ChatGPT

Конспект по продвинутым темам Python (ООП и объекты)

1. Классы, экземпляры, методы, атрибуты, параметр `self`

Класс – это шаблон (описание) объекта, определяющий набор атрибутов и методов, которые будут у его экземпляров. **Экземпляр класса** (или объект класса) – это конкретная реализация класса с собственным состоянием. Создание экземпляра происходит вызовом имени класса как функции (например, `obj = MyClass()`), что выполняет конструктор (инициализатор) `__init__` для настройки объекта.

Атрибуты – это переменные, хранящие состояние объекта или класса. Они бывают **атрибутами экземпляра** (уникальные для каждого объекта, обычно задаются в `__init__` через `self`, например `self.attr = value`) и **атрибутами класса** (общие для всех экземпляров, объявляются прямо в теле класса). Доступ к атрибутам: `obj.attr` – для обращения к атрибуту объекта, `ClassName.attr` – к атрибуту класса.

Методы – это функции, определённые внутри класса и предназначенные для работы с объектами этого класса. Первый параметр метода экземпляра обычно называется `self` – это ссылка на сам объект, для которого вызван метод. Python не добавляет `self` автоматически в определение, поэтому *каждый метод экземпляра обязан принимать* `self` *как первый параметр*. При вызове `obj.method(arg1, arg2)` интерпретатор автоматически передаёт объект `obj` в метод как `self`. Таким образом, внутри метода через `self` можно обращаться к атрибутам и другим методам данного экземпляра. Например:

```
class Point:
    def __init__(self, x, y):
        self.x = x            # атрибуты экземпляра
        self.y = y
    def distance_to_origin(self):
        # self ссылается на конкретный объект, от имени которого вызван метод
        return (self.x**2 + self.y**2) ** 0.5

p = Point(3, 4)
print(p.x, p.y)                # 3 4 – доступ к атрибутам экземпляра
print(p.distance_to_origin())  # 5.0 – вызов метода, self передаётся
автоматически
```

В примере `Point.__init__` получает `self` и координаты, инициализирует атрибуты `x` и `y` для каждого нового объекта. Метод `distance_to_origin` вычисляет расстояние, используя значения `self.x` и `self.y` конкретного экземпляра. Если забыть указать параметр `self` в определении метода, вызов `obj.method()` приведёт к ошибке `TypeError` (Python ожидает, что метод будет принимать `self`).

Резюме: класс определяет структуру и поведение, экземпляр хранит конкретное состояние, методы оперируют данными экземпляра через `self`, а атрибуты хранят состояние (данные) объекта или всего класса.

2. Методы класса (@classmethod), статические методы (@staticmethod), фабричные методы

Помимо обычных методов экземпляра, Python поддерживает *методы класса* и *статические методы*, определяемые с помощью декораторов `@classmethod` и `@staticmethod` соответственно.

- **Метод класса** – объявляется с декоратором `@classmethod` и получает не `self`, а **cls** (ссылку на сам класс) в качестве первого аргумента. Такой метод можно вызывать как через класс, так и через экземпляр. Он имеет доступ ко всему классу в целом: может читать и изменять переменные класса, создавать новые экземпляры (фабричный метод) и т.д. Например, метод класса часто используют для альтернативных конструкторов.

- **Статический метод** – объявляется с декоратором `@staticmethod`. Он не получает ни `self`, ни `cls` автоматически. Фактически, это просто функция, определённая внутри класса для логической организации, которая не зависит от состояния конкретного объекта или самого класса. Вызывать статический метод можно через имя класса или экземпляра. Статические методы полезны для вспомогательных задач, связанных с классом, но не требующих доступа к нему.

Фабричные методы – это обычный термин для методов, которые создают и возвращают новый объект (чаще всего *экземпляр* класса). В Python фабричные методы часто реализуют как classmethod. Например, если требуется несколько способов создать объект (разные форматы входных данных), основную логику размещают в методе класса. Такой метод получает класс `cls` и возвращает `cls(...)` – то есть создаёт экземпляр. Пример:

```
from datetime import datetime

class Event:
    def __init__(self, timestamp):
        # Инициализация из метки времени (число секунд)
        self.timestamp = int(timestamp)
        self.date = datetime.fromtimestamp(self.timestamp)

    @classmethod
    def from_datetime(cls, dt_obj):
        """Фабричный метод: создает Event из объекта datetime."""
        ts = int(dt_obj.timestamp())
        return cls(ts)  # вызывает Event.__init__

    @staticmethod
    def current_year():
        """Статический метод: возвращает текущий год (не зависит от
экземпляра)."""
        return datetime.now().year
```

```
# Использование:
now = datetime.now()
e1 = Event.from_datetime(now)         # создание экземпляра через метод
класса
e2 = Event(1700000000)                # создание напрямую через конструктор
year = Event.current_year()           # вызов статического метода
print(e1.date.year, e2.date.year, year)
```

В примере `Event.from_datetime` принимает класс `cls` и объект `datetime`, из него вычисляет timestamp и создаёт `Event` через `cls(ts)`. Метод `current_year` не трогает ни класс, ни объект – он просто утилита для получения текущего года.

Когда использовать: - `@classmethod` – когда логика метода касается самого класса *в целом* (например, альтернативные конструкторы, логика, влияющая на все экземпляры или на классовые атрибуты). - `@staticmethod` – когда функция связана по смыслу с классом, но не нуждается в доступе к нему. Это позволяет сгруппировать функцию внутри класса, сохранив нейтральную сигнатуру. - Обычные методы (`self`) – для операций, связанных с конкретным экземпляром и его состоянием.

