

Лекция №4

ООП в Python

- Пользовательские типы - классы
- Атрибуты (поля и методы) класса
- self
- Принципы ООП в Python
- Статические и классовые методы
- Магические методы
- Вычисляемые свойства (property)
- Исключения
- Практика

Разминка

1. Написать собственную реализацию встроенной функции `max`. Она принимает любое количество аргументов и возвращает максимальный аргумент:

```
>>> a = [10, 27, 42, 36]
>>> max_value = max(a)
>>> print(max_value)
42
```

2. Написать собственную реализацию встроенной функции `enumerate()` применяется для итерируемых коллекций (строки, списки, словари и др.) и создает объект, который генерирует кортежи, состоящие из двух элементов - индекса элемента и самого элемента:

```
>>> a = [10, 20, 30, 40]
>>> for i in enumerate(a):
...     print(i)
...
(0, 10)
(1, 20)
(2, 30)
(3, 40)
```

Пользовательские типы - классы

Писать через функции - это процедурно-ориентированное программирование. Через классы и объекты - объектно-ориентированное.

В языке программирования Python объекты принято называть также экземплярами. Это связано с тем, что в нем все классы сами являются объектами класса `type`. Поэтому во избежание путаницы объекты, созданные на основе обычных классов, называют экземплярами (но часто и объектами тоже).

```
# определение класса:
```

```
class Class1:  
    pass
```

```
# создание экземпляра класса:
```

```
object1 = Class1()
```

Атрибуты (поля и методы) класса

```
class Employee:
    """Работник"""
    # поля
    first_name = "Unknown"
    last_name = "Unknown"

    def get_full_name(self): # метод
        return self.first_name + ' ' + self.last_name

    @staticmethod
    def get_class_name(): # статический метод класса
        return "Employee"
```

```
Unknown
Unknown Unknown
Employee
```

```
# создаем объект класса Employee и записываем ссылку на него
# в переменную e
e = Employee()
# обращаться к полям и методам объекта и класса нужно через точку
print(e.first_name)
fullname = e.get_full_name()
print(fullname)
print(Employee.get_class_name())
```

self

Методы класса имеют одно отличие от обычных функций: они должны иметь обязательный дополнительный параметр, добавляемый к началу списка параметров. Однако, при вызове метода никакого значения этому параметру присваивать не нужно – его укажет сам Python. Этот параметр указывает на сам экземпляр класса и называется `self`. `self` в Python эквивалентен указателю `this` в C++.

```
class Employee:
```

```
    # __init__ - конструктор класса, аналогично Java и C++
```

```
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self.login = (first_name[0] + last_name).lower()
```

```
    def say(self, word):
        print("{} say {}".format(self, word))
```

self

Выполнение следующего кода завершится ошибкой:

```
e1 = Employee('Al', 'Rid')
print(e1.login)

e2 = Employee() # попробуйте, почему ошибка?

# обратите внимание, что при вызове нужен всего 1 аргумент,
# а параметра при описании 2
e1.say('hi!')
```

```
arid
Traceback (most recent call last):
  File "F:/VirtualBox VMs/ShareWithVirtDeb/PythonCourse/untitled/test2.py", line
17, in <module>
    e2 = Employee2() # попробуйте, почему ошибка?
TypeError: __init__() takes exactly 3 arguments (1 given)
```



self

Исправляем и снова запускаем:

```
e1 = Employee('Al', 'Rid')
print(e1.login)

e2 = Employee('Ivan', 'Ivanov') # исправляем

# обратите внимание, что при вызове нужен всего 1 аргумент,
# а параметра при описании 2
e1.say('hi!')
```

```
arid
<__main__.Employee2 instance at 0x025DD620> say hi!
```


Принципы ООП в Python:

Наследование

Наследование подразумевает то, что дочерний класс содержит все атрибуты родительского класса, при этом некоторые из них могут быть переопределены или добавлены в дочернем.

```
#!/usr/bin/python3
```

```
class Manager(Employee):  
    def __init__(self, first_name, last_name, employees):  
        super().__init__(first_name, last_name)  
        self.employees = employees  
  
    def get_employees_count(self):  
        return len(self.employees)
```

```
m1 = Manager("Lu", "Min", [e1])  
print(m1.get_employees_count())  
print(m1.login)
```

```
1  
lmin
```


Принципы ООП в Python:

Наследование

Для Python2 вызов конструктора базового класса осуществляется напрямую:

```
#!/usr/bin/python
```

```
class Manager(Employee):  
    def __init__(self, first_name, last_name, employees):  
        Employee.__init__(self, first_name, last_name)  
        self.employees = employees
```

