Advanced Python

Магические методы

- Магические методы – это внутренние методы классов, такие как __init__

- Магические методы используются для:
  - Конструирования и инициализации
  - Переопределения операторов
  - Представления своих классов
  - Контроля доступа к атрибутам
  - Магии контейнеров
  - И еще много чего
- Для магического метода обычно есть функция, которая его вызывает

Магические методы сравнения

- __eq__(self, other) *# == other.*
- __ne__(self, other) *# != other.*
- __lt__(self, other) *# < other.*
- __gt__(self, other) *# > other.*
- __le__(self, other) *# <= other.*
- __ge__(self, other) *# >= other.*

@functools.total_ordering позволяет не определять все 6 методов, ограничиваясь __eq__ и еще одним

- __add__(self, other) *# Сложение. instance + other*
- __radd__(self, other) *# Отражённое сложение. other + instance*
- __iadd__(self, other) *# Сложение с присваиванием. intance += other*

```
import functools

@functools.total_ordering
class OrderableBus(EmptyBus):
    def __eq__(self, other):
        return self.passangers == other.passangers

    def __lt__(self, other):
        return self.passangers < other.passangers
```

```
a = OrderableBus(4)
b = OrderableBus(7)
```

```
a > b
```

```
False
```

```
a <= b
```

```
True
```

Магические методы

- __str__(self)
  Определяет поведение функции str(), вызванной для экземпляра вашего класса.

- __repr__(self)
  Определяет поведение функции repr(), вызыванной для экземпляра вашего класса. Главное отличие от str() в целевой аудитории. repr() больше предназначен для машинно-ориентированного вывода (более того, это часто должен быть валидный код на Питоне), а str()предназначен для чтения людьми.

```
class SmileyFaces:
    def __init__(self, num):
        self.num = num

    def __repr__(self):
        return f'SmileyFaces({self.num})'

    def __str__(self):
        return ' '.join([':D'] * self.num)
```

```
c = SmileyFaces(5)
```

```
print(c)
```

```
:D :D :D :D :D
```

```
repr(c)
```

```
'SmileyFaces(5)'
```

Магические методы

Метод __bool__ для проверки значения на истинность, например в условии оператора if.

```
class EmptyBus:
    def __init__(self, passangers):
        self.passangers = passangers

    def __bool__(self):
        return not bool(self.passangers)
```

```
bus = EmptyBus(4)
```

```
bool(bus)
```

```
False
```

Магические методы

Вы можете определить поведение для случая, когда пользователь пытается обратиться к атрибуту, который не существует, с помощью магического метода __getattr__. Это может быть полезным для перехвата и перенаправления частых опечаток, предупреждения об использовании устаревших атрибутов.

```
class MyClass:
    def __init__(self):
        self.foo = 42

    def __getattr__(self, name):
        print(f'Attribute {name} does not exist(')
        print("My attributes:")
        print([x for x in self.__dict__])
```

```
c = MyClass()
```

```
c.bar
```

```
Attribute bar does not exist(
My attributes:
['foo']
```

Магические методы

- Методы __setattr__ и __delattr__ позволяют управлять изменением значения и удалением атрибутов.
- В отличие от __getattr__ они вызываются для всех атрибутов, а не только для несуществующих.
- Функция getattr позволяет безопасно получить атрибут экземпляра класса
- setattr, delattr удобны в том случае, когда мы получаем имя аттрибута в виде строки

```
class Censured:
    prohibited = []

    def __setattr__(self, name, value):
        assert name not in self.prohibited, 'java is banned'
        super().__setattr__(name, value)
```

```
class PythonClass(Censured):
    prohibited = ['java']
```

```
c = PythonClass()
```

```
c.java = 42
```

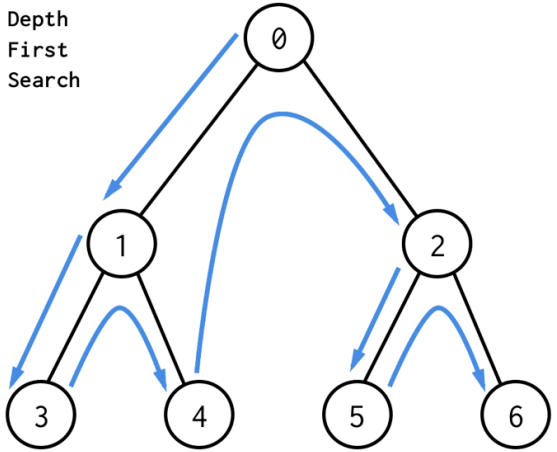
```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-65-77dfd2c89a95> in <module>()
----> 1 c.java = 42

<ipython-input-62-e058e1a7b323> in __setattr__(self, name, value)
      3
      4     def __setattr__(self, name, value):
----> 5         assert name not in self.prohibited, 'java is banned'
      6         super().__setattr__(name, value)

AssertionError: java is banned
```

