Специальные методы классов. Механизм работы классов

Цель занятия

После освоения темы вы:

- узнаете понятие и роль специальных методов классов в программировании на Python
- сможете использовать специальные методы в написании программ

План занятия

Конспект занятия

1. Специальные методы классов

Если с помощью функции `dir()` вывести все методы объекта, то в большом списке можно увидеть **специальные методы** — с двойным нижним подчеркиванием. Например:

```
s = 'Amat victoria curam'
dir(s)
```

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',

'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', ...]

У этих методов в литературе есть и другие названия: dunder-методы (от Double UNDERscore) и магические методы.

Специальные методы обеспечивают механизм работы классов в Python. Они вызываются автоматически для определенных действий. Один из таких специальных методов — это `__init__` (конструктор), который используется при построении нового элемента класса.

По наличию или отсутствию специальных методов можно судить о том, какие операции поддерживаются данным классом объектов. Рассмотрим самые популярные.

Метод `__len__` сообщает размер коллекции и используется функцией `len`. Функция `len` работает со встроенными типами напрямую.

Определив метод `__len__` для собственного класса, мы можем «научить» функцию `len` работать с ним так же, как со встроенными типами.

Метод `__repr__` используется, чтобы показать на экране в более осмысленном виде представление объекта для вывода. Обычно он выдает текст, который можно скопировать и вставить в программу. Вывод выглядит так же, как мы написали ввод при создании объекта:

```
class Point:
      def __init__ (self, x, y):
            self.x = x
            self.y = y
      def __repr__(self):
            “”Возвращает строковое представление объекта для
вывода””
            return f’Point({self.x},{self.y})’
>>> p = Point(1, -1)
>>> p
Point(1, -1)
```

Метод `__eq__` нужен для сравнения на равенство экземпляров классов с точки зрения их внутренней структуры:

```
class Point
      …
      def __eq__(self, other):
            return self.x == other.x and self.y == other.y
>>> p = Point(1, -1)
>>> p = Point(1, -1)
True
```

Нужно внимательно подходить к написанию специальных методов и понимать, что корректность их работы — это ответственность программиста. Можно написать метод `__eq__`, который будет всегда возвращать `True`, и все объекты будут считаться равными друг другу, что вызовет немало проблем в дальнейшей работе с кодом. Иногда для сравнения на равенство можно использовать не все атрибуты.

Важно! У любого класса обычно стоит реализовать как минимум три метода:

1. `__init__` (конструктор)
2. `__repr__` (отображение в текстовом виде)
3. `__eq__` (проверка на равенство)

2. Хеширование

Словари и множества работают как хеш-таблицы. Чтобы работать с объектом из хеш-таблицы, нужно его хешировать.

Давайте рассмотрим, с какими проблемами мы можем столкнуться. В данном примере мы не реализуем метод `__eq__` для класса и можем использовать такой объект в качестве ключа в словаре, но потом не сможем его найти:

```
class Point:
      …
      # Метод __eq__ не реализован
points = {Point(1, 1):10}
>>> points[Point(1, 1)]
KeyError: Point(1, 1)
>>> print('N{CONFUSED FACE}')
```

Без метода `__eq__` Python будет считать каждый объект этого класса уникальным. В случае если мы реализуем метод `__eq__`, то мы не сможем использовать объект в качестве ключа. Python сообщит нам, что объект не хеширован:

```
class Point:
      …
      # Метод __eq__ реализован
>>> points = {Point(1, 1):10}
TypeError: unhashable type; 'Point'
>>> print('N{SHOCKED FACE WITH EXPLODING HEAD}')
```

С решением этой ситуации нам поможет хеш-таблица. Принцип ее работы проиллюстрирован на рисунке 1.

