**Тема занятия №17: Объектно-ориентированное программирование**

**1. Введение**

Что такое магические методы? Они всё в объектно-ориентированном Питоне. Это специальные методы, с помощью которых вы можете добавить в ваши классы «магию». Они всегда обрамлены двумя нижними подчеркиваниями (например, \_\_init\_\_ или \_\_lt\_\_). Ещё, они не так хорошо документированы, как хотелось бы. Все магические методы описаны в документации, но весьма беспорядочно и почти безо всякой организации.

**2. Конструирование и инициализация**

Всем известен самый базовый магический метод, \_\_init\_\_. С его помощью мы можем инициализировать объект. Однако, когда я пишу x = SomeClass(), \_\_init\_\_ не самое первое, что вызывается. На самом деле, экземпляр объекта создаёт метод \_\_new\_\_, а затем аргументы передаются в инициализатор. На другом конце жизненного цикла объекта находится метод \_\_del\_\_. Давайте подробнее рассмотрим эти три магических метода:

\_\_new\_\_(cls, [...)

Это первый метод, который будет вызван при инициализации объекта. Он принимает в качестве параметров класс и потом любые другие аргументы, которые будут переданы в \_\_init\_\_. \_\_new\_\_ используется весьма редко, но иногда бывает полезен, в частности, когда класс наследуется от неизменяемого (immutable) типа, такого как кортеж (tuple) или строка. Я не намерен очень детально останавливаться на \_\_new\_\_, так как он не то чтобы очень часто нужен, но этот метод очень хорошо и детально описан в документации.

\_\_init\_\_(self, [...)

Инициализатор класса. Ему передаётся всё, с чем был вызван первоначальный конструктор (так, например, если мы вызываем x = SomeClass(10, 'foo'), \_\_init\_\_ получит 10 и 'foo' в качестве аргументов. \_\_init\_\_ почти повсеместно используется при определении классов.

\_\_del\_\_(self)

Если \_\_new\_\_ и \_\_init\_\_ образуют конструктор объекта, \_\_del\_\_ это его деструктор. Он не определяет поведение для выражения del x (поэтому этот код не эквивалентен x.\_\_del\_\_()). Скорее, он определяет поведение объекта в то время, когда объект попадает в сборщик мусора. Это может быть довольно удобно для объектов, которые могут требовать дополнительных чисток во время удаления, таких как сокеты или файловыве объекты. Однако, нужно быть осторожным, так как нет гарантии, что \_\_del\_\_ будет вызван, если объект продолжает жить, когда интерпретатор завершает работу. Поэтому \_\_del\_\_ не может служить заменой для хороших программистских практик (всегда завершать соединение, если закончил с ним работать и тому подобное). Фактически, из-за отсутствия гарантии вызова, \_\_del\_\_ не должен использоваться почти никогда; используйте его с осторожностью!

Замечание от переводчика: svetlov отмечает, что здесь автор ошибается, на самом деле \_\_del\_\_ всегда вызывается по завершении работы интерпретатора.

Соединим всё вместе, вот пример \_\_init\_\_ и \_\_del\_\_ в действии:

**from os.path import join**

**class FileObject:**

**'''Обёртка для файлового объекта, чтобы быть уверенным в том, что файл будет закрыт при удалении.'''**

**def \_\_init\_\_(self, filepath='~', filename='sample.txt'):**

**# открыть файл filename в filepath в режиме чтения и записи**

**self.file = open(join(filepath, filename), 'r+')**

**def \_\_del\_\_(self):**

**self.file.close()**

**del self.file**

**3. Переопределение операторов на произвольных классах**

Одно из больших преимуществ использования магических методов в Питоне то, что они предоставляют простой способ заставить объекты вести себя по подобию встроенных типов. Это означает, что вы можете избежать унылого, нелогичного и нестандартного поведения базовых операторов. В некоторых языках обычное явление писать как-нибудь так:

