Python.
Классы. Продолжение

Лекция 8

Преподаватель: *Дмитрий Косицин*

Обработка ошибок и менеджеры контекста

Подходы к обработке ошибок. Предупреждения. Менеджеры контекста.

Множественное наследование

Работа с атрибутами

Свойства и дескрипторы

Статические методы и методы класса. Свойства. Дескрипторы.

Механизм создания классов

Полезные ссылки

Обработка ошибок и менеджеры контекста

Подходы к обработке ошибок. Предупреждения. Менеджеры контекста.

Подходы к обработке ошибок

- Look Before You Leap (LBYL) – более общий и читаемый:

```
def get_second_LBYL(sequence):
    if len(sequence) > 2:
        return sequence[1]
    else:
        return None
```

- Easier to Ask for Forgiveness than Permission (EAFP) – не тратит время на проверку:

```
def get_second_EAFP(sequence):
    try:
        return sequence[1]
    except IndexError:
        return None
```

Вопрос: какой подход лучше и в каком случае?

Предупреждения

Помимо исключений, в Python есть и предупреждения (модуль **warnings**). Они не прерывают поток выполнения программы, а лишь явно указывают на нежелательное действие.

Примеры:

- **DeprecationWarning** – сообщение об устаревшем функционале
- **RuntimeWarning** – некритичное сообщение о некорректном значении

```
>>> def deprecation(message):
>>>     warnings.warn(message, DeprecationWarning,
...                   stacklevel=2)
```

Менеджеры контекста

В процессе работы с файлами важно корректно работать с исключениями: файл необходимо закрыть в любом случае.

Данный синтаксис позволяет закрыть файл по выходе из блока **with**:

```
with open(file_name) as f:
    # some actions
```

Функция **open** возвращает специальный объект – *context manager*. Менеджер контекста последовательно *инициализирует* контекст, *входит* в него и корректно обрабатывает *выход*.

Пример менеджера контекста

```
class ContextManager(object):
    def __init__(self):
        print('__init__()')

    def __enter__(self):
        print('__enter__()')
        return 'some data'

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__({}, {})'.format(
            exc_type.__name__, exc_val))


with ContextManager() as c:
    print('inside context "%s"' % c)
```

Менеджер контекста

Менеджер контекста работает следующим образом:

- создается и инициализируется (метод **__init__**)
- организуется вход в контекст (метод **__enter__**) и возвращается объект контекста (в примере с файлом – объект типа **file**)
- выполняются действия внутри контекста (внутри блока **with**)
- организуется выход из контекста с возможной обработкой исключений (метод **__exit__**)

В примере будет выведено следующее:

```
__init__()
__enter__()
inside context "some data"
__exit__(None, None)
```

Менеджер контекста

Замечание. Если исключения не произошло, то параметры, передаваемые в функцию **__exit__** – тип, значение исключения и ***traceback*** – имеют значения **None**.

Замечание. Менеджер контекста, реализуемый функцией **open**, по выходе из контекста просто вызывает метод ***close*** (см. декоратор *contextlib.closing*).

Менеджеры контекста используются:

- для корректной, более простой и переносимой обработки исключений в некотором блоке кода
- Для управления ресурсами

Декоратор *contextlib.contextmanager* позволяет создать менеджер контекста из функции-генератора, что значительно упрощает синтаксис.

Менеджер контекста из генератора

```
>>> @contextlib.contextmanager
>>> def get_context():
>>>     print('__enter__()’)
>>>     try:
>>>         yield 'some data’
>>>     finally:
>>>         print('__exit__()’)
>>>
>>> with get_context() as c:
>>>     print('inside context "%s"' % c)
__enter__()
inside context "some data"
__exit__()
```

Механизм создания классов

Механизм создания классов

Определение класса приводит к следующим действиям:

1. Определяется подходящий ***метакласс*** (класс, который создает другие классы)
2. Подготавливается namespace класса
3. Выполняется тело класса
4. Создается объект класса и присваивается переменной

```
class X(object):
    a = 0

# equivalent: type(name, bases, namespace)
X = type('X', (object, ), {'a': 0})
```

Замечания по созданию классов

Метаклассом по умолчанию является **type**.

Выполнение тела класса приводит к созданию ***словаря*** всех его атрибутов, который передается в **type**. Далее этот словарь доступен через **__dict__** или с помощью built-in функции **vars**.

Замечание. Изменять, добавлять и удалять атрибуты *можно*, модифицируя **__dict__**. Данный способ **менее явный**, нежели использование **getattr** и пр.

