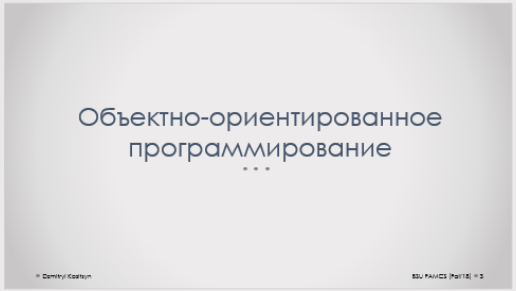
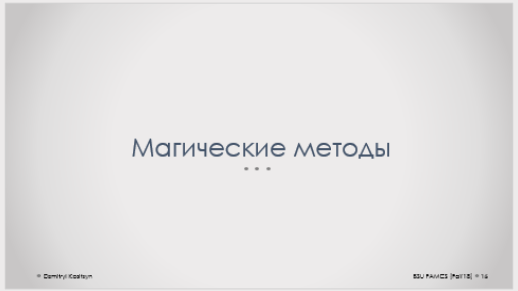
# Python

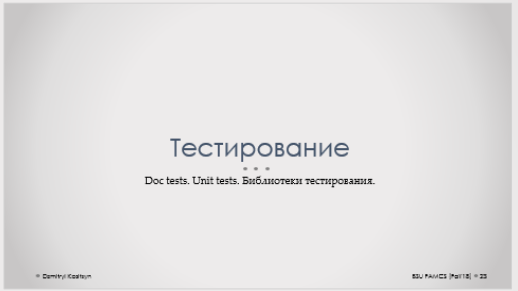
*Лекция 3*

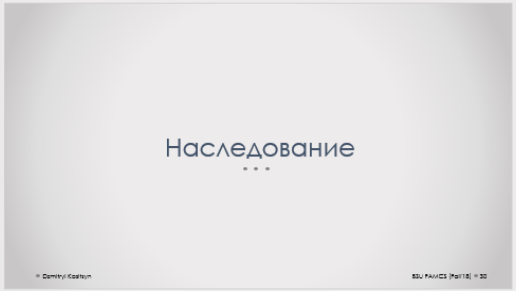
Преподаватель: *Дмитрий Косицин*

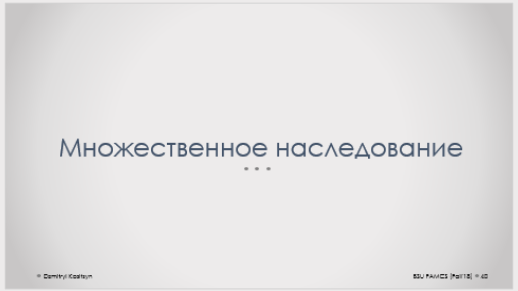
BSU FAMCS (Fall'18)

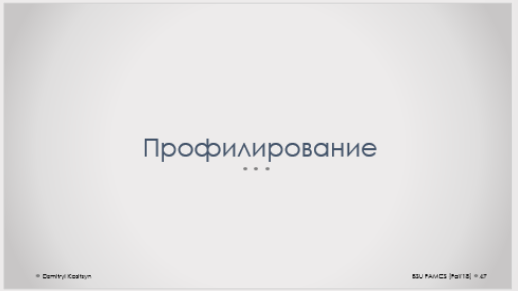

Объектно-ориентированное программмирование
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 3

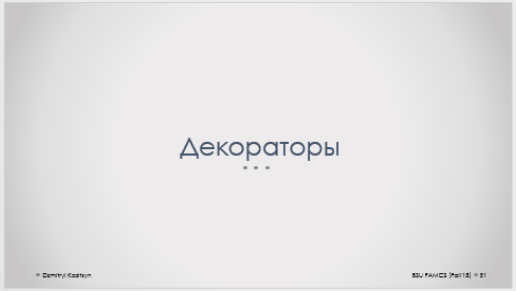

Магические методы
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 16

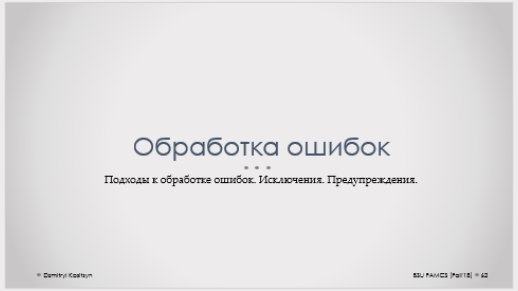

Тестирование
Doc tests. Unit tests. Библиотеки тестирования.
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 23