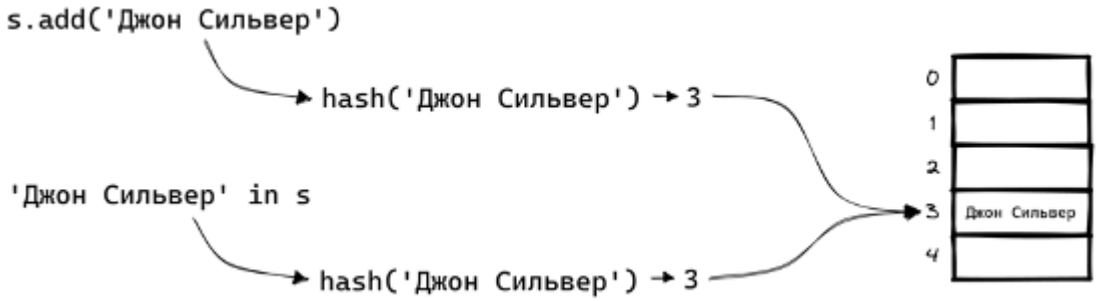

Рисунок 1. Хеш-таблица

Хеш-таблица представляет собой массив бóльшего размера, чем размер данных в нем. При добавлении элемента в хеш-таблицу Python вычисляет хеш-функцию этого элемента – хеш-функция может взять большой объект и превратить его в число. Это число будет давать нам индекс той ячейки, в которой находится элемент. Таким образом, преимущество хеш-таблицы — очень быстрый поиск за счет системы ячеек.

Метод `__hash__` нужно реализовать, чтобы использовать объект в хэш-таблицах (совместно с методом `__eq__`).

```
class Point
      …
      def __eq__(self, other):
            return self.x == other.x and self.y == other.y
      def __hash__(self):
```

```
        # ОПАСНО!
            return hash((self.x, other.y))
>>> p = Point(1, 1)
>>> points = {p: 10}
>>> p.x += 1
>>> points
{Point(2, 1): 10}

>>> points[p]
KeyError: Point(2, 1)
```

После того как мы использовали объект в качестве ключа, он может измениться. Словарь пытается найти данные там, где они больше не лежат. Чтобы правильно реализовать возможность хеширования, нужно гарантировать, что объект не будет изменяться. Сделать это сложно, потому что нет специального модификатора. Но мы можем использовать другой метод — метод `__setattr__`.

Метод `__setattr__` позволяет переопределить поведение объекта при записи в него какого-либо атрибута:

```
class Point:
    __frozen = False
    def __init__ (self, x, y):
        self.x = x
        self.y = y
        self.__frozen = True
    …
    def __setattr__(self, attr, value):
        if not self.__frozen:
            super().__setattr__(attr, value)
        else:
        return TypeError('Point object is immutable')

>>> p = Point(1, 1)
```

```
>>> p.x += 1
TypeError: Point object is immutable
```

В примере выше нам удалось сохранить хеширование объекта. Но сделать объект по-настоящему неизменяемым практически невозможно, но это и не нужно. Мы использовали «заморозку» только для того, чтобы уберечь код от ошибок и случайных действий.

3. Специальные атрибуты

Помимо специальных методов существуют специальные атрибуты, которые вызываются автоматически:

- **`__class__`** — ссылка на класс, экземпляром которого является данный объект;
- **`__dict__`** хранит все атрибуты объекта в виде словаря;
- **`__annotations__`** — словарь, содержащий аннотации типов, описанные в классе;
- **`__doc__`** — документ-строка, описанная в начале класса (именно ее выдает функция `help`). Этот атрибут выдает документацию.

4. Перегрузка операторов

Перегрузка оператора — это реализация оператором разной логики в зависимости от типов операндов, к которым он применяется.

