Python

Лекция 2

Преподаватель: *Дмитрий Косицин*
BSU FAMCS (Fall'20)

Производные коллекции и алгоритмы

Встроенные коллекции

В Python реализованы следующие коллекции:

- **deque** – дек, двухсторонняя очередь
- **defaultdict** – словарь, который возвращает значение по умолчанию в случае отсутствия ключа
- **Counter** – реализация **defaultdict**, когда для всех ключей значение по умолчанию – ноль
- **OrderedDict** – словарь, сохраняющий порядок вставки элементов
- **namedtuple** – именованный кортеж
- **Queue** – потокобезопасная очередь
- **array** – массив, хранящий данные определенного C-совместимого типа

Подробнее о defaultdict

Подробнее о **defaultdict**:

```
>>> import collections
>>> def f():
>>>     return 0
>>> x = collections.defaultdict(f)
>>> print(x[2])
0
```

Важно! Конструктор **defaultdict** требует не число, а объект, при вызове которого будет возвращаться объект.

Подробнее о namedtuple

Подробнее о **namedtuple** (именованный кортеж):

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> print(p.x == p[0] == 1 and p.y == p[1] == 2)
True
```

Методы **namedtuple**:

- ***_fields*** – вернет имена полей (‘x’, ‘y’)
- ***_asdict()*** – вернет **OrderedDict** с соответствующими ключами и значениями
- ***_replace(x=new_value, …)*** – вернет **namedtuple** с замененными значениями

Алгоритмы стандартной библиотеки

В стандартной библиотеке реализованы алгоритмы по работе с кучей и упорядоченным списком:

- **heapq** – модуль, содержащий функции по созданию кучи (heap), добавлению элементов, взятию k-максимальных
- **bisect** – модуль, содержащий функцию бинарного поиска элемента по списку, а также вставки элемента в упорядоченную последовательность

Функции

...

Синтаксис функций

Определение функции:

```
def f(positional, positional_with_default=default_value,
      *variadic_arguments, **keyword_arguments):
    pass
```

В Python нету перегрузки функций – используются значения по умолчанию и динамическая типизация.

Значение по умолчанию вычисляется единожды при определении функции, а потому *должно* быть неизменяемым.

Вызов функций

Примеры вызовов:

```
>>> def f(x, y=0, *args, **kwargs):
>>>     return x, y, args, kwargs

>>> f()   # TypeError
>>> f(1)   # x: 1, y: 0, args: tuple(), kwargs: {}
>>> f(1, 2)   # x: 1, y: 2, args: tuple(), kwargs: {}
>>> f(1, 2, 3)   # x: 1, y: 2, args: tuple(3), kwargs: {}
```

Допустима распаковка аргументов при вызове функции:

```
>>> f(*(1, 2, 3, 4))
# x: 1, y: 2, args: tuple(3, 4), kwargs: {}
```

Передача аргументов по ключевым словам

В Python 3.5+ (PEP-448) допустима передача нескольких аргументов для распаковки:

```
>>> f(*(1, 2, 3), *(5, 6))
# x: 1, y: 2, args: tuple(3, 5, 6), kwargs: {}
```

Аргументы можно передавать по ключевым словам (порядок произвольный):

```
>>> f(y=1, x=2)   # x: 2, y: 1, args: tuple(), kwargs: {}
```

Минусы:

- возможно более медленное выполнение (Issue 27574)
- проблемы с переименованием

Передача аргументов по ключевым словам

Переданные по ключевым словам аргументы, для которых нет имен, помещаются в kwargs:

```
>>> f(x=1, z=2)  # x: 1, y: 0, args: tuple(), kwargs: {'z': 2}
```

Замечание. Порядок kwargs гарантируется с Python 3.6 (PEP-468).

Допустима распаковка аргументов по ключевым словам:

```
>>> f(**{'x': 1, 'z': 2})
# x: 1, y: 0, args: tuple(), kwargs: {'z': 2}
```

Исключительно ключевые аргументы

В Python 3 функции могут принимать аргументы исключительно по ключевому слову (PEP-3102):

```
>>> def f(*skipped, some_value=0):
>>>     pass

>>> f(1, 2, 3)   # skipped: (1, 2, 3), some_value: 0
>>> f(some_value=100)   # skipped: tuple(), some_value: 100
```

Важно! Если значение по умолчанию не указано, функция обязана вызываться с данным ключевым аргументом, иначе возникнет **TypeError**.

Замечание. Имя variadic аргумента может быть опущено.

Алгоритм маппинга аргументов

Значения назначаются аргументам функции по порядку:

- На позиционные слоты
- На variadic аргумент (в кортеж неименованных элементов)
- Переданные по ключевым словам – либо на позиционные, либо в словарь

Важно! Если аргумент позиционный аргумент не был передан или ключевой аргумент был передан более 1 раза, возникнет **TypeError**.

Вопрос: что будет, если имя variadic аргумента опущено (вместо *args оставлен просто символ “*”), но в функцию переданы variadic аргументы?

Полный алгоритм маппинга аргументов также описан в PEP-3102.

Документация функций

Описание функции и их аргументов производится в ***docstring*** (PEP-257):

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    # some code here …
```