Чтоб можно было обращаться через `super()` и пользоваться другими возможностями типа `object`, базовый класс должен наследовать типу `object`.

```
#!/usr/bin/python
```

```
class Employee(object):  
    def __init__(self, first_name, last_name):  
        self.first_name = first_name  
        self.last_name = last_name  
        self.login = (first_name[0] + last_name).lower()  
  
class Manager(Employee):  
    def __init__(self, first_name, last_name, employees):  
        super(Manager, self).__init__(first_name, last_name)  
        self.employees = employees
```

Принципы ООП в Python: Инкапсуляция

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса. Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

```
class Class1:
    # одинарное подчеркивание НЕ меняет имя атрибута
    def _protected(self):
        print('This is protected method')

    # двойное подчеркивание меняет имя атрибута (косвенная защита от доступа)
    # внутри класса метод доступен с указанным именем
    def __private(self):
        print('This is private method')

object1 = Class1()
object1._protected() # метод доступен по указанному имени
object1.__private()  # ошибка, в Class1 такого метода не существует
object1._Class1__private() # настоящее имя метода
```

Принципы ООП в Python: Полиморфизм

Полиморфизм - разное поведение одного и того же метода в разных классах. Например, мы можем сложить два числа, и можем сложить две строки. При этом получим разный результат, так как числа и строки являются разными классами.

```
class Bird:
    def says(self):
        raise NotImplementedError
```

Quack-quack Cuckoo

```
class Duck(Bird):
    def says(self):
        return "Quack-quack"
```

```
class Cuckoo(Bird):
    def says(self):
        return "Cuckoo"
```

```
birds = [Duck(), Cuckoo()]
for bird in birds:
    print(bird.says())
```

Статические и классовые методы

`self` нужен для доступа к атрибутам объекта, но еще есть статические и классовые методы.

```
class Fruit:
```

```
    what = 'fruit'
```

```
    def __init__(self, name):
        self.name = name
```

```
    @classmethod
    def what_is(cls):
        print('I am {}'.format(cls.what))
```

```
    @staticmethod
    def say(color):
        print('I like {} color'.format(color))
```

```
I am fruit
I like green color
I am fruit
```

```
Fruit.what_is() # даже не обязательно создавать объект
Fruit.say('green')
fruit = Fruit('apple')
fruit.what_is() # но можно и создать
```

Магические методы

Python позволяет делать почти все: перегружать операторы сложения и умножения в контексте класса, определять, как присваивать значение конкретному атрибуту, указывать, как объект класса должен вести себя при попытке пройти по нему в цикле, задавать поведение объекта при его удалении. Для всего этого есть магические методы.

```
class Example:
```

```
    def __init__(self, id):
        self.id = id
```

```
    # __del__ определяет поведение при удалении объекта
```

```
    def __del__(self):
        print('Object {} deleted'.format(self.id))
```

```
example = Example(45)
```

```
del example
```

```
Object 45 deleted
```



Магические методы

```
class AllInts:
```

```
    # __contains__ определяет, как вести себя,  
    # если к объекту применяют оператор "in"  
    def __contains__(self, item):  
        if isinstance(item, int):  
            return True
```

```
ints = AllInts()  
if 127 in ints:  
    print('Yes!')
```

```
Yes!
```

Магические методы

```
class Example:
    def __init__(self, id):
        self.id = id

    # __repr__ определяет, как отображать объект в тексте
    def __repr__(self):
        return 'Object with id {}'.format(self.id)

example = Example(42)
print(example)
print(str(example))
print('Obj: {}'.format(example))
```

```
Object with id 42
Object with id 42
Obj: Object with id 42
```


Магические методы

Другие примеры магических методов:

- **`__lt__(self, other)`** - $x < y$ вызывает `x.__lt__(y)`.
- **`__ne__(self, other)`** - $x \neq y$ вызывает `x.__ne__(y)`
- **`__bool__(self)`** - вызывается при проверке истинности. Если этот метод не определён, вызывается метод `__len__` (объекты, имеющие ненулевую длину, считаются истинными).
- **`__getitem__(self, key)`** - доступ по индексу (или ключу).
- **`__sub__(self, other)`** - вычитание ($x - y$).
- **`__float__(self)`** - приведение к float.

