендаций по их удалению. Если же вы сталкиваетесь с

«пропавшим» встроенным именем, можно проверить содержимое `builtins.__dict__` или перезагрузить интерпретатор.

Ссылки на документацию

- Официальная документация по объектной модели Python (Data Model) описывает методы `__new__`, `__del__` [1] [3], методы индексирования и деструкторов [16] [18], *super()* [47] [33], контекстные менеджеры [24], дескрипторы [26] и атрибутный доступ [48] [49].
- HOWTO по **Методу разрешения** (MRO) подробно разбирает C3-линеаризацию [37].
- Руководство по `abc` освещает абстрактные классы и декораторы `@abstractmethod` [41] [44].
- PEP 560 описывает добавление `__class_getitem__` и логику обобщений в ядре Python [9] [11].
- Документация функций: описание `type()` для динамических классов [39] и `builtins` для доступа к встроенным идентификаторам [45].

Эти материалы содержат более формальное изложение упомянутых концепций и служат авторитетными источниками по Python-синтаксису и структуре языка.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [23] [35] [37] [38] 3. Data model — Python 3.13.3 documentation
https://docs.python.org/3/reference/datamodel.html

[16] [18] [19] [20] [22] [28] [29] [30] [31] [32] [48] [49] 3. Data model — Python 3.13.3 documentation
http://docs.python.org/reference/datamodel.html

[17] [21] iterator - Why does defining __getitem__ on a class make it iterable in python? - Stack Overflow
https://stackoverflow.com/questions/926574/why-does-defining-getitem-on-a-class-make-it-iterable-in-python

[24] [25] 8. Compound statements — Python 3.13.3 documentation
https://docs.python.org/3/reference/compound_stmts.html

[26] [27] Descriptor Guide — Python 3.13.3 documentation
https://docs.python.org/3/howto/descriptor.html

[33] [34] [36] [39] [40] [47] Built-in Functions — Python 3.13.3 documentation
https://docs.python.org/3/library/functions.html

[41] [42] [43] [44] abc — Abstract Base Classes — Python 3.13.3 documentation
https://docs.python.org/3/library/abc.html

[45] [46] builtins — Built-in objects — Python 3.13.3 documentation
https://docs.python.org/3/library/builtins.html