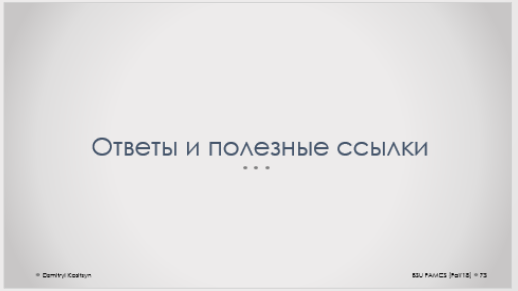

Наследование
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 30


Множественное наследование
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 40


Профилирование
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 47


Декораторы
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 51


Обработка ошибок
Подходы к обработке ошибок. Исключения. Предупреждения.
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 62


Ответы и полезные ссылки
Dzmitryi Kasitsyn
BSU FAMCS (Fall'18) ● 73

# 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***).

# 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)
```

# Магические методы

...

# Магические методы

Методы со специальными именами вида __*__ называют магическими. 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).

# Тестирование

Doc tests. Unit tests. Библиотеки тестирования.

# Проверка аргументов

**assert** – statement, позволяющий проверить на истинность некоторое выражение

```
def calculate_binomial_mean(n, p):
    assert n > 0, 'number of experiments must be positive'
    assert 0 <= p <= 1, 'probability must be in [0; 1]'
    return n * p
```

**Важно**! Скобки **assert** имеют смысл проверки кортежа на пустоту.

# Юнит-тесты

**Общая идея юнит-тестов**

- Разбить код на независимые части (юниты)
- Тестировать каждую часть отдельно

**Преимущества**

- Нужно меньше тестов
- Проще отлаживать

**Недостатки**

- Нужны тесты, проверяющие взаимодействие юнитов

# doctest

Библиотека **doctest** (Py2, Py3) позволяет расположить тест непосредственно в документации, чтобы и показать , и проверить, как работает функция.

Недостаток подхода в его сложности: проверка некорректных входных данных или результатов сложных типов трудоемка.

```
def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> factorial(5)
    120
    """
    pass  # implementation is here

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

# unittest

Тесты можно писать, используя библиотеку **unittest** (Py2, Py3), что позволяет, в частности, группировать тесты в test cases, а также использовать множество удобных проверок.

```
import unittest

class TestFactorial(unittest.TestCase):
    def test_simple(self):
        self.assertEqual(factorial(5), 120)

if __name__ == '__main__':
    unittest.main()
```

# Возможности unittest

- Проверка различных типов и видов возвращаемых значений: ***assertTrue***, ***assertIsNone***, ***assertAlmostEqual***, ***assertRaisesRegexp*** и др.
- Возможность подготовить тестирование: методы ***setup*** (***setUpClass***) и ***teardown*** (***tearDownClass***) – вызываются перед и после каждого запуска теста (класса test case), создавая и удаляя используемые объекты.
- Возможность пропустить тест по некоторому условию (см. unittest.SkipTest).

Также есть библиотека **pytest**, в которой все проверки можно проводить с помощью **assert** statement'ов, и библиотека **nose**, объединяющая все виды тестов.

# Подмена объектов

При помощи библиотеки **mock** (в стандартной библиотеке с Python 3.3, *примеры*) можно подменить любой объект, будь то поток ввода-вывода ***stdout***, некоторый модуль, класс, атрибут, свойство, метод.

```
from io import StringIO


class InterfaceTestCase(unittest.TestCase):
    def setUp(self):
        self._stdout_mock = self._setup_stdout_mock()

    def _setup_stdout_mock(self):
        patcher = mock.patch('sys.stdout', new=StringIO())
        patcher.start()
        self.addCleanup(patcher.stop)
        return patcher.new
```