Итераторы

- Итератор дает возможность последовательно перебрать все элементы коллекции без раскрытия внутреннего устройства



Итераторы

- Протокол итераторов состоит из двух методов:
  - Метод__iter__возвращает экземпляр класса, реализующего протокол итераторов, например, self.
  - Метод__next__возвращает следующий по порядку элемент итератора. Если такого элемента нет, то метод должен поднять исключение StopIteration.
- Важный инвариант метода __next__: если метод поднял исключение StopIteration, то все последующие вызовы метода __next__ тоже должны поднимать исключение.

```
import collections

class ListIterator(collections.abc.Iterator):
    def __init__(self, collection, cursor):
        self._collection = collection
        self._cursor = cursor

    def __next__(self):
        if self._cursor + 1 >= len(self._collection):
            raise StopIteration()
        self._cursor += 1
        return self._collection[self._cursor]

class ListCollection(collections.abc.Iterable):
    def __init__(self, collection):
        self._collection = collection

    def __iter__(self):
        return ListIterator(self._collection, -1)
```

```
c = iter(ListCollection([1, 2, 3, 5, 8]))
```

```
next(c)
```

```
1
```

```
next(c)
```

```
2
```

```
next(c)
```

```
3
```

Генераторы

- Генератор – это функция, которая использует оператор yield
- Генератор является итератором
- В результате работы оператора yield действие функции приостанавливается(сохраняя текущее состояние) до вызова next()
- Не переиспользуйте генераторы!

```
def devisors(x):
    for i in range(1, x + 1):
        if x % i == 0:
            yield i
```

```
d = devisors(10)
list(d)
```

```
[1, 2, 5, 10]
```

```
list(d)
```

```
[]
```

Генераторы-выражения

- (i ** 2 for i in range(100))
  1, 4, 9, …

```
even = (i for i in range(1, 1000000) if i % 2 == 0)
```

```
next(even)
```

```
2
```

```
next(even)
```

```
4
```

- Генераторы списков, словарей

```
my_dict = {'foo1': 'bar1', 'foo2': 'bar2'}
{val: key for key, val in my_dict.items()}
```

```
{'bar1': 'foo1', 'bar2': 'foo2'}
```

Генераторы

Метод send возобновляет исполнение генератора и отправляет аргумент в yield

Функция next отправляет в генератор None

Результат send – следующее значение генератора или StopIteration

Метод close поднимает специальное исключение **GeneratorExit** в месте, где генератор приостановил исполнение:

Если всё хорошо, то метод close завершает работу генератора и ничего не возвращает.

```
def check_div(divisor):
    print(f'checking division by {divisor}')
    res = yield
    while True:
        res = yield res % divisor == 0
```

```
h = check_div(3)
```

```
h.send(None) # = next(h)
```

```
checking division by 3
```

```
h.send(15)
```

```
True
```

```
h.send(4)
```

```
False
```

```
h.send(33)
```

```
True
```

```
h.close()
```

Менеджеры контекста

Менеджеры контекста используются для управления ресурсами:

```
r = setup_resource()
try:
    use_resource()
finally:
    release_resource()
```

Конструкция с with позволяет неявно делать release_resource()

```
with setup_resource() as r:
    use_resource()
```

Менеджеры контекста

Экземпляр любого класса, реализующего магические методы __enter__ и __exit__ является менеджером контекста.

Метод __enter__ инициализирует контекст (setup_resource), например открывает файл. Если он что-то возвращает, этот объект запишется в переменную после оператора as.