Документация может быть получена вызовом **help**(f) или взятием аргумента f.**__doc__**

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)
```

Области видимости переменных

Области видимости переменных. Замыкания. Лямбда-функции. Функционалы из стандартной библиотеки.

Области видимости переменных

Области видимости переменных определяются функциями:

- *built-in* – встроенные общедоступные имена (доступны через модуль builtins или __buitlins__, например, **sum**, **abs** и т.д.)
- *global* – переменные, определенные глобально для модуля
- *enclosing* – переменные, определенные в родительской функции
- *local* – локальные для функции переменные

Локальные переменные в функциях могут в них свободно изменяться, enclosing, global и built-in – только читаться (PEP-227).

Области видимости переменных

Пример:

```
>>> abs(2)  # built-in
>>> abs = dir  # global, overrides
>>> def f():
>>>     abs = sum  # enclosing
>>>     def g():
>>>         abs = max  # local
```

Для справки. (Нужно крайне редко). Для переопределения **abs** из функции **g** в функции **f** используется ключевое слово ***nonlocal***, для переопределения глобальной переменной **abs** – ключевое слово ***global*** (PEP-3104).

```
>>> global abs
>>> abs = max  # переопределит abs, глобальный для модуля
```

Переменные в циклах

Циклы ***не имеют*** своей области видимости: как только переменная была создана, она становится доступной и после цикла.

```
>>> for i in range(10):
>>>     if i > 5:
>>>         break
>>> print(i)
6
```

Замечание. В Python 3.4 и ниже так же «утекали» переменные из comprehension-выражений. Баг был исправлен в Python 3.5.

```
>>> x = [i for i in range(10)]
>>> print(i)
NameError: name 'i' is not defined  # Python 3.5
```

Локальные и глобальные переменные

В Python можно получить доступ ко всем локальным и глобальным переменным:

- **locals**() – словарь видимых локальных переменных
- **globals**() – аналогичный словарь глобальных переменных

Словари автоматически обновляются интерпретатором.

Замыкания

В функции доступны переменные, определенные уровнями выше – они *замыкаются*.

```
>>> def make_adder(x):
>>>     def adder(y):
>>>         return x + y
>>>     return adder

>>> add_five = make_adder(5)
>>> add_five(10) # 15
```

Важно! Значение замкнутой переменной получается ***каждый раз*** при вычислении выражения.

Пример замыкания

```
>>> x = 2

>>> def make_adder():
>>>     def adder(y):
>>>         return x + y
>>>     return adder

>>> add_x = make_adder()
>>> add_x(-2)   # 0

>>> del x   # delete 'x' - unbind variable name with object
>>> add_x(-2)   # NameError: name 'x' is not defined
```

Lambda-функции

Lambda-функции в Python допускают в себе одно *лишь* выражение:

lambda arguments: expression

Эквивалентно:

```
def <lambda_name>(arguments):
    return expression
```

Вопрос: как будет выглядеть **lambda**, которая ничего не принимает и не возвращает?

Важно! С точки зрения bytecode **lambda** аналогична функции с тем же кодом, но при использовании **def** объект-функция еще получает имя.

Пример lambda-функций

Функция, возвращающая сумму аргументов:

```
>>> lambda x, y: x + y
```

Пример списка lambda-функций:

```
>>> collection_of_lambdas = [lambda: i*i for i in range(6)]
>>>
>>> for f in collection_of_lambdas:
>>>     print(f())
```

Вопрос: что будет выведено в результате выполнения?

Пример lambda-функций

Поскольку вычисление происходит run-time, то для всех созданных функций значение **i** будет равно **5**. Переменная **i** была «захвачена» в comprehension-выражении, хоть и вне этого выражения она недоступна (в Python 3.6).

Для «захвата» *значения* можно создать *локальную для* **lambda** *копию*:

```
>>> lambdas = [lambda i=i: i*i for i in range(6)]
```

В модуле **operator** (Py2, Py3) есть множество функционалов, которыми можно пользоваться наряду с **lambda-**функциями:

```
>>> import operator
>>> # аналог: lambda x, y: x + y
>>> operator.add  # operator.__add__
```

Замечания по операторам

Помимо операторов арифметических операций и операций сравнения, есть функционалы для работы с атрибутами и элементами коллекций.