3. Получение имени класса и атрибутов класса

Python предоставляет средства интроспекции, позволяющие во время выполнения узнавать информацию о классах и объектах. Например:

- **Имя класса.** Чтобы получить имя класса в виде строки, используют свойство `MyClass.__name__`. Для экземпляра можно взять `obj.__class__.__name__` (т.к. `obj.__class__` – это объект класса, от которого создан `obj`). Также модуль можно узнать через `MyClass.__module__`. Пример:

```
class MyClass:
    pass

obj = MyClass()
print(MyClass.__name__)         # "MyClass"
print(obj.__class__.__name__)   # "MyClass"
```

- **Атрибуты класса.** Полный список атрибутов (включая методы) класса можно получить с помощью встроенной функции `dir()` – она возвращает список имен атрибутов (включая унаследованные). Например, `dir(MyClass)` вернёт имена всех атрибутов класса `MyClass`.

Если нужны именно атрибуты, определённые в классе (без унаследованных), можно воспользоваться свойством `MyClass.__dict__`. Оно содержит словарь атрибутов класса (атрибуты класса, методы, свойства и т.д.) — правда, возвращается не сам dict, а специальный `mappingproxy`. Его можно обойти, например: `MyClass.__dict__.keys()` даст ключи-имена

атрибутов. Аналогично, для экземпляра `obj.__dict__` хранит словарь его индивидуальных атрибутов.

- **Примеры:** Допустим, есть класс:

```
class Point:
    z = 0  # атрибут класса
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def reset_z(cls):
        cls.z = 0  # может быть @classmethod, но демонстрируем через класс
```

Тогда: - `Point.__name__` вернёт строку `"Point"`. - `Point.__dict__` содержит ключи `'z'`, `'__init__'`, `'reset_z'`, … (и другие служебные атрибуты). - Для `p = Point(1,2)`, `p.__dict__` будет `{'x': 1, 'y': 2}` – только атрибуты этого экземпляра. - `dir(Point)` отобразит в том числе `x`, `y` не покажет (они не атрибуты класса), но покажет `z`, методы и унаследованные служебные методы (`__init__`, `__repr__`, и пр. из `object`). - `dir(p)` покажет и `'x'`, `'y'`, и `'z'`, и методы – всё, чем объект обладает (непосредственно или через класс).

Вывод: для имени класса используйте `.__name__`, для получения атрибутов – `__dict__` (как словарь атрибутов) или `dir()` (как список имён). Также существуют функции `vars()` (возвращает `__dict__` объекта или, без аргумента, локальные переменные) и ряд других, о которых подробнее в разделе 9.

4. Объекты в Python: ссылки, изменяемость, хешируемость, выделение памяти, тип объекта, создание своих типов, структура типов

Python реализует *модель объектов*, в которой все переменные являются ссылками на объекты в памяти. Понимание этой модели важно для правильной работы с изменяемыми и неизменяемыми типами, а также для использования объектов в коллекциях, требующих хеширования.

Ссылки и имена. В Python переменная не хранит сам объект, а лишь *ссылается* на него. Присваивание `a = b` делает так, что имя `a` ссылается на тот же объект, что и `b` (никакого копирования объекта по умолчанию не происходит). Можно считать, что имена – это ярлыки для объектов в области памяти (куче). Например:

```
a = [1, 2, 3]
b = a          # теперь a и b указывают на один и тот же список
b.append(4)
print(a)       # [1, 2, 3, 4] – изменение через b отразилось на a
print(a is b)  # True – оба имени ссылаются на один объект
```

В примере `a is b` возвращает `True`, т.к. это один объект. Понимание ссылок важно: объекты передаются в функции тоже по ссылке (то есть, функция получает копию *ссылки*, но не копию самого объекта).

Изменяемые и неизменяемые объекты. Типы данных делятся на *изменяемые* (mutable) и *неизменяемые* (immutable). **Изменяемые** можно менять после создания (например, список `list`, словарь `dict`, множество `set`, пользовательские объекты с изменяемыми атрибутами). **Неизменяемые** после создания менять нельзя (числа `int`, `float`, булевы, строки `str`, кортежи `tuple` и т.п.). Важное следствие: при попытке изменить неизменяемый объект фактически создаётся новый объект. Например:

```
x = 5
print(id(x))   # id объекта 5 (идентификатор объекта, адрес в памяти в
CPython)
x += 1         # операция создаёт новый int (6), а x теперь ссылается на
новый объект
print(id(x))   # другой id, т.к. это новый объект 6

y = [5]
print(id(y))   # id списка
y.append(6)
print(id(y))   # тот же id – объект списка изменён "на месте"
```

В этом коде для `int` при изменении значения идентификатор изменился (новый объект), а для списка – остался прежним. Функция `id(obj)` возвращает *уникальный идентификатор* объекта (в CPython это адрес в памяти). Таким образом, если несколько переменных имеют одинаковый `id`, значит они ссылаются на один объект.

Хешируемость объектов. Хешируемые объекты – это те, которые имеют хеш-значение (`hash(obj)`) и могут использоваться в качестве ключей в словарях или элементов множеств. Требование для хешируемых: их хеш не должен меняться за время жизни (что обычно означает, что объект неизменяем). Встроенные неизменяемые типы (`int`, `float`, `bool`, `str`, `tuple` (если все элементы тоже хешируемы) и др.) по умолчанию хешируемы – Python реализует для них метод `__hash__`. Изменяемые типы (`list`, `dict`, `set`, пользовательские объекты) по умолчанию *не* хешируемы, т.к. изменение их содержимого нарушило бы соответствие хеша содержимому. Попытка вызвать `hash()` на изменяемом объекте (например, `hash([1,2,3])`) вызовет `TypeError: unhashable type`.