__exit__(self, exception_type, exception_value, traceback)
Определяет действия менеджера контекста после того, как блок будет выполнен (или прерван). Используется для контроллирования исключений, чистки, любых действий которые должны быть выполнены незамедлительно после блока внутри with.

```
class Closer:

    def __init__(self, obj):
        self.obj = obj

    def __enter__(self):
        return self.obj

    def __exit__(self, exception_type, exception_val, trace):
        try:
            self.obj.close()
        except AttributeError: # у объекта нет метода close
            print('Not closable.')
            return True # исключение перехвачено
```

Дескрипторы

Дескриптор - это класс, позволяющий управлять логикой получения изменения и удаления атрибута. Для этого он должен определять хотя бы один из методов:

- __get__
- __set__
- __delete__

Если реализован хотя бы __set__, то это data-дескриптор

```
class ValidDivisor:
    label = '_divisor'
    def __get__(self, instance, owner):
        return getattr(instance, self.label, self)

    def __set__(self, instance, value):
        assert value != 0, "Can't divide by Zero"
        setattr(instance, self.label, value)
```

```
class A:
    x = ValidDivisor()
```

```
a = A()
a.x = 0
```

Дескрипторы

- __get__(self, instance, owner)
  Определяет поведение при возвращении значения из дескриптора. instance это объект, для чьего атрибута-дескриптора вызывается метод. owner это тип (класс) объекта.
- __set__(self, instance, value)
  Определяет поведение при изменении значения из дескриптора. instance это объект, для чьего атрибута-дескриптора вызывается метод. value это значение для установки в дескриптор.
- __delete__(self, instance)
  Определяет поведение для удаления значения из дескриптора. instance это объект, владеющий дескриптором.

```
class Descr:
    def __get__(self, instance, owner):
        print(instance, owner)
        return
```

```
class A:
    attr = Descr()

class B(A):
    pass

A.attr
```

```
None <class '__main__.A'>
```

```
A().attr
```

```
<__main__.A object at 0x102c94908> <class '__main__.A'>
```

```
B.attr
```

```
None <class '__main__.B'>
```

```
B().attr
```

```
<__main__.B object at 0x102c87b70> <class '__main__.B'>
```

Отладка и тестирование

Assert

Не нужно никаких дополнительных средств
Просто и понятно

Сообщения об ошибках мало информативны

```
def divide(a, b):
    assert b != 0, 'Can\'t divide by zero'
    return a / b
```

```
divide(3, 0)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-13-9684ee019cde> in <module>()
----> 1 divide(3, 0)

<ipython-input-12-2c7a137836ef> in divide(a, b)
      1 def divide(a, b):
----> 2     assert b != 0, 'Can\'t divide by zero'
      3     return a / b

AssertionError: Can't divide by zero
```

Отладка и тестирование

Pytest

Удобный assert
Полезное сообщение об ошибке

```
def test_range():
    assert {0, 1, 2, 3} == set(range(5))
```

```
def test_range():
>       assert {0, 1, 2, 3} == set(range(5))
E       assert {0, 1, 2, 3} == {0, 1, 2, 3, 4}
E         Extra items in the right set:
E         4
E         Full diff:
E         - {0, 1, 2, 3}
E         + {0, 1, 2, 3, 4}
E         ?            +++

test_example.py:13: AssertionError
```

Отладка и тестирование

Возможность работать с исключениями

```
def test_division_by_zero():
    with pytest.raises(ZeroDivisionError):
        print(42 / 5)

test_example.py:15 (test_division_by_zero)
def test_division_by_zero():
        with pytest.raises(ZeroDivisionError):
>           print(42 / 5)
E           Failed: DID NOT RAISE <class 'ZeroDivisionError'>

test_example.py:18: Failed
```

Отладка и тестирование

Работа с фикстурами

Фикстура - это функция или метод, которая выпоняется до (и после) теста и подготавливает для него окружение по аналогии с контекстными менеджерами.

```
@pytest.fixture()
def resource():
    resource = resource_setup()
    yield resource
    resource_teardown()

def test_resource_usage(resource):
    assert resource is not None
```

Отладка и тестирование

- Параметризация тестов и фикстур

```
@pytest.mark.paramertize('a, b, expected', [
    (1, 2, 3),
    (3, 0, 3),
    (2, 2, 4),
    (3, 3, 5)
])
def test_sum(a, b, expected):
    assert a + b == expected
```

```
test_example.py::test_test[1-2-3] PASSED                    [ 25%]
test_example.py::test_test[3-0-3] PASSED                    [ 50%]
test_example.py::test_test[2-2-4] PASSED                    [ 75%]
test_example.py::test_test[3-3-5] FAILED                    [100%]
```