
# Углубленный Python

Лекция 2

Опрышко Александр


“

Не забудьте отметиться на занятии!

*Цитата великих*

# Лекция 1. Что было?


1. Типы с одним значение
2. Стандартные типы
3. Магические поля функций (методов)
4. Магические методы дескриптора
5. Методы доступа к атрибутам
6. Методы rich comparison
7. Методы эмуляции контейнеров
8. Методы эмуляции чисел

# Паноптикум rich comparison


**С чего начать копать?**

```
1.  >>> import dis
2.  >>> def c(a, b):
3.  ...     a == b
4.  ...
5.  >>> dis.dis(c)
6.   2           0 LOAD_FAST                0 (a)
7.               2 LOAD_FAST                1 (b)
8.               4 COMPARE_OP               2 (==)
9.               6 POP_TOP
10.              8 LOAD_CONST               0 (None)
11.              10 RETURN_VALUE
12. >>>
13.
```

# Паноптикум rich comparison


1. https://github.com/python/cpython/blob/master/Python/ceval.c#L2723
2. https://github.com/python/cpython/blob/master/Objects/object.c#L734
3. https://github.com/python/cpython/blob/master/Objects/typeobject.c#L6942

# Разгадка rich comparison


https://github.com/python/cpython/blob/master/Include/object.h#L569

```
568
569  /* Rich comparison opcodes */
570  #define Py_LT 0
571  #define Py_LE 1
572  #define Py_EQ 2
573  #define Py_NE 3
574  #define Py_GT 4
575  #define Py_GE 5
576
673
674  /* Map rich comparison operators to their swapped version, e.g. LT <--> GT */
675  int _Py_SwappedOp[] = {Py_GT, Py_GE, Py_EQ, Py_NE, Py_LT, Py_LE};
676
677  static const char * const opstrings[] = {"<", "<=", "==", "!=", ">", ">="};
678
```

# Лекция 2. Что будет?


1. Метаклассы
2. MRO
3. ABC
4. Inspect

# Магические методы


## Кастомизация объектов

object.__**new**__(*cls*[, ...]) – создает новый объект класса, статический метод по преданию.

После создание объекта вызывается (уже у объекта) метод __**init**__. Он ничего не должен возвращать, иначе будет TypeError

>>> test_new.py

# Магические методы


## Кастомизация объектов

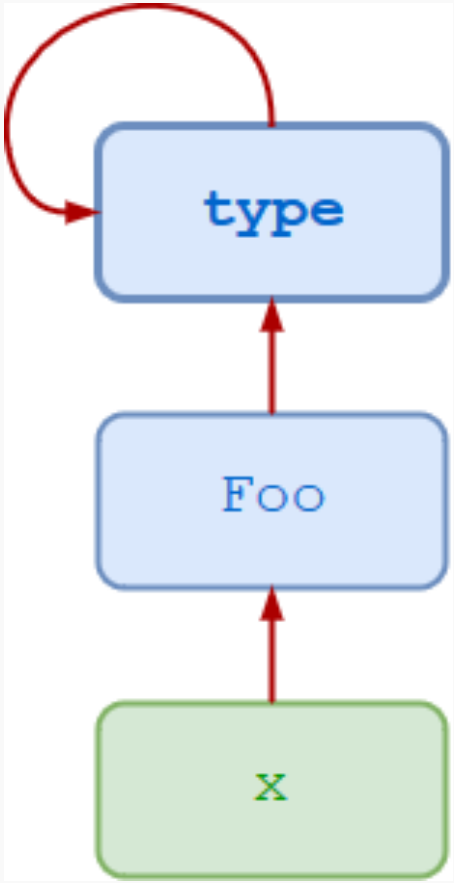
```
1.  >>> class A:
2.  ...     def __init__(self):
3.  ...         return 1
4.  ...
5.  >>> a = A()
6.  Traceback (most recent call last):
7.   File "<stdin>", line 1, in <module>
8.  TypeError: __init__() should return None, not 'int'
9.
```

# Магические методы


**Подумать?**

```
1.  >>> class Foo:
2.  ...     pass
3.  ...
4.  >>> x = Foo()
5.  >>> type(x)
6.  <class '__main__.Foo’>
7.  >>> type(Foo)
8.  ???
9.  >>> type(type)
    ???
```

# Магические методы



# Магические методы


## Метаклассы

Новые классы создаются с помощью вызова type(<name>, <bases>, <classdict>)

name – имя класса (__name__)

bases – базовые классы (__bases__)

classdict – namespace класса (__dict__)