```
f = operator.attrgetter('name.first', 'name.last')
# the call f(b) returns (b.name.first, b.name.last)

g = operator.itemgetter(2, 5, 3)
# the call g(r) returns (r[2], r[5], r[3])

h = operator.methodcaller('name', 'foo', bar=1)
# the call h(b) returns b.name('foo', bar=1)
```

Функционалы и lambda-функции наряду с обычными функциями используются как аргументы, выполняющие некоторое действие, например, в **map**, **filter**.

Обработка ошибок

Подходы к обработке ошибок. Исключения. Предупреждения.

Типы ошибок

Ошибки, вообще говоря, бывают

- *синтаксические* (**SyntaxError**): переменная названа ‘for’, некорректный отступ
- *исключения*
  - некорректный индекс (**IndexError**)
  - деление на 0 (**ZeroDivisionError**)
  - и другие

Базовый класс для почти всех исключений – **Exception**. Однако есть *control flow* исключения: **SystemExit**, **KeybordInterrupt**, **GeneratorExit** – с базовым классом **BaseException**.

Вопрос: для чего такое разделение?

Замечание. **Exception** в свою очередь унаследован от **BaseException** (Py2, Py3).

Пример работы с исключениями

```
class MyValueError(ValueError):
    pass


def crazy_exception_processing():
    try:
        raise MyValueError('incorrect value')
    except (TypeError, ValueError) as e:
        print(e)
        raise
    except Exception:
        raise Exception()
    except:
        pass
    else:
        print('no exception raised')
    finally:
        return -1
```

Работа с исключениями

Можно создавать собственные исключения – их следует наследовать от **Exception** либо его потомком (например, **ValueError**).

Исключение бросается с помощью выражения **raise** <*исключение*>.

Основной блок обработки исключения начинается **try** и заканчивается любым из выражений – **except**, **else** или **finally**.

В блоке **except** обрабатывается исключение определенного типа и, при необходимости, бросается либо то же, либо иное исключение.

Блок **except** можно специфицировать *одним* или *несколькими* исключениями (в скобках через запятую), а присвоить локальной переменной объект исключения можно выражением **as**.

Работа с исключениями

Для обработки всех исключений стоит указывать тип **Exception**.

Замечание. Блок **except** без указания *типа* использовать нужно ***крайне редко***, иначе поток управления может быть некорректно изменен.

В случае исключения в блоке **try** интерпретатор будет последовательно подбирать подходящий блок **except**. Если ни один не подойдет или ни одного блока нет, исключение будет проброшено на уровень выше (по стеку вызовов).

Блок **else** выполнится, если в блоке **try** исключений не было.

Блок finally

Если исключения не было, по окончании блока **try**:

- выполняется блок **else**
- выполняется блок **finally**

Если исключение в **try** было:

- выполняется подходящий блок **except** если есть
- исключение сохраняется
- выполняется блок **finally**
- сохраненное исключение бросается выше по стеку вызовов

Очень тонкий момент: если в блоке **finally** есть **return**, **break** или **continue**, сохраненное исключение сбрасывается. В блоке **finally** оно не доступно.

Обработка исключений

В случае возникновения исключения в блоке **except**, **else** или **finally**, бросается новое исключение, а старое либо присоединяется (Python 3), либо сбрасывается (Python 2).

Сохраненное исключение можно получить, вызвав **sys.exc_info()** (кроме блока **finally**). Функция вернет тройку: тип исключения, объект исключения и ***traceback*** – объект, хранящий информацию о стеке вызовов (обработать его можно с помощью модуля **traceback**).

У исключений есть атрибуты типа *message*, однако набор атрибутов *различен для* разных типов. Преобразование к строке *не гарантирует* получения полной информации о типе ошибки и сообщении.

Особенности работы с Python 3

В Python 3 исключение доступно так же через вызов **sys.exc_info()**.

Если во время обработки будет брошено новое исключение, оригинальное исключение будет присоединено к новому и сохранено в атрибутах ***__cause__*** (явно) и ***__context__*** (неявно), а оригинальный **traceback** в атрибуте ***__traceback__***.

Если во время обработки исключения его нужно передать выше по стеку вызовов, можно использовать **raise** без аргументов.

Бросить новое исключение, явно сообщив информацию о старом или явно указав исходный traceback, можно так:

```
>>> raise Exception() from original_exc
>>> raise Exception().with_traceback(original_tb)
```

Замечание. Значение *original_exc* может быть **None** – в таком случае контекст явно присоединен не будет.

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
```

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)
```

Объектно-ориентированное программирование

Классы и объекты

Класс – тип данных, описывает модель некоторой сущности.

Объект – реализация этого класса.

Пример:

int – класс, **42** – объект этого класса (типа **int**)

Пустой класс:

```
>>> class Empty(object):
>>>     pass
```

Методы классов

Функции – вызываемые с помощью скобок объекты.

Методы – функции, которые первым аргументом принимают экземпляр соответствующего класса (обычно именуют его **self**).