Отметим, что **пользовательские объекты (экземпляры классов)** по умолчанию считаются хешируемыми, но только по своему `id`. То есть если вы не переопределили методы сравнения `__eq__` / `__hash__`, то `obj1 == obj2` по умолчанию сравнивает идентичность (как `is`), и `hash(obj)` базируется на `id(obj)`. Однако если в классе определить свой `__eq__`, Python автоматически пометит класс как не-хешируемый (требуя, чтобы вы явно определили согласованный `__hash__`, иначе такой объект не будет пригоден для ключей dict). Это сделано, чтобы предотвратить некорректное использование изменяемых в качестве ключей.

Выделение памяти и сборка мусора. В Python все объекты динамически размещаются в памяти (в куче). Переменные хранят ссылки на адреса объектов. Для управления памятью используется автоматический **сборщик мусора**: когда на объект больше нет ссылок, его память

может быть освобождена. В CPython основой управления памятью является *подсчёт ссылок*: у каждого объекта есть счетчик, увеличивающийся при присваивании или передаче объекта, и уменьшающийся при удалении ссылок (`del` или выходе из области видимости). Когда счётчик становится нулевым, объект уничтожается. Дополнительно сборщик мусора отслеживает циклические ссылки.

Для пользователя это означает, что не нужно явно освобождать память (нет аналогов `free`), Python делает это автоматически. Однако нужно помнить про ссылочную семантику, чтобы не создавать непреднамеренных копий больших объектов или, наоборот, не изменять объект через одну переменную, не желая затронуть другую.

Определение типа объекта. Чтобы узнать тип (класс) объекта во время выполнения, используют функцию `type(obj)`. Она возвращает объект класса, например: `type(5)` вернёт `<class 'int'>`, а `type(obj)` для экземпляра вернёт его класс. Альтернативно можно воспользоваться атрибутом `obj.__class__` – результат будет тем же самым. Для проверки, принадлежит ли объект классу (или его наследнику), служит функция `isinstance(obj, SomeClass)`. Например, `isinstance([1,2], list)` вернёт True. Для проверки родственных отношений классов (наследования) есть `issubclass(SubClass, BaseClass)`.

Создание своих типов. В Python создать новый тип можно посредством определения класса. `class MyClass: ...` создаёт пользовательский тип `MyClass`, экземпляры которого будут иметь указанные атрибуты и методы. При определении класса неявно вызывается *метакласс* (по умолчанию `type`), который и возвращает объект-класс. То есть классы в Python сами являются объектами (экземплярами метакласса `type`). В большинстве случаев достаточно знать, что синтаксис `class` создаёт новый тип, а `obj = MyClass()` создаёт его экземпляр.

Также можно динамически создавать типы, вызывая `type(name, bases, dict)` – но это более редкий продвинутый приём (обычно для динамических классов или metaprogramming). Пример: `NewClass = type('NewClass', (BaseClass,), {'x': 5})` создаст новый класс `NewClass`, наследующий `BaseClass`, с атрибутом класса `x = 5`. Однако в практике экзамена обычно достаточно стандартного определения через `class`.

Структура объектов разных типов. Внутреннее представление объектов в Python различается для разных типов: - У **пользовательских объектов** (экземпляров пользовательских классов) есть динамический словарь атрибутов `__dict__`. Это значит, что по умолчанию можно в любой момент добавить новый атрибут просто написав `obj.new_attr = val`. Словарь обеспечивает гибкость, но требует дополнительную память на хранение ключей и значений. - У многих **встроенных типов** (например, чисел, кортежей) фиксированная структура, определённая на уровне C: у них **нет** `__dict__` у каждого объекта. Например, целое число `int` хранит только значение числа; у списка `list` объект хранит указатель на массив элементов; у `tuple` – на последовательность элементов; у `dict` – на хеш-таблицу пар ключ-значение. Эти объекты не поддерживают добавление произвольных пользовательских атрибутов. Попытка сделать `some_int.new_attr = 5` приведёт к ошибке `AttributeError`, т.к. у int нет пространства имён для новых атрибутов. - Существует возможность ограничить **структуру** экземпляров пользовательского класса, используя специальный атрибут класса `__slots__`. Если задать в классе `__slots__ = ['x', 'y']`, то экземпляры этого класса смогут иметь только атрибуты `x` и `y` (и не будут иметь **dict**). Это экономит память и ускоряет доступ, но лишает гибкости добавления новых атрибутов на лету. **Slots** применимы, когда точно известен фиксированный набор полей объекта.

Таким образом, Python объекты универсальны в обращении, но их внутренняя реализация оптимизирована по-разному. Для большинства задач достаточно понимать, что у изменяемых объектов состояние хранится внутри них и меняется без смены идентификатора, а у неизменяемых – при "изменении" фактически получаем новый объект. И, конечно, что все имена – это ссылки на объекты в памяти.

5. Магические методы: `__init__`, `__repr__` vs `__str__`, `__len__`, `__call__`, `__add__`, `__sub__`, `__mul__`, `__truediv__`; класс `object` и синтаксический сахар

Магическими (dynder-) методами называют специальные методы, имена которых заключены в двойное подчеркивание (Double UNDERscore). Они определяют поведение объектов в различных языковых конструкциях и операциях. Эти методы не вызываются напрямую пользователем, а автоматически интерпретатором при использовании соответствующего синтаксического сахара. Все магические методы описаны в документации; здесь рассмотрены некоторые из наиболее распространённых:

- `__init__(self, ...)` – инициализатор экземпляра (конструктор в терминологии ООП). Вызывается автоматически сразу после создания нового объекта, чтобы задать его начальное состояние. Именно в `__init__` обычно присваиваются атрибуты `self.attr = ...`. Возвращать значение из `__init__` не нужно (он должен вернуть None). Например, `__init__` класса `Point` выше задаёт координаты.

