Лекция 8

Специальные атрибуты классов

Перегрузка операторов

Преобразование типов

Свойства объектов и операторы

- Неформально возможности объекта определяют как *свойства*
  - Свойства объектов реализуются их методами
  - Применение оператора к объекту эквивалентно вызову метода реализующего данный оператор, то есть код операции над объектом хранится в самом объекте
- Реализация операций над объектами в виде их свойств обеспечивает параметрический полиморфизм
- Понятие протокола как набора свойств
  - Если объект может быть использован в определенном контексте, это называют поддержкой протокола соответствующего контекста
  - Протокол реализуется набором свойств (методов) объекта
- Примеры протоколов: протокол итерации, протокол контекста with

Реализация функции dir()

- Встроенная функция dir() возвращает список атрибутов объекта, переданного ей в качестве параметра
- Функция dir() применима к модулям, так как импортированный модуль также является объектом
- Каждый объект имеет метод __dir__(), реализующий свойство "иметь список атрибутов"
- Вызов dir(obj) приводит к вызову obj.__dir__()
  Результат сортируется и преобразуется в список
- При вызове без аргументов функция dir() возвращает список имен в локальной области видимости
- Функция dir() главный инструмент интроспекции - изучения внутреннего устройства объекта

Функция dir() и метод __dir__() это пример того, как полиморфная операция языка реализуется в виде свойства объекта

Класс object

- Начиная с версии 3 в Питоне структура наследования классов выстроена в единую иерархию. На вершине иерархии находится класс *object*.
- Все объекты явно или неявно наследуют от класса object
  - Класс object становится родительским для класса в определении которого родительские классы не указаны
  - Класс имеющий хотя бы одного явно указанного родителя получает через него наследование от класса object
- Базовый набор свойств всех объектов наследуется от класса object в виде его атрибутов и методов
- Большая часть специальных атрибутов, реализующих объектную модель языка Питон, также наследуется (с возможной модификацией) от класса object

Специальные атрибуты

Атрибут __class__

- Атрибут __class__ создается автоматически при создании объекта
  - Атрибут __class__ любого объекта является ссылкой на его класс. Для *класса* это ссылка на класс *type*. Для *объекта* это ссылка на класс от которого объект был произведен.
  - По атрибуту атрибута __class__.__name__ можно получить строку с именем класса объекта.
  - Встроенная функция *isinstance(obj, classinfo)* позволяет проверить находится ли класс *classinfo* в иерархии наследования объекта *obj*. Параметром *classinfo* также может быть кортеж, содержащий несколько классов.

```
class Empty:
  pass
e = Empty()
isinstance(e, Empty) # => True
isinstance(e, object) # => True
isinstance(e, (str, Empty)) # => True, объект класса str или Empty
```

Атрибут __dict__

- Атрибут __dict__ создается автоматически при создании объекта
- Атрибут __dict__ это словарь содержащий атрибуты объекта. Объект *агрегирует* словарь.
- В каждой паре ключ - значение:
  - ключ это имя атрибута
  - значение это сам атрибут
- Атрибут __dict__ изменяемый, воздействуя на него можно изменять атрибуты объекта:

```
class Point():
  def __init__(self, x, y):
    self.x = x
    self.y = y

p = Point(3, 4)
p.__dict__['color'] = 'red'
print(p.color) # => 'red'
```

Атрибуты __bases__ и __module__

- Атрибуты *__bases__* и *__module__* создаются автоматически при создании класса, это *атрибуты класса*
- Атрибут __bases__ это кортеж содержащий базовые классы в порядке их указания как параметров при определении класса. Для класса *object* это пустой кортеж. Для класса у которого явно не заданы базовые классы это класс object.
- Атрибуты __bases__ являются дугами в графе наследования, проходя по ним класс находит унаследованные атрибуты
- Атрибут __module__ содержит *имя модуля* в котором определен класс, через него класс видит глобальные переменные своего модуля
- *Объект произведенный от класса не получает доступ к атрибутам __bases__ и __module__*
- *Функция dir() не возвращает атрибуты __bases__ и __module__ в общем списке*