```
>>> class Greeter(object):
>>>     def greet(self):
>>>         print("hey, guys!")
```

Атрибуты классов – поля, характеризующие класс и работу с ним. Методы также являются атрибутами – callable-объектами, которые работают с другими атрибутами класса.

Атрибуты объектов

```
>>> class Greeter(object):
>>>     def set_name(self, name):
>>>         self.name = name
>>>
>>>     def greet(self):
>>>         print("hey, %s!" % self.name)
>>>
>>> print(Greeter().greet())
```

Поле классу присвоится *лишь* во время исполнения после вызова метода *set_name*, поэтому в данном случае произойдет **AttributeError**.

Атрибуты классов

```
>>> class Greeter(object):
>>>     DEFAULT_NAME = 'guys'
>>>
>>>     def __init__(self, name=None):
>>>         self.name = name or self.__class__.DEFAULT_NAME
>>>
>>> g = Greeter()
```

Хорошим правилом будет установка всех атрибутов в конструкторе со значениями по умолчанию или **None**.

Здесь `DEFAULT_NAME` – атрибут класса, `name` – атрибут экземпляра класса.

Важно! Значение `DEFAULT_NAME` можно указать в качестве *default* – оно уже будет в контексте функции,– но *default* не изменится при изменении значения.

Атрибуты классов

```
>>> class Greeter(object):
>>>     DEFAULT_NAME = 'guys'
```

К атрибуту класса DEFAULT_NAME можно обращаться:

- по имени класса: Greeter.DEFAULT_NAME
- как к атрибуту класса: self.__class__.DEFAULT_NAME
- как к атрибуту экземпляра класса: self.DEFAULT_NAME

Важно! По сути, атрибуты классов – статические поля, которые доступны всем его экземплярам и разделяются (shared) между ними, а также по имени класса.

Именование полей класса

Для определения конструктора, переопределения операторов, получения служебной информации в Python используются методы/атрибуты со специальными именами вида __*__ (__*init*__, __*class*__ и т.п.).

Все атрибуты доступны извне класса (являются **public**).

Для обозначения **protected** атрибута используют префикс '_' (underscore), для **private** – '__' (two underscores).

Важно! Доступ к **protected** и **private** атрибутам по-прежнему возможен извне – название служит ***предупреждением***.

Именование полей класса

Преимущества

- Легче отлаживать и проверять код, у IDE больше возможностей
- Легче писать группы классов, связанные друг с другом
- Можно «перехватывать» изменение атрибутов

Замечания

- При желании «защиту» можно обойти
- Многие IDE предупреждают о том, что происходит доступ к **protected** или **private** атрибуту извне класса

Замечание. Обычно **private** атрибуты используют крайне редко, ведь доступ к ним извне все равно возможен: они доступны как __C_name, где C – имя класса, а name – имя атрибута.

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())
```

Свойства – декоратор *property*

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 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***).

Магические методы

...

Магические методы

Методы со специальными именами вида __*__ называют магическими (или *dunder*). Они отвечают за многие операции с объектом. Список магических методов можно увидеть в описании DataModel (Py2, Py3), а также на странице модуля **operator**.

Создание, инициализация, удаление класса: ***new***, ***init***, ***del***

Метод **call** переопределяет оператор вызова () (круглые скобки).

Метод **len –** взятие длины (может вызываться как **len**(.); в Python 3 есть также **length_hint**).

Приведение типа:

- к строке – ***repr***, ***str***/***unicode*** (Py2) или ***bytes***/***str*** (Py3)
- к **bool** – ***nonzero*** (Py2) или ***bool*** (Py3)

Сравнение и хеширование

Преобразовать, к строке объект можно вызвав **str**(x) или x.**__str__**(), к **bool** – **bool**(x) или x.**__bool__**() (*nonzero*).

Если метод преобразования к **bool** не реализован, возвращается результат метода ***__len__***, а если и его нет, то все объекты преобразуются к **True**.