Важно! Атрибуты классов при наследовании не перезаписываются, а поиск их происходит последовательно в словарях базовых классов.

Замечания по созданию классов

Вопрос: есть ли разница между реализацией синонима (alias) для функции (функции g и h в примере)?

```
class X(object):
    def f(self):
        return 0

    def g(self):
        return self.f()

    h = f
```

Обычно реализация синонимов необходима при реализации операторов.

Множественное наследование

...

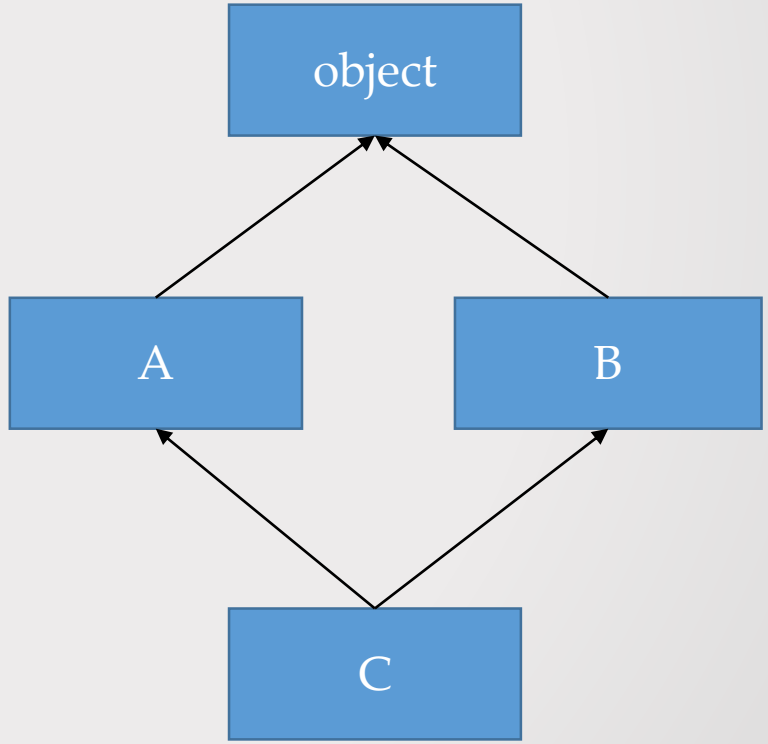
Множественное наследование

В Python допустимо множественное наследование. Не все схемы наследования допустимы.

Наиболее распространенные виды:

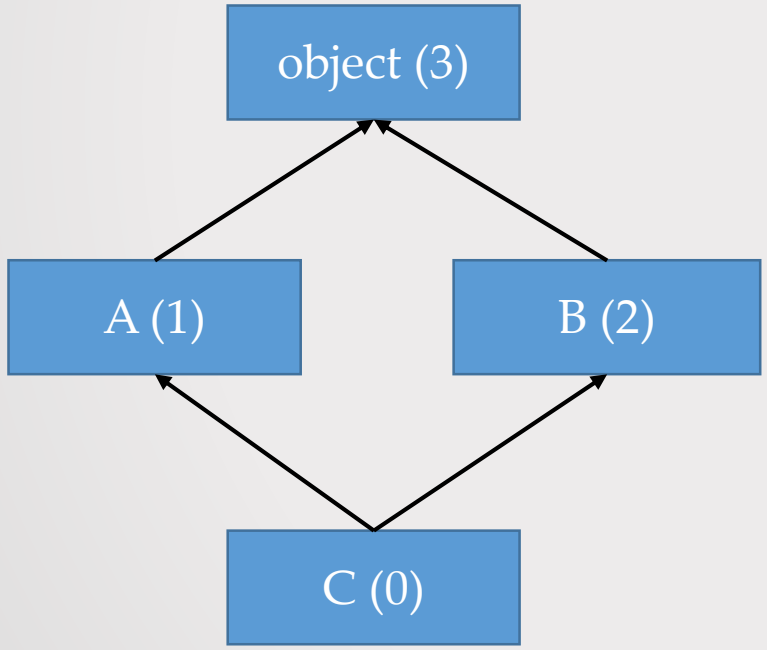
- ромбовидное наследование
- добавление Mixin-классов.