**if instance.equals(other\_instance):**

**# do something**

Вы, конечно, можете поступать так же и в Питоне, но это добавляет путаницы и ненужной многословности. Разные библиотеки могут по-разному называть одни и те же операции, заставляя использующего их программиста совершать больше действий, чем необходимо. Используя силу магических методов, мы можем определить нужный метод (\_\_eq\_\_, в этом случае), и так точно выразить, что мы имели в виду:

**if instance == other\_instance:**

**#do something**

Это одна из сильных сторон магических методов. Подавляющее большинство из них позволяют определить, что будут делать стандартные операторы, так что мы можем использовать операторы на своих классах так, как будто они встроенные типы.

**4. Магические методы сравнения**

В Питоне много магических методов, созданных для определения интуитивного сравнения между объектами используя операторы, а не неуклюжие методы. Кроме того, они предоставляют способ переопределить поведение Питона по умолчанию для сравнения объектов (по ссылке). Вот список этих методов и что они делают:

\_\_cmp\_\_(self, other)

Самый базовый из методов сравнения. Он, в действительности, определяет поведение для всех операторов сравнения (>, ==, !=, итд.), но не всегда так, как вам это нужно (например, если эквивалентность двух экземпляров определяется по одному критерию, а то что один больше другого по какому-нибудь другому). \_\_cmp\_\_ должен вернуть отрицательное число, если self < other, ноль, если self == other, и положительное число в случае self > other. Но, обычно, лучше определить каждое сравнение, которое вам нужно, чем определять их всех в \_\_cmp\_\_. Но \_\_cmp\_\_ может быть хорошим способом избежать повторений и увеличить ясность, когда все необходимые сравнения оперируют одним критерием.

\_\_eq\_\_(self, other)

Определяет поведение оператора равенства, ==.

\_\_ne\_\_(self, other)

Определяет поведение оператора неравенства, !=.

\_\_lt\_\_(self, other)

Определяет поведение оператора меньше, <.

\_\_gt\_\_(self, other)

Определяет поведение оператора больше, >.

\_\_le\_\_(self, other)

Определяет поведение оператора меньше или равно, <=.

\_\_ge\_\_(self, other)

Определяет поведение оператора больше или равно, >=.

Для примера рассмотрим класс, описывающий слово. Мы можем сравнивать слова лексиграфически (по алфавиту), что является дефолтным поведением при сравнении строк, но можем захотеть использовать при сравнении какой-нибудь другой критерий, такой, как длина или количество слогов. В этом примере мы будем сравнивать по длине. Вот реализация:

**class Word(str):**

**'''Класс для слов, определяющий сравнение по длине слов.'''**

**def \_\_new\_\_(cls, word):**

**# Мы должны использовать \_\_new\_\_, так как тип str неизменяемый**

**# и мы должны инициализировать его раньше (при создании)**

**if ' ' in word:**

**print "Value contains spaces. Truncating to first space."**

**word = word[:word.index(' ')] # Теперь Word это все символы до первого пробела**

**return str.\_\_new\_\_(cls, word)**

**def \_\_gt\_\_(self, other):**

**return len(self) > len(other)**

**def \_\_lt\_\_(self, other):**

**return len(self) < len(other)**

**def \_\_ge\_\_(self, other):**

**return len(self) >= len(other)**

**def \_\_le\_\_(self, other):**

**return len(self) <= len(other)**

Теперь мы можем создать два Word (при помощи Word('foo') и Word('bar')) и сравнить их по длине. Заметьте, что мы не определяли \_\_eq\_\_ и \_\_ne\_\_, так как это приведёт к странному поведению (например, Word('foo') == Word('bar') будет расцениваться как истина). В этом нет смысла при тестировании на эквивалентность, основанную на длине, поэтому мы оставляем стандартную проверку на эквивалентность от str.