Атрибут __doc__

- Атрибут __doc__ создается автоматически при создании класса
  - Атрибут __doc__ содержит строку документации класса объекта если она задана. Для класса *object* это строка 'The most base type'.
  - Если строка документации не задана, атрибут __doc__ получает значение None

```
class Empty:
  'Пустой класс'

print(Empty.__doc__) # => 'Пустой класс'

e = Empty()
print(e.__doc__) # => 'Пустой класс'
```

Атрибуты класса object

Методы класса object

```
__init__           # Инициализация (конструктор) объекта

__dir__            # Возвращает список атрибутов объекта
__doc__            # Строка документации

__eq__             # == Оператор сравнения на равенство
__ne__             # != Оператор сравнения на неравенство
__ge__             # >= Оператор больше или равно
__gt__             # >  Оператор больше
__le__             # <= Оператор меньше или равно
__lt__             # <  Оператор меньше

__getattribute__   # Вызывается при чтении значения атрибута
__setattr__        # Вызывается при присваивании значения атрибуту
__delattr__        # Вызывается при удалении атрибута инструкцией del

__hash__           # Возвращает хеш объекта как целое число
__reduce__         # Обеспечивает сериализацию объекта для pickle
__reduce_ex__      # Обеспечивает сериализацию с указанием версии протокола

__repr__           # Возвращает строку, представляющую объект для отладки
__str__            # Возвращает строку, представляющую объект для пользователя
__format__         # Форматирует объект для печати
```

Методы класса object, продолжение

```
__new__(cls)     # Статический метод, создает и возвращает объект
__sizeof__(obj)  # Возвращает размер объекта в байтах

__subclasshook__(subcls)  # Проверка является ли subcls моим субклассом
```

- Встроенная функция *issubclass(cls, classinfo)* позволяет проверить является ли класс *cls* подклассом *classinfo*. Параметром *classinfo* также может быть кортеж, содержащий несколько классов. В данном контексте класс считается подклассом самого себя.

```
issubclass(int, object)  # =>True
issubclass(int, int)     # => True
issubclass(int, str)     # => False
issubclass(int, (str, object)) # => True
```

Методы операций над числами

```
__abs__        # Функция abs() - абсолютная величина
__add__        # Оператор +
__sub__        # Оператор -
__mul__        # Оператор *
__truediv__    # Оператор /
__floordiv__   # Оператор //
__mod__        # Оператор %
__divmod__     # Функция divmod() - результат деления и остаток
__pow__        # Оператор ** - возведение в степень, pow()
__lshift__     # Оператор <<
__rshift__     # Оператор >>
__and__        # Оператор &
__or__         # Оператор |
__xor__        # Оператор ^
```

- Пример, оператор сложения:

```
a + b # => a.__add__(b)
```

Правосторонние варианты операций

- Все методы операций над числами имеют свой правосторонний вариант отличающийся добавлением символа *r* в начало имени метода
- Правосторонние методы арифметических операций используются если
  - в операции участвуют операнды разных типов и
  - у левого операнда не определен необходимый метод
- Работа обычного (левостороннего) оператора __sub__

```
a - b # => a.__sub__(b)
```

- Работа правостороннего оператора __rsub__

```
a - b => b.__rsub__(a)
```

Метод __call__()

- Метод __call__() это реализация оператора вызова функции ()
- Встроенная функция *callable(obj)* возвращает True, если объект может быть вызван как функция
- Встроенная функция более общего вида *hasattr(obj, attr)* возвращает True, если объект *obj* имеет атрибут с именем *attr*
  - прежде чем обратиться к атрибуту можно проверить его наличие, что дает возможность явной проверки с дальнейшей обработкой ошибочной ситуации

```
class Simple:
  def __init__(self):
    self.x = 1
  def __call__(self):
    print('I am object of class Simple')
s = Simple()
s() # => 'I am object of class Simple'
callable(s)            # => True  | callable(Simple)            # => True
hasattr(s, '__call__') # => True  | hasattr(Simple, '__call__') # => True
hasattr(s, 'x')        # => True  | hasattr(Simple, 'x')        # => False
hasattr(s, 'y')        # => False | hasattr(Simple, 'y')        # => False
```

Методы __getitem__() и __setitem__()