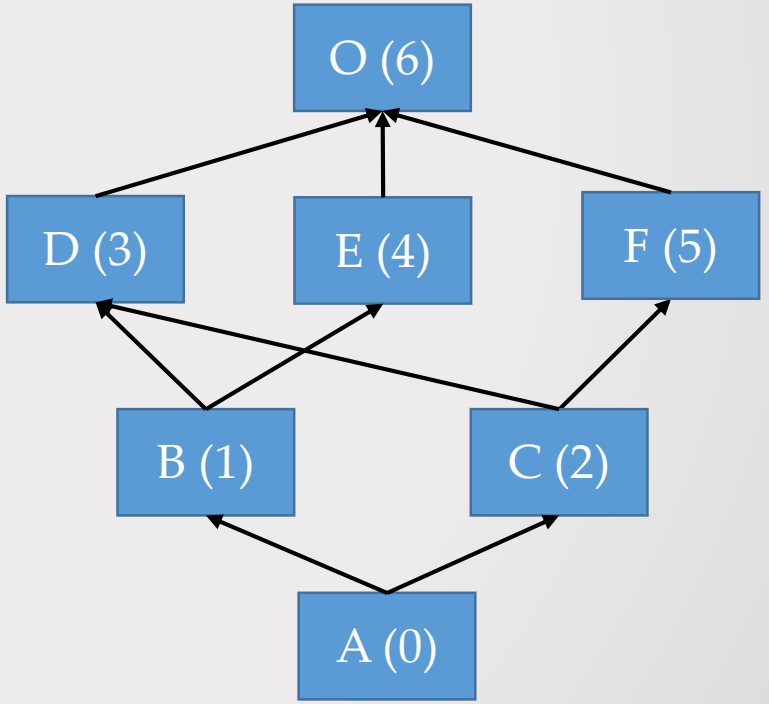
```
class A(object):
    pass

class B(object):
    pass

class C(A, B):
    pass
```



Порядок разрешения имен

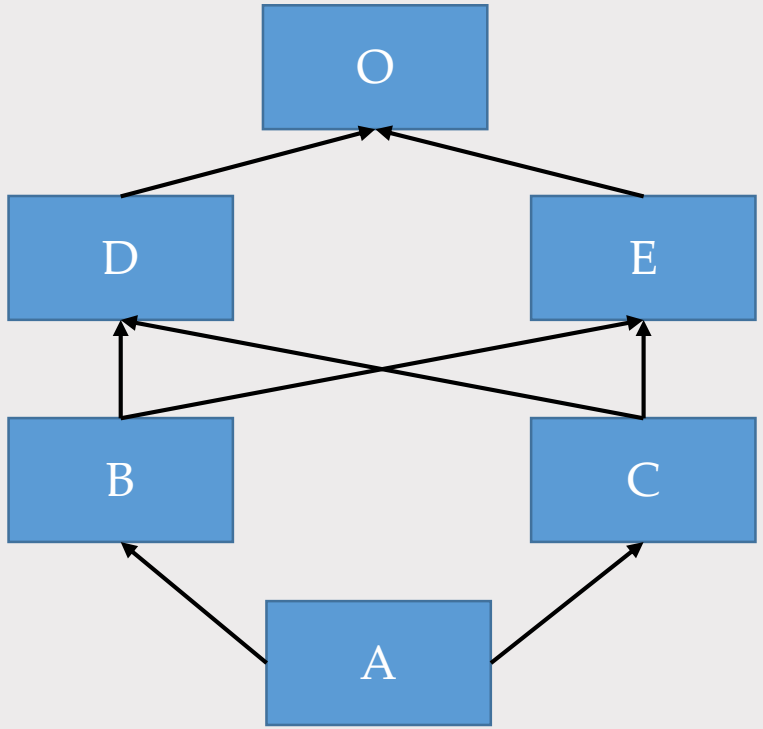
Если атрибут отсутствует в классе, предпринимается попытка найти его в базовых классах согласно MRO (Method resolution order). Алгоритм поиска – C3-линеаризация.





Пример недопустимой иерархии

В случае некорректной иерархии произойдет **TypeError**: Cannot create a consistent method resolution order (MRO). Такая иерархия не линеаризуема.



Порядок разрешения имен

```
>>> class C(object):
>>>     pass
>>>
>>> c = C()
>>> c.attribute
```

Поиск атрибутов:

1. Поискать атрибуты через механизм дескрипторов
2. Поискать атрибут в c.__**dict**__
3. Поискать атрибут в C.__**dict**__
4. Поискать атрибут в родительских классах согласно MRO
5. raise **AttributeError**

__dict__ – словарь атрибутов объекта

Обращение к методам базовых классов

К методам базовых классов можно обращаться как:

- **super**(<*class_name*>, self).method(…)
- <*class_name*>.method(self, …)

```
class C(object):
    def __init__(self):
        super(C, self).__init__()

class D(object):
    def __init__(self):
        D.__init__(self)
```

Случай множественного наследования

Proxy-объект **super** вернет только один основной базовый класс. Вызвать конструктор всех базовых классов нужно явно.

```
class A(object):
    pass

class B(object):
    pass

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
```

Mixin-классы только добавляют функционал (реализуют некоторые методы).