Сравнение и хеширование: ***eq***, ***ne***, ***le***, ***lt***, ***ge***, ***gt*** и ***hash*.**

Замечание. Явно реализовывать все операторы не нужно. Достаточно метода ***eq*** и одного из методов сравнения, а также декоратора **functools.total_ordering.**

Сравнение и хеширование

Если класс переопределяет **__eq__**, то для метода **__hash__** должно выполняться одно из следующих утверждений:

- **__hash__** явно реализован
- явно присвоено **__hash__** = **None** (для изменяемых объектов-коллекций)
- явно присвоено **__hash__** =<ParentClass>.**__hash__** (если не изменен)

Если метод ***eq*** для двух объектов возвращает **True**, то ***hash*** объектов должен также совпадать.

Метод **__eq__** должен либо бросать **TypeError**, либо возвращать **NotImplemented**, если передан объект некорректного для сравнения типа.

Операции с числами

Можно переопределить любые математические операции: + (**add**), - (**sub**), * (**mul**), @ (**matmul**), / (**truediv**), // (**floordiv**), и прочие.

Помимо таких операций есть еще методы-компаньоны. Например, для сложения они называются **radd** и **iadd**. Метод **radd** вызывается, когда у левого операнда метод **add** не реализован, а **iadd** – для операции “+=“.

Также есть возможность переопределить операции приведения к типам **complex**, **float** и **int**, округления **round** и взятия модуля **abs**.

Замечания по операторам

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

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.

Прочие методы

Методы **getitem**, **setitem**, **delitem** – работа с индексами (оператор []).

Методы **iter**, **reversed**, **contains** отвечают за итерирование и проверку вхождения.

Методы **instancecheck** и **subclasscheck** отвечают за проверку типа.

Метод **missing** вызывается словарем, если запрошенный ключ отсутствует (переопределен в defaultdict).

Наследование

Пример наследования

```
class Animal(object):
    pass

class Cat(Animal):
    pass

class Dog(Animal):
    pass


bob = Cat()
```

Проверка типа

Проверка типа с учетом наследования производится с помощью функции **isinstance**:

```
>>> isinstance(bob, Cat)   # True
>>> isinstance(bob, Animal)   # True
>>> isinstance(bob, Dog)   # False
>>> type(bob) is Animal   # False; type is Cat
```

Все объекты наследуются от **object**:

```
>>> isinstance(bob, object)   # True
```

Замечание. Метод **isinstance** вторым аргументом принимает также *tuple* допустимых типов.

Замечание. Для корректной проверки, является ли объект ***x*** целым числом, в Python 2.x следует использовать **isinstance**(*x*, (**int**, **long**)).

Иерархия наследования

Посмотреть иерархию наследования можно с помощью метода **mro**() или атрибута **__mro__**:

```
>>> Cat.mro()
[<class '__main__.Cat'>, <class '__main__.Animal'>, <class 'object'>]
```

Для проверки того, что некоторый класс является подклассом другого класса, используется функция **issubclass**:

```
>>> issubclass(Cat, Animal)  # True
```

Наследование методов и атрибутов

Наследование классов позволяет не переписывать некоторые общие для подклассов методы, оставив их в базовом классе.

```
>>> class A(object):
>>>     X = 1
>>>     def f(self):
>>>         print("Called A.f()", end=' ')
>>>
>>> class B(A):
>>>     pass
>>>
>>> b = B()
>>> b.f()
>>> print(b.X) # Called A.f() 1
```

Переопределение атрибутов

Поведение в дочернем классе, разумеется, можно переопределить.

```
>>> class A(object):
>>>     def f(self):
>>>         print("Called A.f()")
>>>
>>> class B(A):
>>>     def f(self):
>>>         print("Called B.f()")
>>>
>>> a, b  = A(), B()
>>> a.f()
>>> b.f()
Called A.f()
Called B.f()
```

Частичное переопределение атрибутов

```
>>> class A(object):
>>>     NAME = "A"
>>>
>>>     def f(self):
>>>         print(self.NAME)
>>>
>>> class B(A):
>>>     NAME = "B"
>>>
>>> a, b = A(), B()
>>> a.f()
>>> b.f()
A
B
```

Переопределение конструктора

```
>>> class A(object):
>>>     def __init__(self):
>>>         self.x = 1
>>>
>>> class B(A):
>>>     def __init__(self):
>>>         self.y = 2
>>>
>>> b = B()
>>> print(b.x)   # AttributeError
>>> print(b.y)   # 2
```

Вызов методов базового класса

Конструктор базового класса также должен был быть вызван. К методам базовых классов можно обращаться как:

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

Второй вариант нежелателен, однако порой необходим при ***множественном*** наследовании.

Замечание. В Python 3 метод **super** внутри класса можно вызывать без параметров – будут использованы значения по умолчанию.