Сейчас, кажется, удачное время упомянуть, что вы не должны определять каждый из магических методов сравнения, чтобы полностью охватить все сравнения. Стандартная библиотека любезно предоставляет нам класс-декторатор в модуле functools, который и определит все сравнивающие методы, от вас достаточно определить только \_\_eq\_\_ и ещё один (\_\_gt\_\_, \_\_lt\_\_ и т.п.) Эта возможность доступна начиная с 2.7 версии Питона, но если это вас устраивает, вы сэкономите кучу времени и усилий. Для того, чтобы задействовать её, поместите @total\_ordering над вашим определением класса.

**5. Числовые магические методы**

Точно так же, как вы можете определить, каким образом ваши объекты будут сравниваться операторами сравнения, вы можете определить их поведение для числовых операторов.

Унарные операторы и функции

Унарные операторы и функции имеют только один операнд — отрицание, абсолютное значение, и так далее.

\_\_pos\_\_(self)

Определяет поведение для унарного плюса (+some\_object)

\_\_neg\_\_(self)

Определяет поведение для отрицания(-some\_object)

\_\_abs\_\_(self)

Определяет поведение для встроенной функции abs().

\_\_invert\_\_(self)

Определяет поведение для инвертирования оператором ~. Для объяснения что он делает смотри статью в Википедии о бинарных операторах.

\_\_round\_\_(self, n)

Определяет поведение для встроенной функции round(). n это число знаков после запятой, до которого округлить.

\_\_floor\_\_(self)

Определяет поведение для math.floor(), то есть, округления до ближайшего меньшего целого.

\_\_ceil\_\_(self)

Определяет поведение для math.ceil(), то есть, округления до ближайшего большего целого.

\_\_trunc\_\_(self)

Определяет поведение для math.trunc(), то есть, обрезания до целого.

Обычные арифметические операторы

Теперь рассмотрим обычные бинарные операторы (и ещё пару функций): +, -, \* и похожие. Они, по большей части, отлично сами себя описывают.

\_\_add\_\_(self, other)

Сложение.

\_\_sub\_\_(self, other)

Вычитание.

\_\_mul\_\_(self, other)

Умножение.

\_\_floordiv\_\_(self, other)

Целочисленное деление, оператор //.

\_\_div\_\_(self, other)

Деление, оператор /.

\_\_truediv\_\_(self, other)

Правильное деление. Заметьте, что это работает только когда используется from \_\_future\_\_ import division.

\_\_mod\_\_(self, other)

Остаток от деления, оператор %.

\_\_divmod\_\_(self, other)

Определяет поведение для встроенной функции divmod().

\_\_pow\_\_

Возведение в степень, оператор \*\*.

\_\_lshift\_\_(self, other)

Двоичный сдвиг влево, оператор <<.

\_\_rshift\_\_(self, other)

Двоичный сдвиг вправо, оператор >>.

\_\_and\_\_(self, other)

Двоичное И, оператор &.

\_\_or\_\_(self, other)

Двоичное ИЛИ, оператор |.

\_\_xor\_\_(self, other)

Двоичный xor, оператор ^.

Отражённые арифметические операторы

some\_object + other

Это «обычное» сложение. Единственное, чем отличается эквивалентное отражённое выражение, это порядок слагаемых:

other + some\_object

Таким образом, все эти магические методы делают то же самое, что и их обычные версии, за исключением выполнения операции с other в качестве первого операнда и self в качестве второго. В большинстве случаев, результат отражённой операции такой же, как её обычный эквивалент, поэтому при определении \_\_radd\_\_ вы можете ограничиться вызовом \_\_add\_\_ да и всё. Заметьте, что объект слева от оператора (other в примере) не должен иметь обычной неотражённой версии этого метода. В нашем примере, some\_object.\_\_radd\_\_ будет вызван только если в other не определён \_\_add\_\_.

\_\_radd\_\_(self, other)

Отражённое сложение.

\_\_rsub\_\_(self, other)

Отражённое вычитание.

