Классы 1

Классы

Синтаксис объявления классов

```
>>> class Counter:
...     """I count. That is all."""
...     def __init__(self, initial=0):  # конструктор
...         self.value = initial        # запись атрибута
...
...     def increment(self):
...         self.value += 1
...
...     def get(self):
...         return self.value           # чтение атрибута
...
>>> c = Counter(42)
>>> c.increment()
>>> c.get()
43
```

- В отличие от Java и C++ в Python нет “магического” ключевого слова `this`. Первый аргумент конструктора `__init__` и всех остальных методов — экземпляр класса, который принято называть `self`.
- Синтаксис языка не запрещает называть его по-другому, но так делать не рекомендуется:

```
>>> class Noop:
...     def __init__(ego):
...         pass
...
>>> Noop = Noop()
```

- Аналогично другим ООП языкам Python разделяет атрибуты экземпляра и атрибуты класса.
- Атрибуты добавляются к экземпляру посредством присваивания к `self` конструкцией вида:
  `self.some_attribute = value`
- Атрибуты класса объявляются в теле класса или прямым присваиванием к классу:

```
>>> class Counter:
...     all_counters = []
...
...     def __init__(self, initial=0):
...         Counter.all_counters.append(self)
...         # ...
...
>>> Counter.some_other_attribute = 42
```

Соглашения об именовании атрибутов и методов

- В Python нет модификаторов доступа к атрибутам и методам: почти всё можно читать и присваивать.
- Для того чтобы различать публичные и внутренние атрибуты визуально, к внутренним атрибутам добавляют в начало символ подчеркивания:

```
>>> class Noop:
...     some_attribute = 42
...     _internal_attribute = []
```

- Особо ярые любители контроля используют два подчёркивания:

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

```
class MemorizingDict(dict):
    history = deque(maxlen=10)

    def set(self, key, value):
        self.history.append(key)
        self[key] = value

    def get_history(self):
        return self.history

d = MemorizingDict({"foo": 42})
d.set("baz", 100500)
print(d.get_history())  # ==> ?

d = MemorizingDict()
d.set("boo", 500100)
print(d.get_history())  # ==> ?
```

```
>>> class Noop:
...     """I do nothing at all."""
...
>>> Noop.__doc__
'I do nothing at all.'
>>> Noop.__name__
'Noop'
>>> Noop.__module__
'__main__'
>>> Noop.__bases__
(<class 'object'>,)
>>> noop = Noop()
>>> noop.__class__
<class '__main__.Noop'>
>>> noop.__dict__  # словарь атрибутов объекта
{}
```

Вопрос

Как вы думаете, чему равняются `Noop.__class__` и `Noop.__dict__`?

- Все атрибуты объекта доступны в виде словаря:

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

- Очевидные следствия:
  - Добавление, изменение и удаление атрибутов — это фактически операции со словарём.

```
>>> noop.__dict__["some_other_attribute"] = 100500
>>> noop.some_other_attribute
100500
>>> del noop.some_other_attribute
```

  - Поиск значения атрибута происходит динамически в момент выполнения программы.

- Для доступа к словарю атрибутов можно также использовать функцию `vars`:

```
>>> vars(noop)
{'some_attribute': 42}
```

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

```
>>> class Noop:
...     __slots__ = ["some_attribute"]
...
>>> noop = Noop()
>>> noop.some_attribute = 42
>>> noop.some_attribute
42
>>> noop.some_other_attribute = 100500
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Noop' object has no attribute [...]
```

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

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

```
>>> class SomeClass:
...     def do_something(self):
...         print("Doing something.")
...
>>> SomeClass().do_something  # связанный
<bound method SomeClass.do_something of [...]>
>>> SomeClass().do_something()
Doing something.
```

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

```
>>> SomeClass.do_something    # несвязанный
<function SomeClass.do_something at 0x105466a60>
>>> instance = SomeClass()
>>> SomeClass.do_something(instance)
Doing something.
```

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

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

- Можно также переопределить логику изменения и удаления таких атрибутов.

```
>>> class BigDataModel:
...     def __init__(self):
...         self._params = []
...
...     @property
...     def params(self):
...         return self._params
...
...     @params.setter
...     def params(self, new_params):
...         assert all(map(lambda p: p > 0, new_params))
...         self._params = new_params
...
...     @params.deleter
...     def params(self):
...         del self._params
...
>>> model = BigDataModel()
>>> model.params = [0.1, 0.5, 0.4]
>>> model.params
[0.1, 0.5, 0.4]
```

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

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

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

```
>>> c = OtherCounter()  # вызывает Counter.__init__
>>> c.get()             # вызывает OtherCounter.get
0
>>> c.value             # c.__dict__["value"]
```

```
>>> class Counter:
...     all_counters = []
...
...     def __init__(self, initial=0):
...         self.__class__.all_counters.append(self)
...         self.value = initial
...
>>> class OtherCounter(Counter):
...     def __init__(self, initial=0):
...         self.initial = initial
...         super().__init__(initial)
...
>>> oc = OtherCounter()
>>> vars(oc)
{'initial': 0, 'value': 0}
```

Вопрос

Как можно было бы реализовать функцию `super`?