Встроенные базовые классы

В модуле **collections** (**collections.abc** в Py3.3+) находятся некоторые базовые классы. Их используют для:

- проверки типов (Callable, Iterable, Mapping)
- создания собственных типов (например, коллекций вида MidSkipQueue)

В данных классах содержатся *абстрактные* (требующие реализации) методы, а также mixin-методы, выраженные через другие.

Пример. Класс **Sequence** требует наличия реализации методов **__getitem__** и **__len__**, а методы **__contains__**, **__iter__**, **__reversed__**, **index** и **count** реализует, обращаясь к **__getitem__** и **__len__**.

Вывод: для проверки возможности *вызвать* объект или *итерироваться* по нему, следует проверить, что он наследуется от данных базовых классов.

Работа с атрибутами

Динамические атрибуты

Python позволяет *создавать*, *изменять* и *удалять* атрибуты run-time. За это отвечают следующие магические методы и глобальные функции:

- Получение атрибута по имени: **__getattr__**, **__getattribute__** | **getattr**
- Присваивание атрибута по имени: **__setattr__** | **setattr**
- Удаление атрибута по имени: **__delattr__** | **delattr**
- Проверка наличия атрибута: **hasattr** (функция-обертка над **getattr**)

Замечание. Некоторые объекты являются ***readonly***, например, добавлять или удалять атрибуты **object** нельзя.

Пример работы с атрибутами

```
class A(object):
    def f(self):
        pass

a = A()

assert hasattr(a, 'f') and hasattr(A, 'f')  # 'f' is a class attribute
assert getattr(a, 'g', None) is None   # no such attribute -> default

setattr(a, 'x', 2)  # a.x = 2
assert getattr(a, 'x') == 2  # assert a.x == 2
assert not hasattr(A, 'x')  # attribute has been set for instance only

delattr(a, 'x')  # del a.x
assert not (hasattr(a, 'x') or hasattr(A, 'x'))
```

Магические методы работы с атрибутами

Метод	Вызывается	Примечание
__getattr__(self, name) # x.name	При обращении к атрибуту, которого **нет**	• Возвращает значение или бросает AttributeError
__getattribute__(self, name) # x.name	При обращении к ***любому*** атрибуту	• Возвращает значение или бросает AttributeError • Обращение к другим атрибутам: super().__getattribute__(name)
__setattr(self, name, value) # x.name = value	При установке ***любого*** атрибута	• Может изменять другие атрибуты вызовом super().__setattr__(name, value)

Пример реализации методов работы с атрибутами

```
class Proxy(object):
    def __init__(self, inner_object):
        self._inner_object = inner_object

    def __setattr__(self, name, value):
        if name != '_inner_object':
            setattr(self._inner_object, name, value)
        else:
            super(Proxy, self).__setattr__(name, value)

    def __getattribute__(self, name):
        if name == '_inner_object':
            return object.__getattribute__(self, name)
        return getattr(self._inner_object, name)

p = Proxy([1])  # p._inner_object = [1]
p.append(2)  # p._inner_object = [1, 2]
```

Замечания по работе с атрибутами

В примере *Proxy* можно было ограничиться переопределением метода **__getattr__**:

```
class Proxy(object):
    ...
    def __getattr__(self, name):
        return getattr(self._inner_object, name)
```

Важно! Интерпретатор оптимизирует вызов **всех** *магических* методов: явный вызов x.**__len__** обратится к **__getattribute__**, неявный **len**(x) – нет.

Замечание. Создавать новые методы, присваивая функции объекту или классу, не корректно. Присваивать нужно объекты типа types.**MethodType**.

Свойства и дескрипторы

Статические методы и методы класса. Свойства. Дескрипторы.