Решим задачу: определить арифметические операторы для нашего класса.

```
class Vector:
      x: float
      y: float
      # ?
>>> a = Vector(1, 3)
>>> b = Vector(2, 1)
>>> c = a + 2*b # Мы хотим, чтобы это работало!
>>> c
Vector (5, 5)
```

Чтобы сложение сработало, нужно определить **метод `__add__`**:

```
class Vector:
```

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

```
>>> a = Vector(1, 3)
>>> b = Vector(2, 1)
>>> a + b
Vector (3, 4)
```

Если `other` — не `Vector`, то на выводе мы получим ошибку о том, что число не содержит атрибута `x`:

```
>>> a = Vector(1, 3)
>>> a + 2
AttributeError: 'int' object has no attribute 'x'
```

Чтобы получить больше информации в сообщении ошибки, нужно для начала проверить является ли ваше число вектором с помощью `isinstance()`. После этого ошибка станет более информативной.

Для умножения используется **метод `__mul__`**. Он может быть реализован сложнее — например, вектор можно умножить на число.

Что нужно учитывать при написании метода умножение:

- от перестановки мест множителей может поменяться логика;
- Python пытается найти в классе `int` метод умножения на вектор, если в записи умножения число стоит первым.

Для решения этих проблем можно использовать модификатор метода (подставить вперед `r`) — **`__rmul__`**:

```
class Vector:
    …
    def __rmul__(self, other):
        return self * other
```

```
>>> a = Vector (1, 3)
>>> a * 2
```

```
Vector (2, 6)
```

Переставим местами аргументы:

```
>>> 2 * a
Vector (2, 6)
```

Арифметические операторы

Метод	Действие
`__neg__`	`- a`
`__add__`	`a + b`
`__sub__`	`a - b`
`__mul__ (__rmul__`	`a * b`
`__truediv__`	`a /`
`__floordiv__`	`a // b`
`__mod__`	`a % b`
`__pow__`	`a ** b`
`__matmul__`	`a @ b (матричное произведение)`

Битовые операции

Метод	Действие
`__invert__`	`~ a`
`__and__`	`a & b`
`__or__`	`a | b`
`__xor__`	`a ^ b`
`__rshift__`	`a >> b`
`__lshift__`	`a << b`

У всех методов есть модификация с буквой `i` (`inplace`):

`__iadd__` a += b

и с буквой `r` (`right`) — обратный порядок операндов:

`__radd__` b + a

5. Коллекции и итераторы

Все коллекции в Python тоже имеют определенный интерфейс, который описывается специальными методами. Чтобы создать собственный тип коллекций, нужно реализовать ряд специальных методов.

Давайте создадим свой класс, описывающий массив битов. Для хранения данных будем использовать обычный `int`, а получать нужные элементы будем с помощью битовых операций:

```
class Bitarray:
      def __init__(self, length=0, init_value=0):
            self._value = init_value
            self._length = length
      def __repr__(self):
            # Используем len(self), поскольку есть реализованный
            # метод __len__
            return f'Bitarray({self._value:0{len(self)}b})'
      def __eq__(self, other):
            # сначала выполняем более дешевую проверку len
            return len(self) == len(other) and self._value ==
            other._value
      def __len__(self):
            return self._length
```

```
>>> a = Bitarray(32)
>>> a
Bitarray (00000000000000000000000000000000)

>>> len(a)
32
```

Метод `__getitem__` позволяет обратиться к определенному элементу с помощью квадратных скобок:

```
class Bitarray:
    …
    def __getitem__(self, index):
        # Необходимо проверить выход за допустимые пределы
        if index >= len(self) or index < -len(self):
            raise IndexError('Index out of range')
        index %= len(self)
        return (self._value & (1 << index)) >> index

a = Bitarray(4, 0b1100)
for i in range(4):
    print(i, a[i])
```

Написанного метода уже достаточно, чтобы использовать созданный объект в цикле напрямую:

```
a = Bitarray(4, 0b1100)
for x in a:
    print(x, end=' ')
    # 0 0 1 1
```

С помощью `__getitem__` мы также можем превратить наш `Bitarray` в любую другую коллекцию:

```
>>> a = Bitarray(4, 0b1100)
>>> list(a)
[0, 0, 1, 1]

>>> tuple(a)
(0, 0, 1, 1)

>>> set(a)
```

```
{0, 1}
```

Или можем использовать распаковку:

```
>>> print(*a)
1 1 0 0

>>> first, &others = a
>>> first
1

>>> first
[1, 0, 0]
```

Метод `__setitem__` позволяет реализовать изменения в нашей коллекции. Метод принимает еще два аргумента помимо `self` — `index` и `value` (назвать можно и по-другому).