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

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> isinstance(B(), A)
True
```

- В качестве второго аргумента можно также передать кортеж классов:

```
>>> class C:
...     pass
...
>>> isinstance(B(), (A, C))
True
>>> isinstance(B(), A) or isinstance(B(), C)
True
```

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

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> issubclass(B, A)
True
```

- Аналогично `isinstance` второй аргумент может быть кортежем классов:

```
>>> class C:
...     pass
...
>>> issubclass(B, (A, C))
True
>>> issubclass(B, A) or issubclass(B, C)
True
```

Python не запрещает множественное наследование, например, можно определить следующую иерархию:

```
>>> class A:
...     def f(self):
...         print("A.f")
...
>>> class C(A, B):
...     pass
...
```

```
>>> class B:
...     def f(self):
...         print("B.f")
...
```

Вопрос

Что выведет следующий фрагмент кода?

```
>>> C().f()
???
```

- В случае множественного наследования Python использует алгоритм линеаризации C3 для определения метода, который нужно вызвать.
- Получить линеаризацию иерархии наследования можно с помощью метода `mro`:

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

- Результат работы алгоритма C3 далеко не всегда тривиален, поэтому использовать сложные иерархии множественого наследования не рекомендуется.
- Больше примеров, а также реализацию алгоритма C3 на Python можно найти по ссылке: http://bit.ly/c3-mro.

- Классы-примеси позволяют выборочно модифицировать поведение класса в предположении, что класс реализует некоторый интерфейс.
- Продолжая пример со счётчиком:

```
>>> class ThreadSafeMixin:
...     get_lock = ...
...
...     def increment(self):
...         with self.get_lock():
...             super().increment()
...
...     def get(self):
...         with self.get_lock():
...             return super().get()
...
>>> class ThreadSafeCounter(ThreadSafeMixin,
...                         Counter):
...     pass
...
```

- Синтаксис декораторов работает не только для функций, но и для классов[1]:

```
>>> @deco
... class Noop:
...     pass
...
```

```
>>> class Noop:
...     pass
...
>>> Noop = deco(Noop)
```

- В этом случае декоратор — это функция, которая принимает класс и возвращает другой, возможно, преобразованный, класс.
- Декораторы классов можно также использовать вместо чуть более магических классов-примесей.

[1] http://python.org/dev/peps/pep-3129

- Декораторы классов можно использовать вместо классов-примесей.
- Например, `ThreadSafeMixin` в виде декоратора класса выглядит следующим образом образом:

```
>>> def thread_safe(cls):
...     orig_increment = cls.increment
...     orig_get = cls.get
...
...     def increment(self):
...         with self.get_lock():
...             orig_increment(self)
...
...     def get(self):
...         with self.get_lock():
...             return orig_get(self)
...
...     cls.get_lock = ...
...     cls.increment = increment
...     cls.get = get
...     return cls
```

```
>>> def singleton(cls):
...     instance = None
...
...     @functools.wraps(cls)
...     def inner(*args, **kwargs):
...         nonlocal instance
...         if instance is None:
...             instance = cls(*args, **kwargs)
...         return instance
...
...     return inner
...
>>> @singleton
... class Noop:
...     "I do nothing at all."
...
>>> id(Noop())
4383371952
>>> id(Noop())
4383371952
```

Декораторы классов: @deprecated

```
>>> import warnings
>>> def deprecated(cls):
...     orig_init = cls.__init__
...
...     @functools.wraps(cls.__init__)
...     def new_init(self, *args, **kwargs):
...         warnings.warn(
...             cls.__name__ + " is deprecated.",
...             category=DeprecationWarning)
...         orig_init(self, *args, **kwargs)
...
...     cls.__init__ = new_init
...     return cls
...
>>> @deprecated
... class Counter:
...     def __init__(self, initial=0):
...         self.value = initial
...
>>> c = Counter()
__main__:6: DeprecationWarning: Counter is deprecated.
```

- Синтаксис объявления классов в Python:

```
>>> class SomeClass(Base1, Base2, ...):
...     """Useful documentation."""
...     class_attr = ...
...
...     def __init__(self, some_arg):
...         self.instance_attr = some_arg
...
...     def do_something(self):
...         pass
```

- В отличие от большинства объектно-ориентированных языков Python:
  - делает передачу ссылки на экземпляр явной, `self` — первый аргумент каждого метода,
  - реализует механим свойств — динамически вычисляемых атрибутов,
  - поддерживает изменение классов с помощью декораторов.