Пример реализации инкапсуляции

```
class Animal(object):
    def __init__(self, age=0):
        self._age = age

    def get_age(self):
        """age of animal"""
        return self._age

    def set_age(self, age):
        assert age >= self._age
        self._age = age

    def increment_age(self):
        self.set_age(1 + self.get_age())
```

Доступ к атрибутам

Проблема: для каждого атрибута помимо методов работы с ним нужны *getter* и *setter*, иначе атрибут можно произвольно изменять извне.

```
class Animal(object):
    def __init__(self, age=0):
        self._age = age

    @property
    def age(self):
        """age of animal"""
        return self._age

    @age.setter
    def age(self, age):
        assert age >= self._age
        self._age = age
```

Свойства

Для доступа к атрибутам используются свойства – методы с декоратором **property** (docstring свойства получается из getter'а):

- getter – @property
- setter – @<name>.setter
- deleter – @<name>.delete

Полный синтаксис декоратора **property** имеет вид:

```
age = property(fget, fset, fdelete, doc)
```

Замечание. Создать *write-only* свойство можно только явно вызвав **property** с параметром *fget* равным **None**.

Свойства как замена функций

Для того, чтобы не хранить атрибуты, напрямую зависимые от других, можно реализовать доступ к ним с помощью свойств.

```
class PathInfo(object):
    def __init__(self, file_path):
        self._file_path = file_path

    @property
    def file_path(self):
        return self._file_path

    @property
    def folder(self):
        return os.path.dirname(self.file_path)
```

Декораторы ***staticmethod*** и ***classmethod***

Для реализации статических методов в классах используют специальный декоратор **staticmethod**, *при этом параметр* ***self*** *при вызове не передается.*

```
>>> class A(object):
>>>     @staticmethod
>>>     def f(a, b):
>>>         return a + b
>>>
>>> A.f(1, 2) == A().f(1, 2)
True
```

В функцию, декорированную **classmethod**, *первым параметром вместо объекта класса (instance) передается сам класс (параметр обычно называют* ***cls***).

Дескрипторы

Свойства (**property**) и декораторы **staticmethod** и **classmethod** являются *дескрипторами* – специальными объектами, реализованными как атрибуты класса (непосредственного или одного из родителей).

В классах в зависимости от типа дескриптора реализуются методы:

- **__get__**(self, instance, owner)
- **__set__** (self, instance, value)
- **__delete__**(self, instance)

Стандартное поведение декрипторов заключается в работе со словарями объекта, класса или базовых классов.

Пример. Вызов **a.x** приводит к вызову **a.__dict__['x']**, потом **type(a).__dict__['x']** и далее по ценпочке наследования.

Пример дескриптора

```
class Descriptor(object):
    def __init__(self, label):
        self.label = label

    def __get__(self, instance, owner):
        return instance.__dict__.get(self.label)

    def __set__(self, instance, value):
        instance.__dict__[self.label] = value

class C(object):
    x = Descriptor('x')

c = C()
c.x = 5
print(c.x)
```

Замечания

Замечания по стилю

Для улучшения читаемости зачастую логически разделенные блоки кода отделяют пустой строкой:

```
>>> for i in range(20):
>>>     print(i)
>>>
>>> for j in range(2, 10):
>>>     print(j**2)
```

Поскольку функции в Python нельзя перегрузить (сделать с разными сигнатурами), используют параметры по умолчанию. Не измененные параметры принято не указывать:

```
>>> range(10)   # range(0, 10, 1)
>>> range(2, 7)   # range(2, 7, 1)
```

Замечания по работе с функциями

В Python стек рекурсии ограничен (по умолчанию 1000, может быть изменена). Оптимизация хвостовой рекурсии отсутствует.

Напоминание. **Mutable** значения по умолчанию не создаются вновь при каждом вызове! Используйте **None** или *sentinel object*, чтобы избежать ошибок.

В модуле *functools* содержатся интересные функции:

- ***partial*** – возвращает новую функцию, фиксируя некоторые аргументы (замена, например **lambda** x: f(x, True); также ***partialmethod*** в Py3.4+)
- ***lru_cache*** – декоратор для кэширования результатов функции (Py3.2+)
- ***singledispatch*** – декоратор, позволяющий вызывать функцию в зависимости от типа ее аргумента (Py3.4+, примеры)

Замечания по операторам

Проверка на тип в методе **__eq__** обязательна, поскольку требуемых атрибутов для сравнения у другого класса может не быть.

Если тип неверный, бросается исключение **TypeError** (неявно переходит в **False** при сравнении с помощью оператора "==").

Реализация – оператора "==" не означает, что "!=" будет работать корректно. Для этого следует явно перегрузить метод **__ne__**.

Для реализации сложения (как и других арифметических операций) есть методы **__add__** ("+") и **__iadd__** ("+="). Первый должен создавать копию объекта, а второй – модифицировать исходный объект и возвращать его.

Полезные ссылки

...

Полезные ссылки

- C3-линеаризация (цепочка поиска метода среди предков): https://en.wikipedia.org/wiki/C3_linearization