# Магические методы


## Метаклассы

```
1.  >>> Bar = type('Bar', (Foo,), dict(attr=100))
2.  >>> x = Bar()
3.  >>> x.attr
4.  100
5.  >>> x.__class__
6.  <class '__main__.Bar’>
7.  >>> x.__class__.__bases__
8.  (<class '__main__.Foo'>,)

9.  >>> class Bar(Foo):
10. ...     attr = 100
11. ...
12. >>> x = Bar()
13. >>> x.attr
14. 100
15. >>> x.__class__.__bases__
16. (<class '__main__.Foo'>,)
```

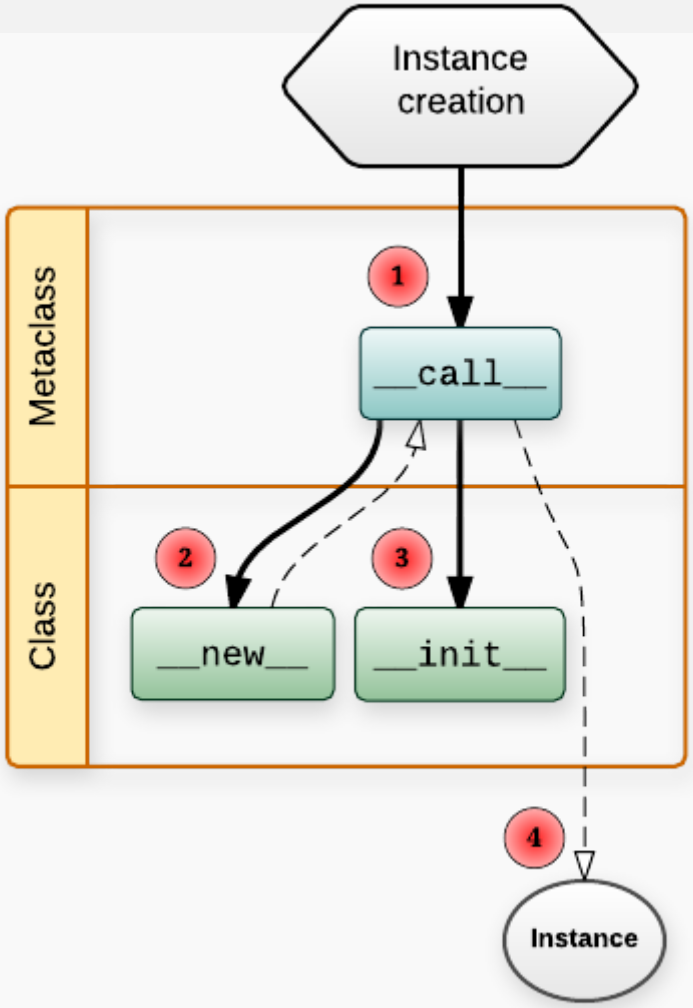
# Магические методы


## Кастомные метаклассы

```
1.  >>> class Meta(type):
2.  ...     Pass
3.  ...
4.  >>> class Foo(metaclass=Meta):
5.  ...     Pass
6.  ...
7.  >>> Foo()
```