# 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

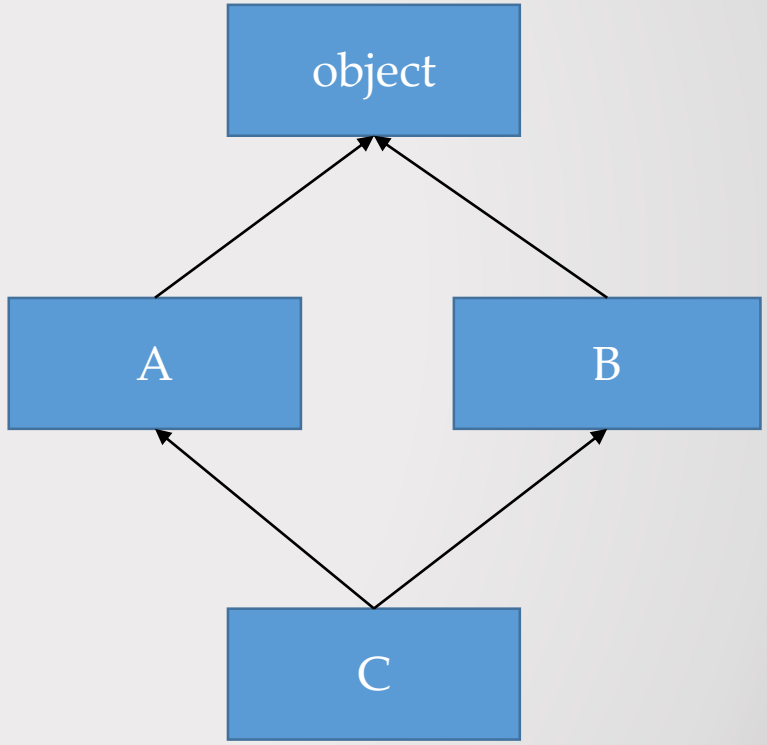
...

# Множественное наследование

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

Наиболее распространенные виды:

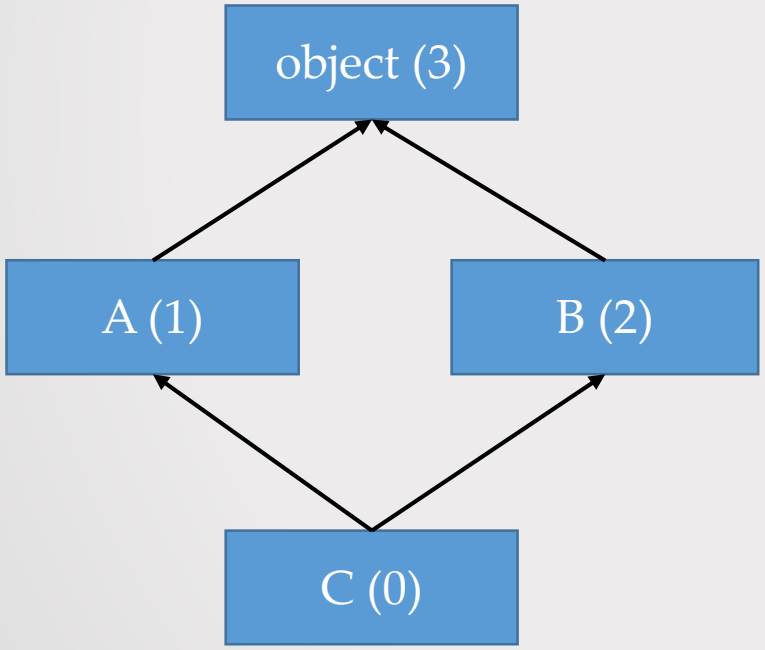
- ромбовидное наследование
- добавление Mixin-классов (реализация некоторого функционала , выраженная через другие методы).