\_\_rmul\_\_(self, other)

Отражённое умножение.

\_\_rfloordiv\_\_(self, other)

Отражённое целочисленное деление, оператор //.

\_\_rdiv\_\_(self, other)

Отражённое деление, оператор /.

\_\_rtruediv\_\_(self, other)

Отражённое правильное деление. Заметьте, что работает только когда используется from \_\_future\_\_ import division.

\_\_rmod\_\_(self, other)

Отражённый остаток от деления, оператор %.

\_\_rdivmod\_\_(self, other)

Определяет поведение для встроенной функции divmod(), когда вызывается divmod(other, self).

\_\_rpow\_\_

Отражённое возведение в степень, оператор \*\*.

\_\_rlshift\_\_(self, other)

Отражённый двоичный сдвиг влево, оператор <<.

\_\_rrshift\_\_(self, other)

Отражённый двоичный сдвиг вправо, оператор >>.

\_\_rand\_\_(self, other)

Отражённое двоичное И, оператор &.

\_\_ror\_\_(self, other)

Отражённое двоичное ИЛИ, оператор |.

\_\_rxor\_\_(self, other)

Отражённый двоичный xor, оператор ^.

Составное присваивание

В Питоне широко представлены и магические методы для составного присваивания. Вы скорее всего уже знакомы с составным присваиванием, это комбинация «обычного» оператора и присваивания. Если всё ещё непонятно, вот пример:

x = 5

x += 1 # другими словами x = x + 1

Каждый из этих методов должен возвращать значение, которое будет присвоено переменной слева (например, для a += b, \_\_iadd\_\_ должен вернуть a + b, что будет присвоено a). Вот список:

\_\_iadd\_\_(self, other)

Сложение с присваиванием.

\_\_isub\_\_(self, other)

Вычитание с присваиванием.

\_\_imul\_\_(self, other)

Умножение с присваиванием.

\_\_ifloordiv\_\_(self, other)

Целочисленное деление с присваиванием, оператор //=.

\_\_idiv\_\_(self, other)

Деление с присваиванием, оператор /=.

\_\_itruediv\_\_(self, other)

Правильное деление с присваиванием. Заметьте, что работает только если используется from \_\_future\_\_ import division.

\_\_imod\_(self, other)

Остаток от деления с присваиванием, оператор %=.

\_\_ipow\_\_

Возведение в степерь с присваиванием, оператор \*\*=.

\_\_ilshift\_\_(self, other)

Двоичный сдвиг влево с присваиванием, оператор <<=.

\_\_irshift\_\_(self, other)

Двоичный сдвиг вправо с присваиванием, оператор >>=.

\_\_iand\_\_(self, other)

Двоичное И с присваиванием, оператор &=.

\_\_ior\_\_(self, other)

Двоичное ИЛИ с присваиванием, оператор |=.

\_\_ixor\_\_(self, other)

Двоичный xor с присваиванием, оператор ^=.

**6. Магические методы преобразования типов**

Кроме того, в Питоне множество магических методов, предназначенных для определения поведения для встроенных функций преобразования типов, таких как float(). Вот они все:

\_\_int\_\_(self)

Преобразование типа в int.

\_\_long\_\_(self)

Преобразование типа в long.

\_\_float\_\_(self)

Преобразование типа в float.

\_\_complex\_\_(self)

Преобразование типа в комплексное число.

\_\_oct\_\_(self)

Преобразование типа в восьмеричное число.

\_\_hex\_\_(self)

Преобразование типа в шестнадцатеричное число.

\_\_index\_\_(self)

Преобразование типа к int, когда объект используется в срезах (выражения вида [start:stop:step]). Если вы определяете свой числовой тип, который может использоваться как индекс списка, вы должны определить \_\_index\_\_.

\_\_trunc\_\_(self)

Вызывается при math.trunc(self). Должен вернуть своё значение, обрезанное до целочисленного типа (обычно long).