- Метод __getitem__() это реализация работы оператора [] при *считывании* элемента коллекции
- Метод __setitem__() это реализация работы оператора [] при *модификации* элемента коллекции
- Пример, словарь как список величин:

```
class DictToList:
  def __init__(self, content):
    self.d = content
  def __getitem__(self, index):
    return self.d[sorted(self.d.keys())[index]]
  def __setitem__(self, index, value):
    self.d[sorted(self.d.keys())[index]] = value

c = DictToList({'a': 0, 'b': 1, 'c': 2})
print(c[2]) # => 2
c[2] = 22
print(c[2]) # => 22
print(c[3]) # => IndexError: list index out of range
```

Работа с фрагментами

- Методы __getitem__() и __setitem__() имеют только один параметр индексации - аргумент *index*
- Для работы с фрагментом в качестве параметра index передается объект типа *slice*
- Пример:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(a[2:8:2])        # => [2, 4, 6] # Эти две строки
print(a[slice(2,8,2)]) # => [2, 4, 6] # эквивалентны
```

- При реализации методов __getitem__() и __setitem__() следует проанализировать тип параметра *index* и реализовать две ветви алгоритма:
  - работа с фрагментом если параметр index имеет тип slice,
  - или работа с единственным элементом в противном случае
- Здесь явная проверка типа это пример ad hoc полиморфизма

Приведение объекта к встроенному типу данных

- Методы объекта и встроенные функции:

```
__bool__   # вызывается функцией bool()
__int__    # вызывается функцией int()
__float__  # вызывается функцией float()
__str__    # вызывается функцией str()
__repr__   # вызывается функцией repr()
```

- Для того, что бы объект приводился к определенному типу данных нужно реализовать соответствующий метод

- Логические операции в языке Питон не переопределяются. Поведение объекта в логической операции определяется методом *__bool__*. Если метод не определен, объект как непустая сущность имеет значение True.

Приведение типа должно быть явным

- Методы преобразования типа используются только при явном вызове соответствующей встроенной функции
- Пример:

```
class A:
  def __int__(self):
    return 1
a = A()
print(a + 1) # => TypeError: unsupported operand type(s) for +
print(int(a) + 1) # => 2
```

- Автоматическое преобразования типов предусмотрено для классов чисел:

```
print(1 + 2.2 + 3+5j + Fraction(6, 2)) # => (9.2+5j)
print(1 + Decimal(3)) # => 4
```

- При сочетании типов int и Decimal тип int автоматически преобразуется в тип Decimal

Функция id()

- Встроенная функция id() возвращает целое число - уникальный идентификатор объекта
- Оператор *is* сравнивает идентификаторы, возвращенные функцией id()
- Новый объект может получить идентификатор, ранее принадлежавший другому, уже уничтоженному объекту
  - идентификаторы не рекомендуется кешировать (сохранять в переменных)
- Уникальный идентификатор это пример свойства объекта, которое нельзя изменить. На уникальных идентификаторах основана система управления объектами в интерпретаторе.

Функции getattr() и setattr()

- Встроенные функции getattr() и setattr() это функциональная запись оператора точка

```
Запись  v = a.x  эквивалентна записи  v = getattr(a, 'x')
Запись  a.x = v  эквивалентна записи  setattr(a, 'x', v)
```

- При использовании функциональной записи в качестве имени атрибута можно использовать значение хранящееся в переменной или полученное в результате вычисления выражения

```
a.x = 10
attribute_name = input('Which attribute print ? ')
  # => пользователь вводит x

print(a.attribute_name) # => Error: no attribute 'attribute_name'
print(getattr(a, attribute_name)) # => 10
```

Методы __getattr__() и __setattr__()

- Методы __getattr__() и __setattr__() позволяют имитировать несуществующие атрибуты
- При обращении к атрибуту посредством точечной нотации
  - изменение значения атрибута приводит к вызову метода __setattr__()
  - получение значения атрибута приводит к вызову метода __getattr__(), но *только в том случае если атрибут не существует*
- Метод __getattribute__(), аналогичен __getattr__(), но он вызывается и для существующих атрибутов также
- Переопределение методов __getattr__() и __setattr__() позволяет перегрузить оператор точка

Управление созданием атрибутов