Замечание. Метод **super** возвращает специальный proxy-объект, а потому обратиться к некоторым «магическим» методам. Например, обратиться по индексу к нему невозможно.

Вызов конструктора базового класса

```
>>> class A(object):
>>>     def __init__(self):
>>>         self.x = 1
>>>
>>> class B(A):
>>>     def __init__(self):
>>>         super(B, self).__init__()
>>>         # A.__init__(self) - второй нежелательный вариант
>>>         self.y = 2
>>>
>>> b = B()
>>> print b.x, b.y
1 2
```

Множественное наследование

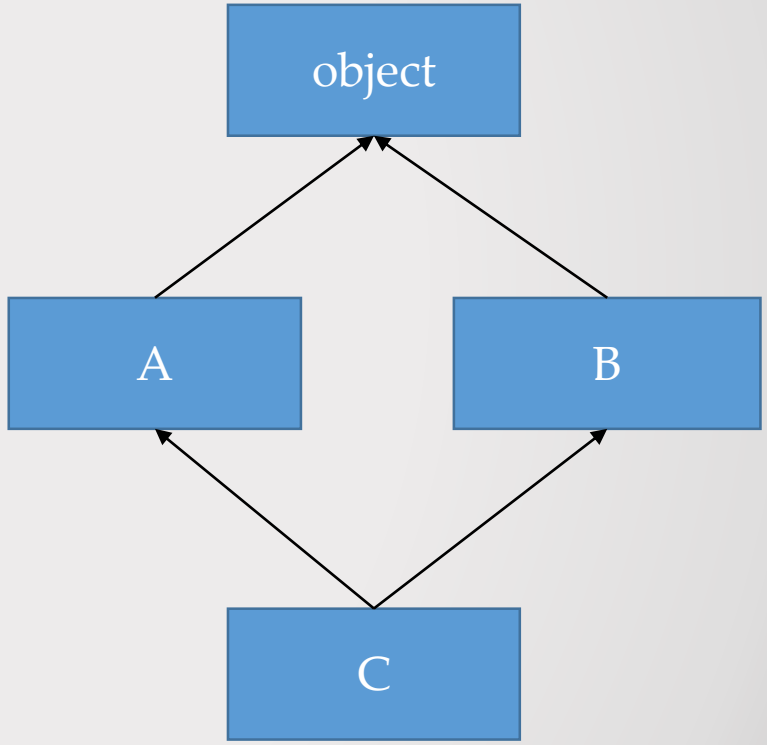
...

Множественное наследование

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

Наиболее распространенные виды:

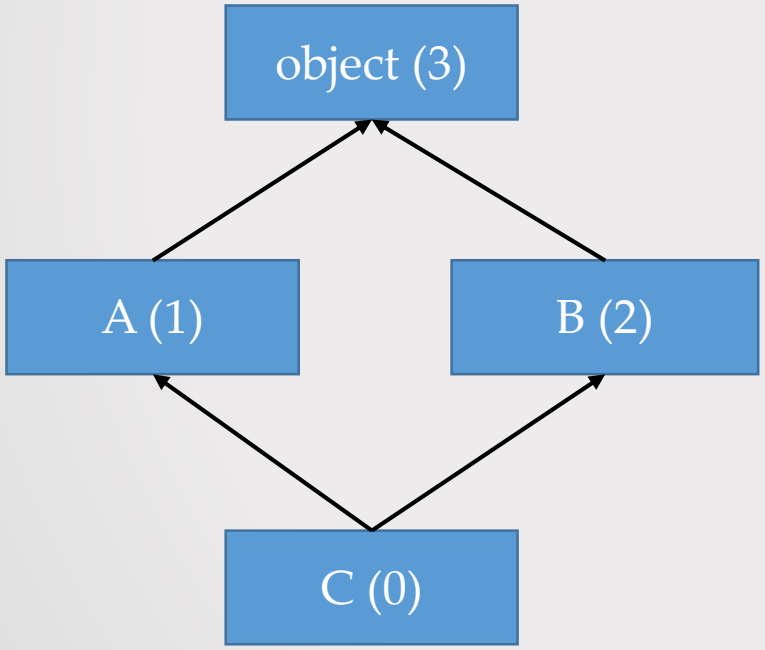
- ромбовидное наследование
- добавление Mixin-классов (реализация некоторого функционала , выраженная через другие методы).