\_\_coerce\_\_(self, other)

Метод для реализации арифметики с операндами разных типов. \_\_coerce\_\_ должен вернуть None если преобразование типов невозможно. Если преобразование возможно, он должен вернуть пару (кортеж из 2-х элементов) из self и other, преобразованные к одному типу.

**7. Представление своих классов**

Часто бывает полезно представление класса в виде строки. В Питоне существует несколько методов, которые вы можете определить для настройки поведения встроенных функций при представлении вашего класса.

\_\_str\_\_(self)

Определяет поведение функции str(), вызванной для экземпляра вашего класса.

\_\_repr\_\_(self)

Определяет поведение функции repr(), вызванной для экземпляра вашего класса. Главное отличие от str() в целевой аудитории. repr() больше предназначен для машинно-ориентированного вывода (более того, это часто должен быть валидный код на Питоне), а str() предназначен для чтения людьми.

\_\_unicode\_\_(self)

Определяет поведение функции unicode(), вызванной для экземпляра вашего класса. unicode() похож на str(), но возвращает строку в юникоде. Будте осторожны: если клиент вызывает str() на экземпляре вашего класса, а вы определили только \_\_unicode\_\_(), то это не будет работать. Постарайтесь всегда определять \_\_str\_\_() для случая, когда кто-то не имеет такой роскоши как юникод.

\_\_format\_\_(self, formatstr)

Определяет поведение, когда экземпляр вашего класса используется в форматировании строк нового стиля. Например, "Hello, {0:abc}!".format(a) приведёт к вызову a.\_\_format\_\_("abc"). Это может быть полезно для определения ваших собственных числовых или строковых типов, которым вы можете захотеть предоставить какие-нибудь специальные опции форматирования.

\_\_hash\_\_(self)

Определяет поведение функции hash(), вызванной для экземпляра вашего класса. Метод должен возвращать целочисленное значение, которое будет использоваться для быстрого сравнения ключей в словарях. Заметьте, что в таком случае обычно нужно определять и \_\_eq\_\_ тоже. Руководствуйтесь следующим правилом: a == b подразумевает hash(a) == hash(b).

\_\_nonzero\_\_(self)

Определяет поведение функции bool(), вызванной для экземпляра вашего класса. Должна вернуть True или False, в зависимости от того, когда вы считаете экземпляр соответствующим True или False.

\_\_dir\_\_(self)

Определяет поведение функции dir(), вызванной на экземпляре вашего класса. Этот метод должен возвращать пользователю список атрибутов. Обычно, определение \_\_dir\_\_ не требуется, но может быть жизненно важно для интерактивного использования вашего класса, если вы переопределили \_\_getattr\_\_ или \_\_getattribute\_\_ (с которыми вы встретитесь в следующей части), или каким-либо другим образом динамически создаёте атрибуты.

\_\_sizeof\_\_(self)

Определяет поведение функции sys.getsizeof(), вызванной на экземпляре вашего класса. Метод должен вернуть размер вашего объекта в байтах.

Согласно модели данных Python, язык предлагает три вида методов: статические, класса и экземпляра класса. Давайте посмотрим, что же происходит за кулисами каждого из видов методов. Понимание принципов их работы поможет в создании красивого и эффективного кода. Начнём с самого простого примера, в котором демонстрируются все три вида методов.

**class ToyClass:**

**def instancemethod(self):**

**return 'instance method called', self**

**@classmethod**

**def classmethod(cls):**

**return 'class method called', cls**

**@staticmethod**

**def staticmethod():**

**return 'static method called'**

Ранее было сказано, с определенным допущением классы можно рассматривать как модули, содержащие переменные со значениями и функции. Только здесь переменные называются полями или свойствами, а функции – методами. Вместе поля и методы называются атрибутами.