- `__repr__(self)` **и** `__str__(self)` – методы, возвращающие строковые представления объекта. `__repr__` предназначен для разработки и отладки: должен возвращать официальное, однозначное представление объекта, по возможности *валидное выражение Python*, создающее такой объект (например, `<Vector x=1, y=2>` или `"Vector(1, 2)"`). Вызывается функцией `repr(obj)` или при выводе объекта в интерактивной консоли, а также по умолчанию используется для `str` если `__str__` не определён. `__str__` – более "читаемое" строковое представление для пользователя. Вызывается функцией `str(obj)` и, например, при печати `print(obj)`. Если определить только `__repr__`, то `str(obj)` будет использовать его. Обычно `__str__` – это удобочитаемая форма, а `__repr__` – более подробная. Пример:

```
class Person:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return f"Person(name={self.name!r})"  # официальное представление
    def __str__(self):
        return f"Person {self.name}"          # удобочитаемый вывод

p = Person("Alice")
print(p)             # Person Alice   (использует __str__)
print([p])           # [Person(name='Alice')]  (в списке используется
__repr__)
```

В примере `__repr__` возвращает строку `"Person(name='Alice')"` (в которой имя заключено в кавычки через `{self.name!r}`), а `__str__` – `"Person Alice"`. При печати объекта напрямую вызывается `__str__`, а при выводе объекта в коллекции (где не применяют `str` к элементам) используется `__repr__`. Это демонстрирует разницу назначения двух методов.

- `__len__(self)` – определяет поведение функции `len(obj)`. Должен вернуть целое число – "длину" объекта. Обычно имеет смысл для коллекций (список, строка, и т.д.). Если класс реализует `__len__`, его экземпляры можно передавать в `len()`. Например, если мы хотим, чтобы наш объект `Point` имел длину (например, количество координат), можно определить `__len__` возвращающий 2. Встроенные типы `list`, `str` и др. уже имеют этот метод. Следует возвращать неотрицательное число; `len()` использует его для разных проверок, например, пустой ли объект (правда, в Python для truth-value теста используют отдельный `__bool__`, но если его нет, вызывается `__len__ > 0`).

- `__call__(self, *args, **kwargs)` – делает объект *вызываемым как функция*. Если класс реализует `__call__`, то его экземпляры могут быть использованы с круглыми скобками, как `obj()`. Это вызывает метод `obj.__call__()`. Применение: функции-объекты, фабрики, настройка поведения при вызове. Например, можно сделать класс-счётчик вызовов:

```
class Counter:
    def __init__(self):
        self.count = 0
    def __call__(self, increment=1):
        self.count += increment
        return self.count

c = Counter()
print(c())       # 1  (первый вызов увеличил count на 1)
print(c(5))      # 6  (второй вызов увеличил count на 5, суммарно 6)
```

Объект `c` ведёт себя как функция: при каждом вызове увеличивает свой внутренний счётчик. Метод `__call__` используется в Python, например, для реализации декораторов (классы-декораторы) или для создания **функторов**.

- **Арифметические операторы** (`__add__`, `__sub__`, `__mul__`, `__truediv__`, и др.). Эти методы перегружают операторы `+ - * /` и другие. Когда в коде пишется `a + b`, на самом деле Python вызывает `a.__add__(b)` (а если его нет, попробует `b.__radd__(a)`, для случая, когда правый операнд знает как складывать слева). Аналогично:
- `a - b` вызывает `a.__sub__(b)`
- `a * b` -> `a.__mul__(b)`
- `a / b` -> `a.__truediv__(b)`
- Для целочисленного деления `a // b` вызывается `a.__floordiv__(b)`, остаток `%` -> `__mod__`, возведение в степень `**` -> `__pow__`, и т.д.
- Операторы сравнения: `==` -> `__eq__`, `!=` -> `__ne__`, `<` -> `__lt__`, `>` -> `__gt__`, и т.д.
- Побитовые: `|` -> `__or__`, `&` -> `__and__`, `^` -> `__xor__`, `<<` -> `__lshift__`, `>>` -> `__rshift__`.

- Унарные: `-a` -> `__neg__`, `+a` -> `__pos__`, логическое не `not a` -> `__bool__` (возвращает True/False).

Благодаря этим методам можно определять, как экземпляры вашего класса будут вести себя при использовании операторов. Например, реализуем сложение для пользовательского класса `Point`:

```
class Point:
    def __init__(self, x, y):
        self.x = x; self.y = y
    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        raise TypeError("Unsupported operand type for +")
    def __repr__(self):
        return f"Point({self.x}, {self.y})"

p1 = Point(1, 2)
p2 = Point(3, 4)
print(p1 + p2)    # Point(4, 6)
# p1.__add__(p2) была вызвана неявно при вычислении p1 + p2
```

Здесь определено, что `Point + Point` возвращает новый `Point` как сумму координат. Если `other` не `Point`, выбрасывается исключение. Встроенные числовые классы (`int`, `float`) при сложении возвращают новый объект-результат, не изменяя операнды. В своих классах можно выбирать: например, `__add__` для изменяемого типа мог бы изменить свой объект и вернуть его же (хотя так делать не рекомендуется, лучше чтобы `+` не мутировал объекты, а `+=` – уже по ситуации).

- **Другие магические методы.** Существует множество специальных методов для различных протоколов:
- Контекстный менеджер: `__enter__` и `__exit__` (для поддержки `with`).
- Итерирование: `__iter__` (возвращает итератор), `__next__` (для итератора), позволяющие делать объекты итерируемыми (поддержка цикла `for`).
- Управление доступом к атрибутам: `__getattr__`, `__setattr__`, `__delattr__` (вызываются при обращении/назначении/удалении атрибутов), а также `__getattribute__` (на каждом обращении), `__setitem__`, `__getitem__` для индексирования `obj[key]`, и т.д.
- Создание/уничтожение объектов: `__new__` (вызывается перед `__init__` для создания экземпляра, редко изменяется, кроме случаев изменения поведения выделения памяти) и `__del__` (деструктор, вызывается при сборке объекта).