Property - вычисляемое свойство

Используется, если необходимо задать поведение отдельного атрибута.

```
#!/usr/bin/python3
```

```
class Mine(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """This is a property of x."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = 'Value: ' + str(value)

    @x.deleter
    def x(self):
        self._x = 'No more'
```

```
m = Mine()
print(m.x)
m.x = 'Test'
print(m.x)
del m.x print(m.x)
```

```
None
Value: Test
No more
```

Property - вычисляемое свойство

Чтобы воспользоваться возможностями property в Python2 надо унаследовать класс от типа object.

```
#!/usr/bin/python
```

```
class Mine(object):  
    def __init__(self):  
        self._x = None  
  
    @property  
    def x(self):  
        """This is a property of x."""  
        return self._x  
  
    @x.setter  
    def x(self, value):  
        self._x = 'Value: ' + str(value)  
  
    @x.deleter  
    def x(self):  
        self._x = 'No more'
```

Исключения

В первую очередь исключения нужны для обработки ошибок и уведомления о произошедшей ошибке вышестоящего кода в тех местах кода, откуда нельзя вернуть значение (конструкторы, деструкторы). Также они позволяют эффективно разделять нормальную логику программы и логику обработки ошибок и исключительных ситуаций.

```
>>> print('hi')
hi
>>> Print('hi')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'Print' is not defined
>>> print('hi')a
File "<input>", line 1
    print('hi')a ^
SyntaxError: invalid syntax
>>> 5 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

Исключения

При выбрасывании исключения стек программы начинает «раскручиваться» с удалением его содержимого, пока не будет найден обработчик исключения (если он есть). Если обработчик не найден – программа аварийно закрывается.

исключения можно ловить и обрабатывать

```
def proc_int(a, b):
    try:
        result = a / b
    except ZeroDivisionError: # указать, какой тип исключений ловить
        print("Don't divide by zero!")
    except Exception as exc: # ловить все остальные типы исключений
        print("Some unexpected error: {}".format(exc))
    else: # если исключения не возникли
        print("No exceptions raised. Result: {}".format(result))
    finally: # выполняется в любом случае
        print("In any case show this message")
```

Исключения

```
>>> proc_int(1, 0)
```

```
Don't divide by zero!
```

```
In any case show this message
```

```
>>> proc_int(1, 'future error')
```

```
Some unexpected error: unsupported operand type(s) for /: 'int' and 'str'
```

```
In any case show this message
```

```
>>> proc_int(5, 5)
```

```
No exceptions raised. Result: 1.0
```

```
In any case show this message
```

Исключения

Исключения можно не только отлавливать и обрабатывать, но и задавать, и выбрасывать, поскольку это такие же классы и объекты.

```
# задаем свой класс исключений
# обязательно должен наследоваться от встроенного класса Exception
class NonStringError(Exception):
    pass # как правило, тело класса исключения остается пустым

def proccess_string(name):
    if not isinstance(name, str):
        raise NonStringError() # выбрасываем свое исключение, если нужно

name = 999
try:
    proccess_string(name)
except NonStringError: # обрабатываем только свой тип исключений
    print('Trying to process non-string object')
```


Практика

1. Написать класс `Man`, который принимает имя в конструкторе. Имеет метод `solve_task`, который просто выводит "I'm not ready yet".
2. Написать класс `Pupil`, у которого переопределен метод `solve_task`. На этот раз он будет думать от 3 до 6 секунд (с помощью метода `sleep` библиотеки `time` и `randint` библиотеки `random`).

Практика*

3. Написать класс `WrapStrToFile`, который будет иметь одно вычисляемое свойство (property) под названием `content`. В конструкторе класс должен инициализировать атрибут `filepath`, путем присваивания результата функции `mktemp` библиотеки `tempfile`. При попытке чтения свойства `content` должен внутри кода свойства открываться файл, используя атрибут `filepath` (с помощью функции `open`, из этого файла читается все содержимое и возвращается из свойства. Если файл не существует, то возникает ошибка, поэтому должна быть обертка вокруг открытия файла на чтение (`try...except`), с помощью которого будет возвращаться 'Файл еще не существует'. При присваивании значения свойству `content` файл по указанному пути должен открываться на запись и записываться содержимое. Не забудьте закрывать файл после чтения или записи. При удалении атрибута `content`, должен удаляться и файл.

Практика*

```
class WrapStrToFile:
    def __init__(self):
        # здесь инициализируется атрибут filepath,
        # он содержит путь до файла-хранилища

    @property
    def content(self):
        # попытка чтения из файла, в случае успеха возвращаем содержимое
        # в случае неудачи возвращаем 'File doesn't exist'

    @content.setter
    def content(self, value):
        # попытка записи в файл указанного содержимого

    @content.deleter
    def content(self):
        # удаляем файл: os.remove(имя_файла)
```

Практика*

```
wstf = WrapStrToFile()  
print(wstf.content)    # Output: File doesn't exist  
wstf.content = 'test str'  
print(wstf.content)    # Output: test_str  
wstf.content = 'text 2'  
print(wstf.content)    # Output: text 2  
del wstf.content       # после этого файла не существует
```