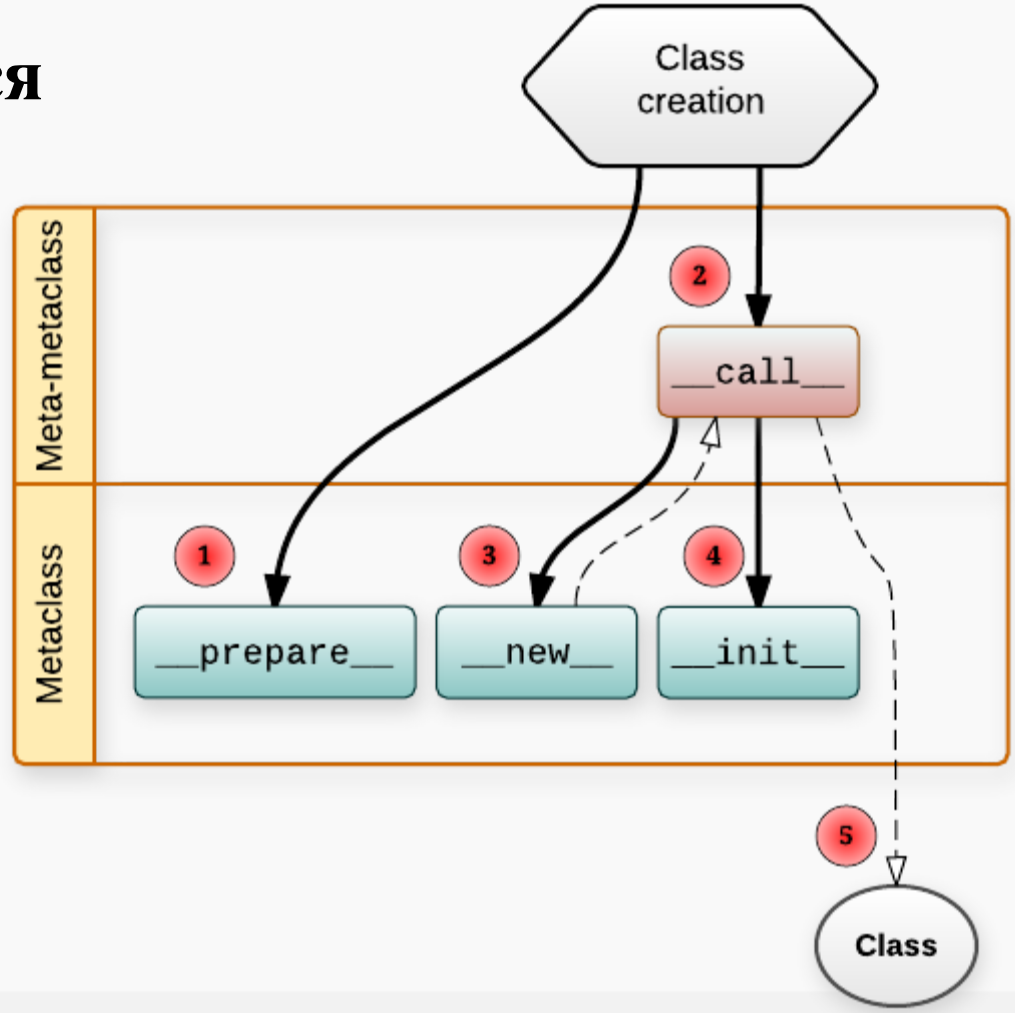
# Магические методы

## Повторим как происходит создание объекта



# Магические методы

## Как создается класс?



# Магические методы


## Как создается класс?

- определяются базовые классы
- определяется метакласс
- подготавливается namespace класса (__prepare__)
- выполняется тело класса
- создается класс (__new__, __init__)

>>> test_meta.py

>>> test_meta_example.py

# Магические методы


## Другие методы кастомизации классов

__init_subclass__

>>> test_init_subclass.py

class decorators

>>> test_ class_decorators.py

# MRO


Порядок разрешения методов (method resolution order) позволяет python выяснить, из какого класса-предка нужно вызывать метод, если он не обнаружен непосредственно в классе-потомке.

.__mro__

.mro()

# MRO до python 2.2


```
1.     A   C
2.     |   |
3.     B   D
4.      \ /
5.       E

6.  # E, B, A, D и C
```

# MRO c python 2.2


```
1.    object
2.     /   \
3.    A     B
4.     \   /
5.       C

6.   # C, A, object, B
```

# Проблема «ромбовидной структуры»


Если у нас есть классы A и B, от которых наследуется класс C, то при поиске метода по старому алгоритму получается, что если метод не определён в классах C и A он будет извлечён из object, даже если он определён в B.

Упорядоченный список классов, в которых будет производиться поиск метода слева направо будем называть **линеаризацией** класса.

# Решение

Линеаризация должна быть **монотонной**.

Если в линеаризации некого класса C класс A следует за классом B (она имеет вид [C, …, B, …, A]) и для любого его потомка D класс B будет следовать за A в его линеаризации (она будет иметь вид [D, …, C, …, B, …, A]), то линеаризация будет **монотонной.**

L[A] = [A, object]; L[B] = [B, object]

L[C] = [C, A, object, B] => L[C] – не удовлетворяет условию

# Локальный порядок старшинства


Линеаризация, которые удовлетворяют свойству монотонности:

L[C] = [C, A, B, object]

L[C] = [C, B, A, object]

Какой выбрать?

Определяется **порядок локального старшинства** — это свойство, которое требует соблюдения в линеаризации класса-потомка того же порядка следования классов-родителей, что и в его объявлении.

# Локальный порядок старшинства


```
1.  >>> class A:
2.  ...     pass
3.  ...
4.  >>> class B:
5.  ...     pass
6.  ...
7.  >>> class C(A, B):
8.  ...     pass
9.  ...
10. >>> C.mro()
11. [<class '__main__.C'>, <class '__main__.A'>, <class
    '__main__.B'>, <class 'object'>]
12. >>>
13. >>> class C(B, A):
14. ...     pass
15. ...
16. >>> C.mro()
17. [<class '__main__.C'>, <class '__main__.B'>, <class
    '__main__.A'>, <class 'object'>]
```

# C3

The head of the list is its first element:

**head** = C1

whereas the tail is the rest of the list:

**tail** = C2 ... CN.

The linearization of C is the sum of C plus the merge of the linearizations of the parents and the list of the parents.