В контексте данного конспекта главная идея: магические методы позволяют перегружать поведение объектов под синтаксис Python. Сам по себе класс `object` (базовый класс всех классов) предоставляет базовые реализации многих dunder-методов: например, `object.__str__` и `object.__repr__` по умолчанию возвращают строку вида `<__main__.ClassName object at 0x...>` (адрес в памяти); `object.__eq__` сравнивает объекты по идентичности (то же что `is`); `object.__hash__` возвращает уникальный хеш (обычно на основе адреса). Путём переопределения этих методов в своём классе, мы получаем

так называемый *синтаксический сахар* – возможность использовать привычные операторы и встроенные функции (len, str, etc.) с нашими объектами.

Итого: магические методы всегда начинаются и заканчиваются на двойное подчеркивание. Они не предназначены для прямого вызова извне (хотя можно, например `obj.__len__()` – но лучше `len(obj)`). Их цель – чтобы объекты интегрировались в язык: чтобы экземпляр можно было печатать, складывать, вызывать, сравнивать и т.д., используя стандартный синтаксис Python.

6. Инкапсуляция: сокрытие данных, изменение имён (name mangling), геттеры и сеттеры, границы архитектуры, декоратор `@property`

Инкапсуляция – принцип ООП, согласно которому внутреннее устройство объекта (его данные и детали реализации) скрываются от внешнего кода, предоставляя взамен публичный интерфейс (методы/свойства). Цель – защитить внутреннее состояние от некорректного использования и уменьшить связанность компонентов системы.

В Python **нет жесткого механизма** для ограничения доступа к атрибутам (как, например, `private/protected` модификаторов в C++/Java). Вместо этого действует соглашение об именах: - Атрибут, имя которого начинается с одного подчёркивания `_attr`, считается **«защищённым»** (protected) – означает, что он не предназначен для внешнего использования напрямую. Это лишь договорённость: синтаксически Python не запрещает обращаться к нему извне, но по стилю программирования такой доступ нежелателен. - Имя, начинающееся с двойного подчёркивания `__attr`, трактуется как **«приватный»**. Python применяет к нему механизм **name mangling** («искажение имени»): внутри класса имя сохраняется с префиксом класса, например `__attr` превращается в `_ClassName__attr`. Это сделано, чтобы затруднить случайный доступ извне и избежать конфликтов имён в наследниках. Однако и это не абсолютная защита: зная правило, извне можно обратиться к такому атрибуту как `_ClassName__attr`. Обычно же двойное подчёркивание используется, чтобы явно показать: «не лезь – внутреннее».

Пример:

```
class Robot:
    def __init__(self, name):
        self.name = name          # публичный атрибут
        self._model = "RX-78"     # защищённый (не менять снаружи)
        self.__password = "s3cr3t"  # приватный (сугубо внутренний)

r = Robot("R2D2")
print(r.name)       # "R2D2", нормальный доступ
print(r._model)     # "RX-78", технически доступно, но не рекомендуется
# print(r.__password)  # AttributeError: 'Robot' object has no attribute
'__password'
print(r._Robot__password)  # "s3cr3t" – получили приватный атрибут по
mangled-имени (неделать!)
print(r.__dict__)
# {'name': 'R2D2', '_model': 'RX-78', '_Robot__password': 's3cr3t'}
```

Видно, что прямой доступ `r.__password` провалился – интерпретатор поменял имя на `_Robot__password`. В словаре `__dict__` экземпляра ключи отображают реальные имена. Пользоваться mangled-именем вне класса не стоит – это нарушает инкапсуляцию.

Name mangling не предназначен для абсолютного сокрытия (как Reflection все равно может доступ получить), а скорее, чтобы избежать случайных конфликтов имен при наследовании. Например, если класс-наследник определит свой `__password`, он не переопределит поле базового класса, а создаст *собственное* `_Child__password`, независимо от `_Parent__password`. Таким образом, двойное подчёркивание – инструмент для *внутренних нужд класса*.

Геттеры и сеттеры. В классическом ООП (Java, C#) для доступа к приватным полям пишут методы-геттеры (getValue) и сеттеры (setValue). В Python прямой доступ к атрибутам допускается, но если требуется логика при получении или присвоении, нужно использовать методы. Простейший способ – явно написать методы:

```
class Player:
    def __init__(self, name):
        self.__name = None
        self.set_name(name)
    def get_name(self):
        return self.__name
    def set_name(self, name):
        if isinstance(name, str):
            self.__name = name.strip()
        else:
            raise TypeError("Name must be string")

p = Player(" Mario ")
print(p.get_name())  # "Mario"
p.set_name("Luigi")
```

Здесь `__name` приватный, а доступ осуществляется через функции. Но в Python есть более элегантный подход – **дескрипторы-свойства**.

Декоратор `@property` позволяет превратить методы класса в атрибуты для пользователя класса. Это способ реализовать геттеры/сеттеры, сохраняя синтаксис доступа как к обычному атрибуту. Делается так: - Описываем метод, который возвращает значение, и декорируем его `@property`. Этот метод называется геттером (хотя имя можно любое, обычно совпадает с именем "виртуального" атрибута). - Опционально описываем метод для присваивания (с той же логикой проверки), декорируем его `@<property_name>.setter`. - Также можно описать deleter (`@<property_name>.deleter`) для удаления атрибута.