```
class Sample:
  def __getattr__(self, name):
    return 'Unknown'
  def __setattr__(self, name, value):
    #setattr(self, value, value.upper()) # => Бесконечная рекурсия
    self.__dict__[name] = value.upper()

d = Sample()
print(d.__dict__) # => {}

print(d.color) # => Unknown
d.color = 'green'
print(d.color) # => GREEN

print(d.mode) # => Unknown
d.mode = 'simple'
print(d.mode) # => SIMPLE

print(d.__dict__) # => {'color': 'GREEN', 'mode': 'SIMPLE'}
```

Виртуальные атрибуты

- Виртуальный атрибут это имя, при обращении к которому как к атрибуту происходит вызов определенных для этого функций
- Функция property создает виртуальный атрибут

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- Виртуальный атрибут определен тремя функциями:
  - getter - реализует чтение значения атрибута
  - setter - реализует запись (установку) атрибута
  - destroyer - уничтожает атрибут
- для виртуального атрибута может быть задана строка документации как четвертый параметр функции property()
- Создание виртуального атрибута это своеобразное перепрограммирование операции присваивания

Виртуальный атрибут, пример

```
class VoltageRegulator:
  def __init__(self):
    self.v_exists = True

  def get_v(self):
    return get_voltage_from_remote_device() if self.v_exists else error()

  def set_v(self, v):
    set_voltage_on_remote_device(v) if self.v_exists else error()

  def del_v(self):
    self.v_exists = False

  v = property(get_v, set_v, del_v, "Voltage property")

r = VoltageRegulator()
r.v = 215
print('Remote device voltage is', r.v, 'V')
del r.v
print('After del r.v voltage is', r.v)
help(r)
```

Преобразование типов

и

перегрузка операторов

Класс для демонстрации перегрузки операторов

```
class Device:
  def __init__(self, serial, precision):
    self.serial = int(serial)
    self.precision = float(precision)

# Объекты - элементы класса
d = Device(serial=830274, precision=0.02)
e = Device(serial=316839, precision=0.01)
```

Функции в последующих примерах будут методами этого класса

Преобразование к числу

- Преобразование к целому числу

```
def __int__(self):
  return self.serial
```

- Преобразование к вещественному числу

```
def __float__(self):
  return self.precision
```

Преобразование к строке

- Преобразование для лучшего восприятия

```
def __str__(self):
  return 'Device No ' + str(self.serial) + \
  ' with precision ' + str(self.precision) + ' %'
```

- Преобразование для передачи структуры объекта

```
def __repr__(self):
  return 'Object of class Device, serial = ' + \
  str(self.serial) + ', precision = ' + str(self.precision)
```

- Функция print() для преобразования аргументов в строку использует метод __str__(), если он определен, если нет, то метод __repr__()
- Для *явного* преобразования объекта в строку служат встроенные функции str() и repr()

Перегрузка двуместного оператора

```
def __mod__(self, other):
  return int(self) % int(other)

def __rmod__(self, other):
  print('Call to __rmod__() => ', end='')
  return int(other) % int(self)
```

- В операциях можно смешивать объекты типа Device и целые числа

```
print(d % e)        # => 196596
print(d % 7)        # => 4
print(12394727 % d) # => Call to __rmod__() => 770891
```

Придание объекту логического значения

- Пример:

```
class EvenNumber:
  def __init__(self, n):
    self.n = n
  def __bool__(self):
    return bool(self.n % 2 == 0)

i = EvenNumber(10)
bool(i) # => True
'abc' and i # => Объект i класса EvenNumber
not i   # => False

j = EvenNumber(11)
bool(j) # => False
[] or j # => Объект j класса EvenNumber
not i   # => True
```

Общие правила преобразования типов

- При передаче объекта в функцию, следует сохранять полиморфизм насколько это возможно. Если операция требует конкретный тип данных следует явно преобразовать объект к этому типу.
- При вычислении выражения все преобразования типов происходят только явным вызовом встроенных функций

```
print(d / 2)      # => TypeError: unsupported operand type(s)
                  # => for /: 'Device' and 'int'
print(int(d) / 2) # => 415137.0
```

- В логических операциях преобразование типов не происходит, объект *интерпретируется* как логическое значение, однако *используется* в выражении "как есть"
- Объект может быть явно преобразован к логическому типу с результатом True или False
- *Преобразование типов не происходит автоматически. Числа - исключение.*