*L[C(B1 ... BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)*

*L[object] = object.*

# C3 Merge


Take the **head** of the first list, i.e L[B1][0];

If this **head** is not in the **tail** of any of the other lists, then add it to the linearization of C and remove it from the lists in the merge, otherwise look at the **head** of the **next list** and take it, if it is a good head.

Then repeat the operation until all the class are removed or it is impossible to find good heads. In this case, it is impossible to construct the merge, Python will refuse to create the class C and will raise an exception.

>>> test_mro.py

# ABC


Модуль, который позволяет определять абстрактные базовые классы (abstract base classes).

# ABC Example


## Hashable

```
1.  class Hashable(metaclass=ABCMeta):
2.      __slots__ = ()
3.
4.      @abstractmethod
5.      def __hash__(self):
6.          return 0
7.
8.      @classmethod
9.      def __subclasshook__(cls, C):
10.         if cls is Hashable:
11.             return _check_methods(C, "__hash__")
12.
13.         return NotImplemented
```

# ABC


1. **abstractmethod**
2. **abstractclassmethod** – *deprecated*
3. **abstractstaticmethod** – *deprecated*
4. **abstractproperty** – *deprecated*

>>> test_abc.py

# Inspect


Модуль, который предоставляет пачку полезных функций для получения информации об объектах в python

# Inspect


inspect.**getmembers**

Return all the members of an object in a list of (name, value) pairs sorted by name.

# Inspect


**Подумать?**

```
1.  >>> import inspect
2.  >>> class B:
3.  ...     def __le__(self, other):
4.  ...         print('__le__')
5.  ...         return False
6.  ...
7.  >>> inspect.getmembers(B)
8.  [('__class__', <class 'type'>), ..., ('__le__', <function
    B.__le__ at 0x101779510>), ('__lt__', <slot wrapper '__lt__' of
    'object' objects>), ...]
9.
10. >>> inspect.getmembers(B())
11. [('__class__', <class '__main__.B'>), ..., ('__le__', <bound
    method B.__le__ of <__main__.B object at 0x10169e8d0>>),
    ('__lt__', <method-wrapper '__lt__' of B object at
    0x10169e8d0>), ...]
12.
```

# Inspect


## Source code

- inspect.**getdoc**
- inspect.**getfile**
- inspect.**getmodule**
- inspect.**getsourcefile**
- inspect.**getsource**

# Inspect


inspect.**signature**

```
1.  >>> def foo(a, *, b:int, **kwargs):
2.  ...    pass
3.
    >>> sig = signature(foo)
4.
    >>> str(sig)
5.  '(a, *, b:int, **kwargs)'
6.
    >>> str(sig.parameters['b'])
7.  'b:int'
8.
    >>> sig.parameters['b'].annotation
9.  <class 'int'>
10.
```

# Inspect


## Стек интерпретатора

- inspect.**currentframe**
- inspect.**stack**

>>> test_inspect.py

https://github.com/pallets/jinja/blob/master/jinja2/debug.py

https://github.com/getsentry/raven-python

# Пишем ORM

# Итого


1. Метаклассы
2. MRO
3. ABC
4. Inspect

# Домашнее задание № 1


Нужно написать ORM для реляционной базы (MySQL, PostgreSQL)

https://en.wikipedia.org/wiki/Active_record_pattern

https://en.wikipedia.org/wiki/Data_mapper_pattern

https://medium.com/oceanize-geeks/the-active-record-and-data-mappers-of-orm-pattern-eefb8262b7bb

**Срок сдачи**

29.03.2019

# Домашнее задание № 1


Нужно реализовать CRUD:

- Метод создания .create
- Метод извлечения данных .all + .get
- Метод обновления .update()
- Метод удаления .delete()

**Срок сдачи**

29.03.2019

# Ссылки


1. https://github.com/python/cpython/
2. https://docs.python.org/3/reference/datamodel.html
3. https://www.python.org/download/releases/2.3/mro/
4. https://habr.com/ru/post/62203/
5. https://docs.python.org/3/library/abc.html
6. https://docs.python.org/3/library/inspect.html


**telegram: alexopryshko**
**email: alexopryshko@gmail.com**

**telegram: igorcoding**
**email: igor.latkin@outlook.com**

**Спасибо за внимание!**