Пример использования `@property` – переделаем класс Player выше:

```
class Player:
    def __init__(self, name):
        self._name = None      # защищённый атрибут (реальное хранение)
```

```
        self.name = name       # вызывает сеттер

    @property
    def name(self):
        """Имя игрока (строка, не пустая)."""
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError("Name must be a string")
        value = value.strip()
        if value == "":
            raise ValueError("Name cannot be empty")
        self._name = value

p = Player(" Mario ")
print(p.name)    # "Mario" – вызвалась функция name(self) как геттер
p.name = "Luigi" # вызвался name.setter
# p.name = ""    # ValueError: Name cannot be empty
```

В этом примере внешний код не знает о `_name` и работает с `p.name` как с обычным свойством. При чтении вызывается геттер `name()`, при присваивании – сеттер `name()`. Внутри сеттера можно валидировать данные. Таким образом достигается инкапсуляция: реализация хранения (`_name`) скрыта за интерфейсом свойства `name`. При необходимости можно изменить реализацию (например, хранить имя в другом формате) без изменения внешнего кода.

Архитектурная граница. Инкапсуляция устанавливает *границу* между внутренним устройством объекта и внешним миром. Внешний код должен полагаться только на публичные методы и свойства. Это позволяет модифицировать внутренности класса, не влияя на остальную программу, что особенно важно в больших проектах. Правило: *«Минимум деталей наружу»* – публикуйте только то, что необходимо для использования класса, остальное прячьте (хотя бы через соглашения). `@property` помогает соблюсти этот принцип, предоставляя контролируемый доступ к данным.

В Python, даже используя `__` и `@property`, нет абсолютной приватности, но дисциплина разработки предполагает уважение этих механизмов. Если видите имя с `_` – это сигнал "внутреннее, не трогать". Если видите свойство (property) – используйте его, а не лезьте во внутренний `_attr`.

7. Наследование: назначение, множественное и иерархическое наследование, `super()`, переопределение свойств (`@property`)

Наследование – механизм ООП, позволяющий одному классу (*подкласс*, класс-наследник) получать атрибуты и методы другого класса (*базового, родительского*). Подкласс *расширяет или специализирует* поведение базового. Главные цели наследования: - **Повторное использование кода.** Общая функциональность выносится в базовый класс, наследники используют её, не дублируя код. - **Логическое моделирование.** Связь «является» (*is-a*): можно выразить, что,

скажем, класс `Car` является частным случаем более общего класса `Vehicle`. Подкласс получает свойства базового, но также может иметь свои особенности. - **Полиморфизм.** К объектам подклассов можно обращаться через интерфейс базового класса, и они будут вести себя согласно своей реализации (об этом – в следующем разделе).

Наследование в Python задаётся в объявлении класса: `class B(SuperClass): ...`. Если указано несколько базовых через запятую – получается **множественное наследование**. Если базовый класс не указан, неявно наследуется от `object` (все новые классы являются наследниками `object` – *«новый стиль»* классов).

Когда уместно наследование: когда классы действительно находятся в отношении обобщения/специализации. Например, есть база `Shape` (фигура) с методом `area()`, и подклассы `Circle`, `Rectangle` с собственной реализацией площади – наследование оправдано. Нельзя использовать наследование без смысловой из-а связи (если вам просто нужно повторно использовать код, но объекты не подходят под отношение "является", лучше использовать композицию/агрегацию – держать объект как атрибут).

Иерархическое наследование – это обычное однонаследование, создающее древовидную иерархию: один базовый -> потомки -> их потомки и т.д. Например: `Animal` -> `Mammal` -> `Dog`. **Множественное наследование** позволяет классу иметь более одного родителя: `class C(A, B): ...`. Python поддерживает это, но важно понимать порядок разрешения методов (MRO).

MRO (Method Resolution Order) – порядок, в котором Python ищет атрибуты в иерархии наследования. При сложном (множественном) наследовании Python использует алгоритм C3-линеаризации, обеспечивающий разрешение конфликтов. Узнать MRO класса можно через атрибут `ClassName.__mro__` или функцией `help(ClassName)`. Обычно MRO – это глубина-вперёд слева-направо, с разрешением "ромбов" (diamond problem) корректно: базовый класс вызывается один раз. Это важно при вызове `super()`.

Функция `super()`. В Python она возвращает прокси-объект, делегирующий вызовы методам следующего класса по MRO. Проще говоря, `super(SubClass, self)` позволяет вызвать метод родителя, не явно указывая имя базового класса. В Python обычно `super()` вызывают внутри методов подкласса, чтобы использовать базовую реализацию. Например, переопределяя `__init__`, мы часто хотим вызвать `__init__` базового, чтобы он инициализировал свою часть:

```
class A:
    def __init__(self, x):
        self.x = x
        print("A init")

class B(A):
    def __init__(self, x, y):
        super().__init__(x)      # вызов A.__init__, передаем параметр x
        self.y = y
        print("B init")

b = B(5, 10)
# Вывод:
```

```
# A init
# B init
```

Здесь `super().__init__(x)` внутри `B.__init__` вызывает конструктор класса `A`. `super()` без аргументов внутри класса B автоматически определяется как `super(B, self)`. Важно: в случае множественного наследования `super()` не просто обращается к конкретному родителю, а идёт по цепочке MRO. То есть, если класс C наследует A и B, и оба они наследуют object, `super(C, self)` в методе класса C пойдёт в следующий по MRO класс (например, B), а `super(B, self)` внутри B – дальше (к A), и т.д. Таким образом, при правильном использовании `super()` везде, даже в ромбовидной схеме каждый родительский метод будет вызван ровно один раз.