Однако в случае классов, когда метод применяется к объекту, этот экземпляр передается в метод в качестве первого аргумента:

**>>> class A:**

**... def meth(self):**

**... print('meth')**

**...**

**>>> a = A()**

**>>> a.meth()**

**meth**

**>>> A.meth(a)**

**meth**

Вызов a.meth() на самом деле преобразуется к A.meth(a), то есть мы идем к "модулю A" и в его пространстве имен ищем атрибут meth. Там оказывается, что meth это функция, принимающая один обязательный аргумент. Тогда ничего не мешает сделать так:

**>>> b = 10**

**>>> A.meth(b)**

**meth**

В таком "модульном формате" вызова методов передавать объект-экземпляр именно класса A совсем не обязательно. Однако нельзя сделать так:

**>>> b = 10**

**>>> b.meth()**

**Traceback (most recent call last):**

**File "<stdin>", line 1, in <module>**

**AttributeError: 'int' object**

**has no attribute 'meth'**

Если объект передается методу в нотации через точку, то этот метод должен быть описан в том классе, которому принадлежит объект, или в родительских классах. В данном случае у класса int нет метода meth(). Объект b классу A не принадлежит. Поэтому интерпретатор никогда не найдет метод meth().

Что делать, если возникает необходимость в методе, который не принимал бы объект данного класса в качестве аргумента? Да, мы можем объявить метод вообще без параметров и вызывать его только через класс:

**>>> class A:**

**... def meth():**

**... print('meth')**

**...**

**>>> A.meth()**

**meth**

**>>> a = A()**

**>>> a.meth()**

**Traceback (most recent call last):**

**File "<stdin>", line 1, in <module>**

**TypeError: meth() takes 0 positional**

**arguments but 1 was given**

Получается странная ситуация. Ведь meth() вызывается не только через класса, но и через порожденные от него объекты. Однако в последнем случае всегда будет возникать ошибка. То есть имеется потенциально ошибочный код. Кроме того, может понадобиться метод с параметрами, но которому не надо передавать экземпляр данного класса.

В ряде языков программирования, например в Java, для таких ситуаций предназначены статические методы. При описании этих методов в их заголовке ставится ключевое слово static. Такие методы могут вызываться через объекты данного класса, но сам объект в качестве аргумента в них не передается.

В Python острой необходимости в статических методах нет, так как код может находиться за пределами класса, и программа не начинает выполняться из класса. Если нам нужна просто какая-нибудь функция, мы можем определить ее в основной ветке. В Java это не так. Там, не считая импортов, весь код находится внутри классов. Поэтому методы, не принимающие объект данного класса и играющие роль обычных функций, необходимы. Статические методы решают эту проблему.

Однако в Python тоже можно реализовать подобное, то есть статические методы, с помощью декоратора @staticmethod:

**>>> class A:**

**... @staticmethod**

**... def meth():**

**... print('meth')**

**...**

**>>> a = A()**

**>>> a.meth()**

**meth**

**>>> A.meth()**

**meth**

**Пример с параметром:**

**>>> class A:**

**... @staticmethod**

**... def meth(value):**

**... print(value)**

**...**

**>>> a = A()**

**>>> a.meth(1)**

**1**

**>>> A.meth('hello')**

**Hello**

Методы класса

Методы класса принимают класс в качестве параметра, который принято обозначать как cls. Он указывает на класс ToyClass, а не на объект этого класса. При декларации методов этого вида используется декоратор classmethod.

Методы класса привязаны к самому классу, а не его экземпляру. Они могут менять состояние класса, что отразится на всех объектах этого класса, но не могут менять конкретный объект.

Встроенный пример метода класса — dict.fromkeys()— возвращает новый словарь с переданными элементами в качестве ключей.

**dict.fromkeys('AEIOU') # <- вызывается при помощи класса dict**

**{'A': None, 'E': None, 'I': None, 'O': None, 'U': None}**

Статические методы

Статические методы декларируются при помощи декоратора staticmethod. Им не нужен определённый первый аргумент (ни self, ни cls).