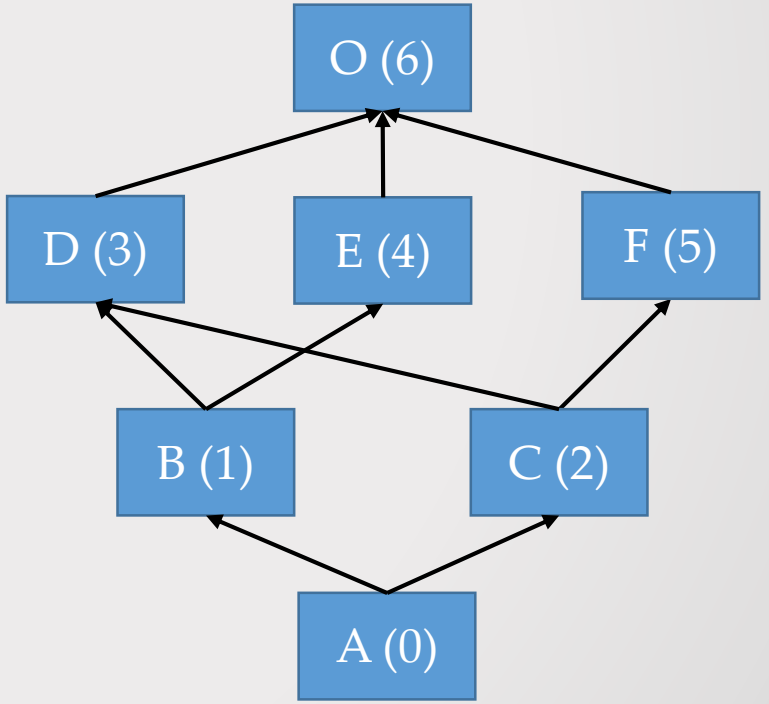



```
class A(object):
    pass

class B(object):
    pass

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

Порядок разрешения имен

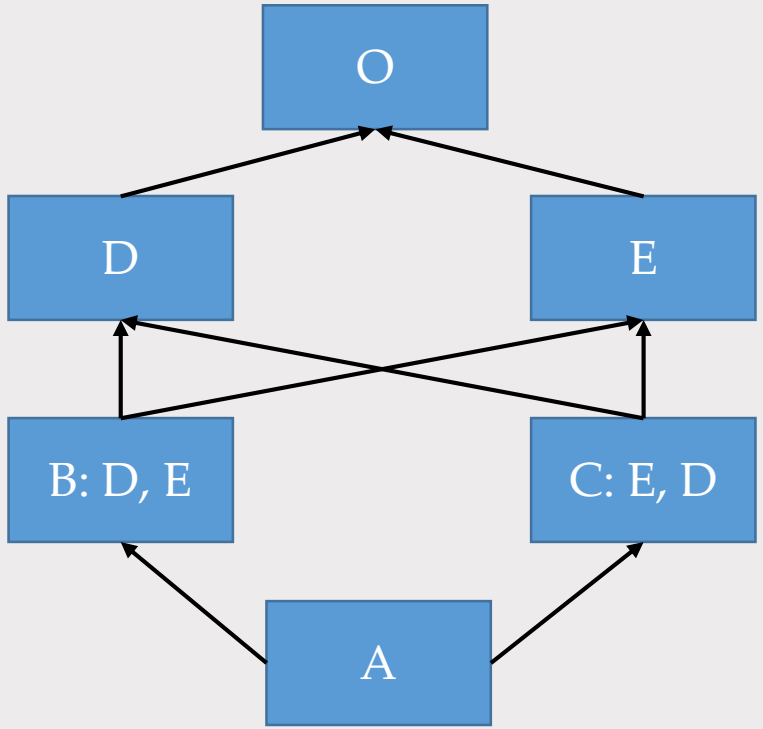
Если атрибут отсутствует в классе, предпринимается попытка найти его в базовых классах согласно MRO (Method resolution order). Алгоритм поиска – C3-линеаризация.





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

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)
```

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

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.

Замечание. Изменять атрибуты можно (*не рекомендуется*), модифицируя **__dict__**.

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().__setattr__(name, value)

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

Пример реализации методов работы с атрибутами

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

Напоминание. Обратите внимание на вызовы методов базовых классов:

- `super(Proxy, self).__getattribute__(name, value)`
- `object.__getattribute__(self, name)`

Такие вызовы (в случае одинаковых методов, разумеется) эквивалентны, как через **`super(...)`**, так и через `<base class>.method_name`.