“Магические” методы

- “Магическими” называются внутренние методы классов, например, метод `__init__`.
- С помощью “магических” методов можно:
  - управлять доступом к атрибутам экземпляра,
  - перегрузить операторы, например, операторы сравнения или арифметические операторы,
  - определить строковое представление экземпляра или изменить способ его хеширования.
- Мы рассмотрим только часть наиболее используемых методов.
- Подробное описание всех “магических” методов можно найти в документации языка[2].

[2] http://bit.ly/magic-methods

“Магические” методы: `__getattr__`

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

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

- Определим метод `__getattr__` для класса `Noop`:

```
>>> class Noop:
...     def __getattr__(self, name):
...         return name  # identity
...
>>> Noop().foobar
'foobar'
```

“Магические” методы: `__setattr__` и `__delattr__`

- Методы `__setattr__` и `__delattr__` позволяют управлять изменением значения и удалением атрибутов.
- В отличие от `__getattr__` они вызываются для всех атрибутов, а не только для несуществующих.
- Пример, запрещающий изменять значение некоторых атрибутов:

```
>>> class Guarded:
...     guarded = []
...
...     def __setattr__(self, name, value):
...         assert name not in self.guarded
...         super().__setattr__(name, value)
...
>>> class Noop(Guarded):
...     guarded = ["foobar"]
...
...     def __init__(self):
...         self.__dict__["foobar"] = 42  # Зачем это?
...
```

Функции getattr, setattr и delattr

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

```
>>> class Noop:
...     some_attribute = 42
...
>>> noop = Noop()
>>> getattr(noop, "some_attribute")
42
>>> getattr(noop, "some_other_attribute", 100500)
100500
```

- Комплементарные функции `setattr` и `delattr` добавляют и удаляют атрибут:

```
>>> setattr(noop, "some_other_attribute", 100500)
>>> delattr(noop, "some_other_attribute")
```

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

```
instance.__eq__(other)  # instance == other
instance.__ne__(other)  # instance != other
instance.__lt__(other)  # instance < other
instance.__le__(other)  # instance <= other
instance.__gt__(other)  # instance > other
instance.__ge__(other)  # instance >= other
```

- В уже знакомом нам модуле `functools` есть декоратор, облегчающий реализацию операторов сравнения:

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

“Магический” метод `__call__`

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

```
>>> class Identity:
...     def __call__(self, x):
...         return x
...
>>> Identity()(42)
42
```

- Как это можно использовать?

Декораторы с аргументами на основе классов

```
>>> class trace:
...     def __init__(self, handle):
...         self.handle = handle
...
...     def __call__(self, func):
...         @functools.wraps(func)
...         def inner(*args, **kwargs):
...             print(func.__name__, args, kwargs,
...                   file=self.handle)
...             return func(*args, **kwargs)
...         return inner
...
>>> @trace(sys.stderr)
... def identity(x):
...     return x
...
>>> identity(42)
identity (42,) {}
42
```

“Магические” методы для преобразования в строку

- Напоминание: в Python есть две различных по смыслу функции для преобразования объекта в строку: `repr` и `str`.
- Для каждой из них существует одноимённый “магический” метод:

```
>>> class Counter:
...     def __init__(self, initial=0):
...         self.value = initial
...
...     def __repr__(self):
...         return "Counter({})".format(self.value)
...
...     def __str__(self):
...         return "Counted to {}".format(self.value)
...
>>> c = Counter(42)
>>> c
'Counter(42)'
>>> print(c)
Counted to 42
```

Расширение format через “магические” методы

```
>>> class Counter:
...     def __init__(self, initial=0):
...         self.value = initial
...
...     def __format__(self, format_spec):
...         return self.value.__format__(format_spec)
...
>>> c = Counter(42)
>>> "Counted to {:b}".format(c)
'Counted to 101010'
```

- Метод `__hash__` используется для вычисления значения хеш-функции.
- Реализация по умолчанию гарантирует, что одинаковое значение хеш функции будет только у физически одинаковых объектов, то есть:

  `x is y <=> hash(x) == hash(y)`.
- Несколько очевидных рекомендаций:
  - Метод `__hash__` имеет смысл реализовывать только вместе с методом `__eq__`. При этом реализация `__hash__` должна удовлетворять: `x == y => hash(x) == hash(y)`
  - Для изменяемых объектов можно ограничиться только методом `__eq__`.

- Метод `__bool__` для проверки значения на истинность, например в условии оператора **`if`**.
- Для класса `Counter` реализация `__bool__` тривиальна:

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

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