Их можно воспринимать как методы, которые “не знают, к какому классу относятся”.

Таким образом, статические методы прикреплены к классу лишь для удобства и не могут менять состояние ни класса, ни его экземпляра.

Статические методы в Python – по-сути обычные функции, помещенные в класс для удобства и находящиеся в пространстве имен этого класса. Это может быть какой-то вспомогательный код. Вообще, если в теле метода не используется self, то есть ссылка на конкретный объект, следует задуматься, чтобы сделать метод статическим. Если такой метод необходим только для обеспечения внутренних механизмов работы класса, то возможно его не только надо объявить статическим, но и скрыть от доступа из вне.

Пусть у нас будет класс "Цилиндр". При создании объектов от этого класса у них заводятся поля высота и диаметр, а также площадь поверхности. Вычисление площади можно поместить в отдельную статическую функцию. Она вроде и относится к цилиндрам, но, с другой стороны, само вычисление объекта не требует и может быть использовано где угодно.

**from math import pi**

**class Cylinder:**

**@staticmethod**

**def make\_area(d, h):**

**circle = pi \* d \*\* 2 / 4**

**side = pi \* d \* h**

**return round(circle\*2 + side, 2)**

**def \_\_init\_\_(self, di, hi):**

**self.dia = diameter**

**self.h = high**

**self.area = self.make\_area(di, hi)**

**a = Cylinder(1, 2)**

**print(a.area)**

**print(a.make\_area(2, 2))**

В примере вызов make\_area() за пределами класса возможен в том числе через экземпляр. При этом понятно, в данном случае свойство area самого объекта a не меняется. Мы просто вызываем функцию, находящуюся в пространстве имен класса.

Рассмотрим следующий пример:

**from datetime import date**

**class Person:**

**def \_\_init\_\_(self, name, age):**

**self.name = name**

**self.age = age**

**@classmethod**

**def from\_birth\_year(cls, name, year):**

**return cls(name, date.today().year - year)**

**@staticmethod**

**def is\_adult(age):**

**return age > 18**

**person1 = Person('Sarah', 25)**

**person2 = Person.from\_birth\_year('Roark', 1994)**

**>>> person1.name, person1.age**

**Sarah 25**

**>>> person2.name, person2.age**

**Roark 24**

**>>> Person.is\_adult(25)**

**True**

Рассмотрим следующий пример:

**class Class1(object):**

**@staticmethod**

**def sum1(x, y): # Статический метод**

**return x + y**

**def sum2(self, x, y): # Обычный метод в классе**

**return x + y**

**def sum3(self, x, y):**

**return Class1.sum1(x, y) # Вызов из метода класса**

**print Class1.sum1(10, 20) # Вызываем статический метод**

**c1 = Class1()**

**print c1.sum2(15, 6) # Вызываем метод класса**

**print c1.sum1(50, 12) # Вызываем статический метод**

**# через экземпляр класса**

**print c1.sum3(23, 5) # Вызываем статический метод**

**# внутри класса**

Когда использовать каждый из методов?

Выбор того, какой из методов использовать, может показаться достаточно сложным. Тем не менее с опытом этот выбор делать гораздо проще.

Чаще всего метод класса используется тогда, когда нужен генерирующий метод, возвращающий объект класса. Как видим, метод класса from\_birth\_year используется для создания объекта класса Person по году рождения, а не возрасту.

Статические методы в основном используются как вспомогательные функции и работают с данными, которые им передаются.

Запомнить следующее:

Методы экземпляра класса получают доступ к объекту класса через параметр self и к классу через self.\_\_class\_\_.

Методы класса не могут получить доступ к определённому объекту класса, но имеют доступ к самому классу через cls.

Статические методы работают как обычные функции, но принадлежат области имён класса. Они не имеют доступа ни к самому классу, ни к его экземплярам.