Переопределение методов и свойств. Подкласс может предоставлять свою реализацию метода, объявленного в базовом классе – это называется *override* (переопределение). При этом, вызов метода на экземпляре подкласса будет использовать вариант подкласса, даже если вызов сделан через переменную типа базового класса. Пример:

```
class Animal:
    def voice(self):
        print("Some generic animal sound")

class Dog(Animal):
    def voice(self):
        print("Woof!")  # переопределяем voice

class Cat(Animal):
    def voice(self):
        print("Meow!")

animals = [Dog(), Cat(), Animal()]
for a in animals:
    a.voice()
# Вывод:
# Woof!
# Meow!
# Some generic animal sound
```

Даже хоть `animals` объявлен как список Animal, реально вызываются методы Dog/Cat там где нужно – это и есть проявление полиморфизма с наследованием.

Аналогично можно переопределять и атрибуты, и свойства. Если в базовом классе определён метод-геттер @property, в подклассе его можно переопределить, просто объявив метод с тем же именем и тоже пометив `@property`. Сеттеры/делитеры тоже можно переопределять аналогично. Пример:

```
class Base:
    @property
    def value(self):
```

```
        return 10

class Sub(Base):
    @property
    def value(self):
        # используем базовое значение и видоизменяем
        return super().value * 2

print(Sub().value)  # 20
```

Здесь `Sub.value` переопределяет свойство `Base.value`. При вызове использован `super().value` чтобы получить исходное значение из Base и умножить на 2. Можно и полностью заменить логику. Если базовый класс имеет property с сеттером, в подклассе можно переопределить только геттер (тогда сеттер унаследуется, но может не соответствовать новой логике) или определить и свой сеттер (синтаксис `@value.setter` внутри Sub). В Python 3 нет прямого аналога `protected`: либо "приватим" через `__`, либо считаем все методы, не начинающиеся с _, публичными и допускаем их переопределение.

Множественное наследование и конфликт имён. Если два родительских класса имеют одноимённый метод, а в наследнике он не переопределён, то вызов метода пойдёт по MRO – то есть к первому родителю по порядку, и если не найден – дальше. Чтобы избежать конфликтов, иногда используют *миксины* – шаблон, когда один из родителей предназначен не как самостоятельный объект, а как дополнительный функционал. Его имя указывает на роль (например, класс LoggingMixin с методом `log()`), а основной функционал идёт от другого родителя. В миксине важно выбирать имена методов, чтобы не столкнуться случайно с основным классом.

Общее правило: наследование – мощный инструмент, но применять его надо осмысленно. Глубокие иерархии затрудняют сопровождение. Множественное наследование – очень осторожно, внимательно изучая MRO. Python даёт свободу, но ответственность за дизайн лежит на разработчике.

8. Полиморфизм

Полиморфизм – свойство, позволяющее обращаться с разными объектами единообразно (через общий интерфейс), при этом они могут вести себя по-разному. Проще: один и тот же код (один и тот же вызов метода) автоматически применяется к объектам разных классов. Различают полиморфизм на основе наследования и полиморфизм "утиный" (duck typing) в динамически типизированных языках.

В классическом ООП полиморфизм достигается через наследование: если класс-наследник переопределяет метод базового класса, то, будучи передан туда, где ожидается базовый класс, он будет выполнять свою версию метода. Мы видели это в примере с `Animal` в разделе наследования: функция не знала, Dog это или Cat – она звала `a.voice()`, и каждая реализация сама себя проявила. Таким образом, **полиморфизм подтипов** – способность экземпляра подкласса выступать в роли экземпляра базового класса.

В Python же полиморфизм часто понимается шире благодаря **утиной типизации**: *«Если что-то выглядит как утка и крякает как утка, то это утка»*. Это означает, что объект не обязан наследоваться от конкретного базового класса, чтобы быть полиморфно применимым –

достаточно, чтобы у него были нужные методы. Например, Python-функция, которая читает из любого объекта методом `.read()`, не требует, чтобы объект наследовался от какого-то общего класса File – достаточно, что у него есть метод `read`. То же с операторами: любая последовательность, у которой есть метод `__len__`, будет работать с `len()`, не обязательно наследоваться от общего Sequence.

Пример полиморфизма (утиная типизация):

```
class Cat:
    def speak(self):
        print("Meow")
class Dog:
    def speak(self):
        print("Woof")
class Human:
    def speak(self):
        print("Hello")

def make_speak(creature):
    # Полиморфная функция - вызывает метод speak, не важно у кого
    creature.speak()

for creature in (Cat(), Dog(), Human()):
    make_speak(creature)
# Вывод:
# Meow
# Woof
# Hello
```

Здесь `make_speak` вызывает `speak()` у любого переданного объекта. Объекты `Cat`, `Dog`, `Human` никак не связаны по наследованию, но все умеют `speak`, поэтому функция с ними справляется – это полиморфизм через протокол (наличие метода).

Полиморфизм и встроенные функции. Многие встроенные функции и операторы в Python работают полиморфно: - `len(obj)` – сработает с любым объектом, у которого есть метод `__len__`. - оператор `+` – сработает для любых объектов, у которых есть реализация `__add__` (и/или `__radd__`). - итерация `for x in obj` – пойдёт по элементам любого объекта, реализующего итераторный протокол (`__iter__` и `__next__`). - функция `sorted(seq)` – сможет отсортировать любую последовательность, если её элементы сравнимы друг с другом через `__lt__`, и сама последовательность итерируема.

Таким образом, Python очень гибок: вы можете написать свои классы, и если реализуете необходимые методы, их экземпляры будут "полиморфно" работать там, где ожидаются стандартные.

Абстрактные базовые классы (ABC). Хотя Python не требует явной реализации интерфейсов, для крупных проектов иногда используют модуль `abc` для объявления абстрактных методов (методов, которые должны быть переопределены в наследниках). ABC могут задать “контракт”,

что класс должен иметь определенный набор методов, тем самым определяя интерфейс. Но даже без ABC, динамическая природа Python позволяет полиморфизм на уровне соглашений.