Работает следующим образом:

```
# Мы вызываем индексом элемент и записываем в него свое значение
>>> a = Bitarray(10)
>>> a
Bitarray(0000000000)

>>> a[5] = 1
>>> a
Bitarray(0000100000)
```

Как обратиться к элементу в цикле?

Метод `__iter__` — это специальный метод, который возвращает особый объект-итератор для использования в цикле `for`. Если метод `__iter__` не реализован, но есть `__getitem__`, то цикл будет перебирать индексы от 0 и далее, пока не получит `IndexError`.

Помимо этого метода мы можем создать и собственный **итератор** — объект, который реализует метод `__next__`. Цикл `for` вызывает этот метод, чтобы получить

следующий элемент, пока не получит специальное исключение `StopIteration`. Реализация собственного итератора не всегда имеет смысл, но может пригодиться, если все элементы коллекции не хранятся в памяти, а каким-либо образом вычисляются.

Иногда итератор реализуют в виде отдельного класса. Получить итератор можно с помощью функции `iter`:

```
>>> range(10)
range(0, 10)

>>> iter(range(10))
<range_iterator object at 0x7fc4ac3c8ff0>
```

6. Контекстные менеджеры

Вы уже знакомы с контекстным менеджером на примере работы с файлом:

```
with open('filename') as file:
    …
    # Работаем с файлом
    …
# Файл закрыт автоматически
```

Контекстный менеджер — это объект, который можно подставить в блок `with`. Его суть в том, что мы можем выполнить некоторое действие ПЕРЕД и некоторое действие ПОСЛЕ определенного блока кода.

Контекстный менеджер – это объект, который реализует методы **`__enter__`** и **`__exit__`**.

В качестве примера рассмотрим контекстный менеджер, который замеряет время выполнения модуля для понимания производительности кода:

```
from time import perf_counter

class timeit:
    def __init__ (self, process_name):
        self.name = process_name

    def __enter__ (self):
        self.start = perf_counter()
```

```
            return self

      def __exit__ (self, exp_type, exc_val, exc_tb):
            if exc_type is None: # Блок кода завершился без ошибок
                  print(f'{self.name}: {perf_counter() -
                  self.start:0.3f} sec')

with timeit('Возведение в квадрат миллиона чисел'):
      # вызван метод __enter__
      for i in range(1000000):
            i = i ** 2
      # вызван метод __exit__
```

```
Возведение в квадрат миллиона чисел: 0.459 сек.
```

С помощью этой логики вы можете реализовывать собственные контекстные менеджеры.

7. Callable-объекты и декораторы

Реализация **метода `__call__`** позволяет сделать объект «вызываемым» (callable).

К примеру, можно создать класс `Function`:

```
class Function:
      …
      def __call__ (self):
            print('Hello from Function object!')
>>> f = Function()
>>> f()
Hello from Function object!
```

Для чего это может пригодиться? Один из сценариев использования — создание «фабрики функций».

Другой более распространенный метод для Python — это создание **декораторов** (мы изучали их на прошлой неделе). Декоратор используют для того, чтобы расширить функциональность какой-либо функции:

```
@decorator
```

```
def function(*args):
    …
# Данная запись эквивалентна следующей:
function = decorator(function)
```

Давайте рассмотрим на более сложном примере. Используем декоратор для того, чтобы добавить функции способность посчитать, сколько раз она была вызвана:

```
class call_count:
    def __init__ (self, function):
        self.function = function
        self.count = 0

    def __call__ (self, *args, **kwargs):
        self.count += 1
        return self.function(*args, **kwargs)

@call_count
def fibonacci(n):
    if n > 2: return n
    return fibonacci (n-1) + fibonacci (n-2)

fibonacci(30)
print(fibonacci.count)
```

```
2692537
```

Делаем вывод, что такая реализация невыгодна, поскольку функция пересчитывает те же самые значения несколько раз. Общее число операций получается больше, чем само число Фибоначчи.