Замечания по работе с атрибутами

Если *Proxy* не нужно добавлять атрибуты, то можно было ограничиться переопределением метода **__getattr__** и реализацией конструктора:

```
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**.

Декораторы

Декораторы

Декораторы (PEP-318):

- Выполняют некоторое дополнительное действие при вызове или создании функции
- Модифицируют функцию после создания
- Могут принимать аргументы
- Упрощают написание кода

Рассмотрим пример декоратора – функции, которая при вызове декорированной функции проверяет, возвращенное ей значение имеет тип *float*. Функцию-декоратор назовем *check_return_type_float*.

Пример реализации

Пример использования (проверяет, что возвращаемое значение типа *float*):

```
>>> @check_return_type_float
>>> def g():
>>>     return 'not a float'
```

Эквивалентной записью будет следующая:

```
>>> def g():
>>>     return 'not a float'
>>>
>>> g = check_return_type_float(g)
```

Пример реализации

Декоратор реализован как функция, которая возвращает другую функцию – *wrapper*:

```
>>> def check_return_type_float(f):
>>>     def wrapper(*args, **kwargs):
>>>         result = f(*args, **kwargs)
>>>         assert isinstance(result, float)
>>>         return result
>>>     return wrapper
```

Нюансы декорирования

Поскольку декоратор возвращает другую функцию, в примере *check_return_type_float* у переменной **f** будет имя '*wrapper*'.

Для того, чтобы метаданные (имя, документация) были корректными, внутренней функции (wrapper'у) добавляют декоратор **functools.wraps**:

```
>>> def check_return_type_float(f):
>>>     @functools.wraps(f)
>>>     def wrapper(*args, **kwargs):
>>>         # some code
>>>     return wrapper
```

Замечание. В Python 3 декорировать можно не только функции, но и классы (PEP-3129).

Реализация декоратора с помощью класса

```
>>> class FloatTypeChecker(object):
>>>     def __init__(self, f):
>>>         self.f = f
>>>
>>>     def __call__(self, *args, **kwargs):
>>>         result = self.f(*args, **kwargs)
>>>         assert isinstance(result, float)
>>>         return result
>>>
>>> check_return_type_float = FloatTypeChecker
```

Вопрос: Как применить здесь **functools.wraps**?

Замечание. Добавлять данный декоратор желательно везде. Это и хороший стиль кода, и так остается возможность узнать имя вызванной функции run-time.

Декораторы с параметрами

Для создания более общего декоратора, логично ему добавить возможность принимать параметры.

```
>>> def check_return_type(type_):
>>>     def wrapper(f):
>>>         @functools.wraps(f)
>>>         def wrapped(*args, **kwargs):
>>>             result = f(*args, **kwargs)
>>>             assert isinstance(result, type_)
>>>             return result
>>>         return wrapped
>>>     return wrapper
>>>
>>> @check_return_type(float)
>>> def g():
>>>    return 'not a float'
```

Несколько декораторов

Эквивалентной записью будет следующая:

```
>>> def g():
>>>     return 'not a float'
>>>
>>> g = check_return_type(float)(g)
```

Допустимо применять несколько декораторов – один над другим.

```
>>> @decorator2
>>> @decorator1
>>> def f():
>>>     pass
```

Эквивалентная запись применения декораторов к функции **f** имеет вид:

```
>>> f = decorator2(decorator1(f))
```

Параметризованный декоратор-класс

Декораторы с параметрами можно реализовать как класс. В таком случае параметры будут сохраняться в методе ***__init__***, а декорированную функцию следует возвращать в ***__call__***.

```
>>> class FloatTypeChecker(object):
>>>     def __init__(self, result_type):
>>>         self._result_type = result_type
>>>
>>>     def __call__(self, f):
>>>         @functools.wraps(f)
>>>         def wrapper(*args, **kwargs):
>>>             result = f(*args, **kwargs)
>>>             assert isinstance(result, self._result_type)
>>>             return result
>>>         return wrapper
```

Свойства (пример)

```
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

Свойства – это *дескрипторы*, которые можно создать, декорируя методы с помощью **property** (docstring свойства получается из getter'а):

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

Полный синтаксис декоратора-дескриптора **property** имеет вид:

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

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

Ответы и полезные ссылки

Ответы на вопросы

Аргументы функций

Если функция имеет неименованный variadic параметр (Python 3), но variadic аргументы переданы, то произойдет исключение **TypeError**.

Lambda-функция

Пустая lambda-функция имеет следующий вид:

```
>>> lambda: None
```

Ответы на вопросы

- При создании декораторов-классов для применения **functools.wraps** обычно переопределяют метод ***__new__***. Применить напрямую к классу его не удастся. Второй вариант – применить **functools.update_wrapper** в методе ***__init__*** ко вновь созданному объекту класса.

Полезные ссылки

- Множество *shortcuts* можно найти в книге Pilgrim, M. Dive Into Python 3.
- C3-линеаризация (цепочка поиска метода среди предков): https://en.wikipedia.org/wiki/C3_linearization