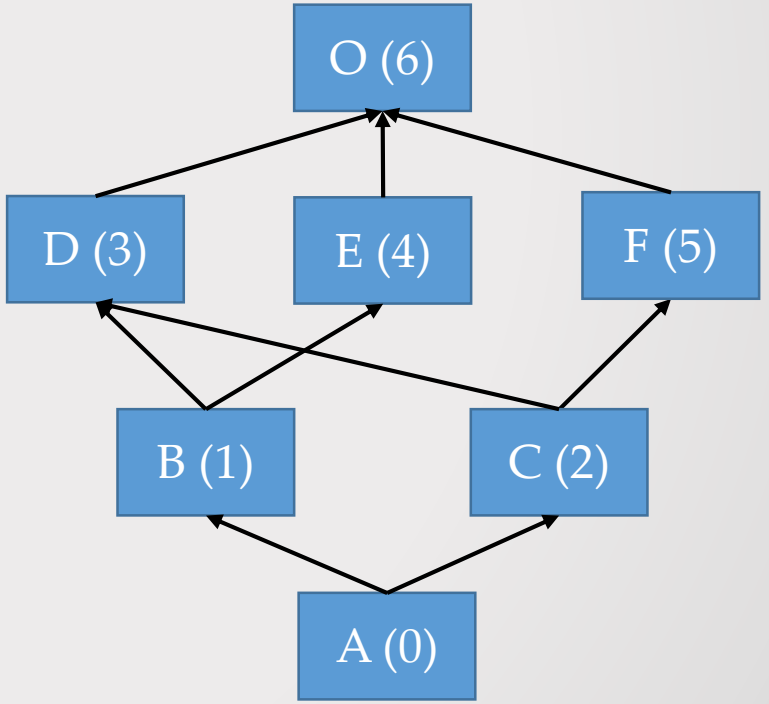



```
class A(object):
    pass

class B(object):
    pass

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

# Порядок разрешения имен

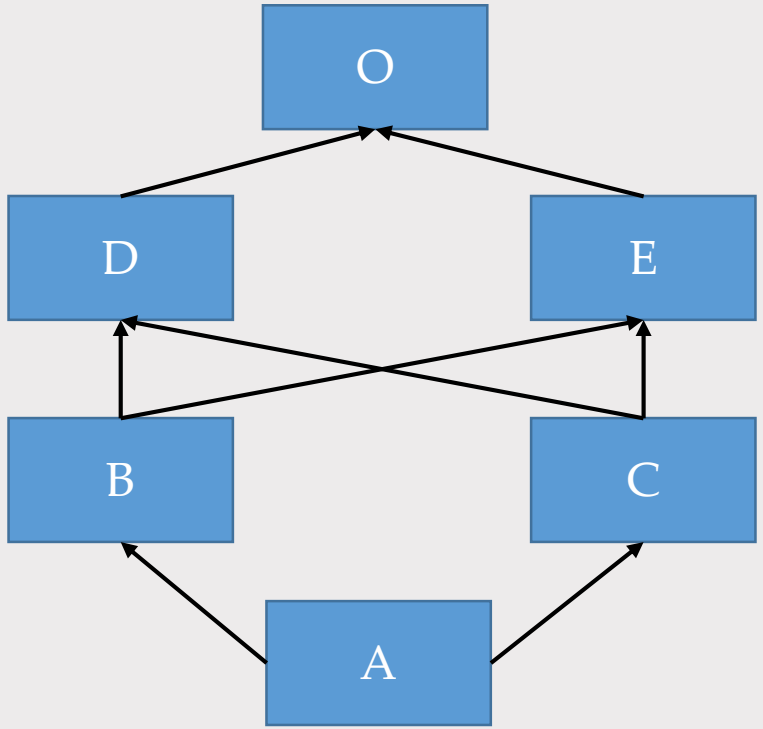
Если атрибут отсутствует в классе, предпринимается попытка найти его в базовых классах согласно MRO (Method resolution order). Алгоритм поиска – C3-линеаризация.



# 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 (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.

# Профилирование

# Замер времени исполнения

Для замера времени исполнения используйте модуль **timeit**.

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in
xrange(10000))', number=1000)
2.027786269876367 3
>>> timeit.timeit('"-".join(str(n) for n in range(10000))',
number=1000)
2.2692865730951 44
```

**Замечание**. Также доступна функция *repeat* (повторять эксперимент несколько раз).

# Профилирование

Для профилирования есть модули **cProfile** и **Profile**.

```
>>> import cProfile  # or Profile - pure Python implementation
>>> profiler = cProfile.Profile()
>>> profiler.run_call(calculate_binomial_mean, 10, 0.5)
# another way: run('calculate_binomial_mean(10, 0.5)')
>>> profiler.print_stats()
```

Вызов выведет статистику по времени выполнения функции, в том числе по всем вложенным (если есть).

# Библиотеки для профилирования

Библиотеки для профилирования:

- Просмотр времени выполнения каждой строки: line_profiler
- Использование памяти: memory_profiler и др.
- Визуализация профилирования: SnakeViz
- Прочие инструменты: ссылка

# 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**.

# 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 2

Если во время обработки исключения его нужно передать выше по стеку вызовов или бросить новое исключение, сохранив информацию о старом, можно использовать специальный синтаксис **raise**.

*Вопрос*: когда применим второй случай?

```
def process_exception(exc_type):
    try:
        raise exc_type()
    except ValueError:
        # some actions here
        exc_type, exc_instance, exc_traceback = sys.exc_info()
        # raise other exception with original traceback
        raise Exception, Exception(), exc_traceback
    except Exception:
        # some actions here
        raise  # re-raise the same exception
```

# 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__***.

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)
```

# Ответы и полезные ссылки

# 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