Еще один полезный декоратор из стандартной библиотеки — **`property`**. Он позволяет создавать динамические атрибуты.

Рассмотрим другой способ его применения:

```
class PositiveInt:
```

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

    @property
    def value(self):
        return self.value

    @value.setter
    def value(self, new_value):
        if new_value < 0:
            raise ValueError('value of PosotiveInt must be
            positive')
        self._value = new_value
```

```
>>> x = PositiveInt(1)
>>> x.value = 2
>>> x.value
2

>>> x.value = -1
ValueError: value of PositiveInt must be positive
```

Декоратор `property` позволяет описать произвольную логику при получении атрибута объекта и при установке этого атрибута. Метод `value` начинает вести себя как атрибут здесь.

Еще один довольно полезный декоратор содержится в модуле **`functools.cache`** — мемоизация (запоминание тех значений, которые функция уже посчитала). Это нужно, чтобы функция не считала значения заново, а брала данные из кеша. Так мы сможем упростить код для вычисления чисел Фибоначчи:

```
from functools import cache
class call_count:
    …
@call_count
```

```
@cache
def fibonacci(n):
      if n < 2: return n
      return fibonacci(n-1) + fibonacci(n-2)

fibonacci(30)
print(fibonacci.count)

59
```

Вычисление тридцатого числа Фибоначчи потребует от функции только 59 вызовов.

8. Построитель классов данных `dataclass`

Метапрограммирование — это создание программных средств, позволяющих управлять кодом программы.

Декоратор — одно из средств метапрограммирования, поскольку он позволяет «на ходу» подменить функции объекта другими свойствами. Декораторы могут применяться не только к функции, но и к классу.

Давайте рассмотрим один популярный декоратор — **`dataclass`**. Его идея в том, чтобы уменьшить количество одного и того же кода. Он автоматически генерирует стандартные методы `__init__`, `__repr__`, `__eq__`, а также ряд других на основе аннотаций типов. Если мы знаем, какие атрибуты есть у класса, нам этого достаточно, чтобы описать их.

Использование `dataclass` может выглядеть следующим образом:

```
from dataclasses import dataclass

@dataclass
class Team:
      id: int
      name: str
      score: int = 0

>>> my_team = Team(1, 'The Tetris Masters')
>>> my_team
Team(id=1, name='The Tetris Masters', score=0)
```

```
>>> my_team == Team(1, 'The Tetris Masters')
True
```

Чтобы написать декоратор, мы создаем класс, описываем только аннотации типов с атрибутами. Некоторым атрибутам можем назначить значения по умолчанию. Остальное декоратор сделает сам.

Также декоратор можно настраивать с помощью дополнительных параметров.

Рассмотрим, какие параметры по умолчанию установлены у декоратора `dataclass`:

- `init (True)` — сгенерировать метод `__init__`;
- `repr (True)` — сгенерировать метод `__repr__`;
- `eq (True)` — сгенерировать метод `__eq__м`;
- `order (False)` — сгенерировать методы `__lt__`, `__le__`, `__gt__`, `__ge__` (объекты будут сравниваться как кортежи);
- `unsafe_hash (False)` — сгенерировать метод `__hash__`, даже если это опасно (не рекомендуется);
- `frozen (False)` — сделать объекты неизменяемыми и сгенерировать метод `__hash__` (теперь это безопасно);
- `kw_only (False)` — аргументы в конструктор можно передавать только по имени;
- `slots (False)` — создает атрибут `__slots__` (позволяет хранить объекты более компактно). Пример применения атрибута `__slots__`:

```
class Person:
__slots__ = ('first_name', 'last_name')
first_name: str
last_name: str
```