Вывод: Полиморфизм облегчает расширяемость кода – можно добавлять новые классы, которые работают с уже написанными функциями, если реализуют нужные методы. В Python полиморфизм достигается как через иерархии классов, так и через duck typing, что делает код гибким и менее зависимым от конкретных типов.

9. Вспомогательные функции и атрибуты: `__dir__()`, `__dict__`, `vars()`, `help()`, `__class__`, `id()`, `hex()`

Наконец, рассмотрим различные встроенные средства интроспекции и дополнительные функции, помогающие работать с объектами:

- `__dict__` – атрибут, содержащий словарь *локальных атрибутов* объекта (или класса). Через `obj.__dict__` можно получить или даже изменить значения атрибутов объекта. Например, `obj.__dict__` вернёт что-то вроде `{'x': 10, 'y': 20}` для объекта Point. У класса `MyClass.__dict__` хранит методы и атрибуты класса (в виде специального объекта-словаря). Обычно `__dict__` используется для интроспекции или копирования атрибутов, но прямое присвоение в него нечасто применяется (лучше `setattr`).

- `vars([object])` – встроенная функция, возвращающая словарь атрибутов. По сути, `vars(obj)` то же самое, что `obj.__dict__` (если у объекта есть **dict**). Если вызвать `vars()` без аргументов внутри функции, она вернёт словарь локальных переменных. Часто `vars()` удобен тем, что не надо писать напрямую имя атрибута, но разницы немного. Пример: `vars(obj)` -> `{'x': 10, 'y': 20}`. Для объектов без **dict** (например, у которых **slots**, либо для встроенных, у которых нет динамических атрибутов) `vars()` может не работать.

- `__dir__(self)` – магический метод, определяющий поведение функции `dir(obj)`. По умолчанию, `dir(obj)` возвращает упорядоченный список имен атрибутов объекта: оно собирает атрибуты экземпляра (из **dict**), атрибуты класса и его базовых классов. Если в классе определить метод `__dir__`, то вызов `dir(obj)` будет возвращать то, что вернёт этот метод (должен возвращать список строк). Это можно использовать, чтобы скрывать или, наоборот, динамически добавлять "виртуальные" атрибуты в вывод `dir`. Но обычно `__dir__` редко переопределяют. Пример (для понимания):

```
class Secretive:
    def __dir__(self):
        return ['magic_attr', 'help']
s = Secretive()
print(dir(s))
# ['help', 'magic_attr'] – вместо стандартных атрибутов выводится заданный
список
```

В жизни такое нечасто нужно, `dir()` удобнее использовать по умолчанию для отладки – узнать, какие атрибуты у объекта.

- `help(obj)` – встроенная функция, запускающая интерактивную справку (из модуля `pydoc`). Она показывает документацию (docstring) объекта, его методы и т.п. Например, `help(str)` выведет подробности о классе `str` и его методах. `help(obj)` для пользовательского объекта покажет докстринг его класса и методResolutionOrder, атрибуты. `help(SomeModule)` – выдаст содержание модуля. **Примечание:** В среде REPL (например, CPython) часто эквивалентно использовать `obj?` в IPython или `help(obj)`.

- `__class__` – атрибут каждого объекта, указывающий на его класс (то же, что `type(obj)`). Можно использовать, чтобы получить сам класс, а затем его имя или другие свойства. Пример:

```
obj = [1, 2, 3]
print(obj.__class__)          # <class 'list'>
print(obj.__class__.__name__) # "list"
```

У класса тоже есть **class** – это метакласс, обычно `<class 'type'>`. Но чаще для класса интересен **mro** или **bases** (базовые классы).

- `id(obj)` – возвращает целое число, идентифицирующее объект. В CPython это фактически адрес объекта в памяти (для живых объектов он уникален). После удаления объекта его id может быть повторно использован для других объектов. `id` полезен, чтобы отличить объекты (например, проверить `id(a) == id(b)` эквивалентно `a is b`). Однако напрямую адреса обычно не используют, это скорее отладочная функция. Например:

```
a = {1, 2, 3}
b = a
print(id(a), id(b))   # числа совпадут
c = {1, 2, 3}
print(id(a), id(c))   # разные, с хоть и равен по содержимому, но другой
объект
```

В выводе `id` часто представляют в шестнадцатеричном виде, что приводит нас к следующей функции.

- `hex(x)` – функция, возвращающая строковое шестнадцатеричное представление числа `x`. Чаще всего здесь актуально, что если `x` – это результат `id(obj)`, то `hex(id(obj))` даст привычный адрес вида `'0x7f9c1234abcd'`. Например:

```
obj = object()
print(id(obj))        # 140366927954096 (десятичное число, у вас будет
другое)
print(hex(id(obj)))   # 0x7fa4b6c4cf50   (шестнадцатеричная форма того же
адреса)
```

Именно такую форму (с `0x`) вы видите, когда печатаете объект без спец. **str**: `<object object at 0x7fa4b6c4cf50>`. `hex` полезна исключительно для форматирования чисел, в том числе id.

Замечание: Модуля *sys* есть функция `sys.getsizeof(obj)`, возвращающая размер объекта в байтах, а в модуле *gc* – `gc.get_referrers(obj)` и прочие, но они выходят за рамки конспекта.

Итого по разделу: - `__dict__` / `vars` дают доступ к атрибутам объекта в виде словаря; - `__dir__` / `dir()` – список имён атрибутов (по умолчанию формируется автоматически); - `help()` – интерактивная справка (docstrings + структура); - `__class__` – ссылка на класс объекта; - `id()` – уникальный идентификатор (адрес) объекта; - `hex()` – представление числа (например, id) в hex-формате.

Эти инструменты облегчают отладку, рефлексию и понимание объектов во время разработки.
