Лекция 9

Классы,

дополнительные возможности

Декораторы

Наследование и делегирование

Наследование от встроенных классов

- Наследование от встроенных классов позволяет полностью повторить поведение встроенного класса включая все его протоколы
- Для внесения новых возможностей достаточно минимального объема кода

```
class Results(list):
  def __getitem__(self, i):
    'Перепрограммирование оператора []'
    return str(list.__getitem__(self, i)) + ' V'

r = Results([187, 193, 202, 215, 226, 239])
r.append(241)
del r[4]
print(r[2]) # => 202 V
# Итератор не перепрограммирован !
print(r) # => [187, 193, 202, 215, 239, 241]
print([r[i] for i in range(len(r))])
# => ['187 V', '193 V', '202 V', '215 V', '239 V', '241 V']
```

Делегирование

- Делегирование это методика создания объекта-контроллера, содержащего внутри себя объект-исполнитель.
- Обращения к объекту-контроллеру транслируется к объекту-исполнителю с возможной модификацией как обращения, так и результата.
- Делегирующий класс или объект также называют классом-оберткой или объектом-оберткой (wrapper)
- Делегирование это частный случай *агрегирования*: объект-контроллер *содержит* объект-исполнитель

Делегирование, пример

```
class ResultsReader:
  'Массив результатов доступных только по чтению'
  def __init__(self, obj):
    self.obj = obj
  def __getitem__(self, i):
    'Перепрограммирование оператора []'
    return str(self.obj[i]) + ' V'
  def __len__(self):
    'Метод __len__ вызывается функцией len()'
    return len(self.obj)

r = ResultsReader([187, 193, 202, 215, 226, 239])
# r.append(241) # append не существует
# r[2] = 241    # элемент не модифицируется
# del r[4]      # del не поддерживается
print(r[2]) # => 202 V
# Итератор не существует
print(r) # => <... ResultsReader object at 0x7f025dc3d198>
print([r[i] for i in range(len(r))])
# => ['187 V', '193 V', '202 V', '215 V', '226 V', '239 V']
```

Наследование и делегирование

- Наследование передает все свойства, за исключением тех, которые были модифицированы явно
- Делегирование передает лишь те свойства, которые были заданы явно, то есть делегирование маскирует все свойства, кроме необходимых
- Наследование выражает отношение "является" (is)
- Делегирование в своей реализации является отношением "содержит" (contains). Однако, с точки зрения интерфейса это отношение скорее можно назвать "имеет отдельные свойства делегируемого объекта"
- В процедурном стиле программирования аналогичные действия выполняет функция, получающая функцию как параметр:

```
def wrapper(f, *pargs, **nargs):
  # предварительные действия
  result = f(*modified_pargs, **modified_nargs)
  # завершающие действия
  return modified_result
```

Псевдочастные атрибуты

Псевдочастные атрибуты

- Один символ подчеркивания в начале имени атрибута это соглашение для именования *внутренних* атрибутов: `_attrname`
- Псевдочастный атрибут это атрибут имя которого начинается с двух символов подчеркивания
  - и которое *не заканчивается* двумя символами подчеркивания: `__attrname`
- Имя псевдочастного атрибута при чтении кода интерпретатором автоматически модифицируется по схеме:

```
__attrname => _ClassName__attrname
```

- Автоматическая модификация имени атрибута (mangling) происходит только внутри класса, в котором атрибут определен
- Модифицируются как атрибуты объекта, так и атрибуты класса
- Псевдочастные атрибуты позволяют избежать конфликта имен при множественном наследовании и затруднить их использование вне класса, где они определены

Пример конфликта имен

```
class DevA:
  def __init__(self): self.precision = 0.02
  def prec(self):
    print('DevA:', self.precision)
    return self.precision
class DevB:
  def __init__(self): self.precision = 0.05
  def prec(self):
    print('DevB:', self.precision)
    return self.precision
class Dev(DevA, DevB):
  def __init__(self):
    DevA.__init__(self)
    DevB.__init__(self)
  def prec_a(self): return DevA.prec(self)
  def prec_b(self): return DevB.prec(self)
d = Dev()
d.prec()
d.prec_a()
d.prec_b()
```

Разрешение конфликта имен

```
class DevA:
  def __init__(self): self.__precision = 0.02
  def prec(self):
    print('DevA:', self.__precision)
    return self.__precision
class DevB:
  def __init__(self): self.__precision = 0.05
  def prec(self):
    print('DevB:', self.__precision)
    return self.__precision
class Dev(DevA, DevB):
  def __init__(self):
    DevA.__init__(self)
    DevB.__init__(self)
  def prec_a(self): return DevA.prec(self)
  def prec_b(self): return DevB.prec(self)
d = Dev()
d.prec()   # => DevA: 0.02 ( _DevA__precision )
d.prec_a() # => DevA: 0.02 ( _DevA__precision )
d.prec_b() # => DevB: 0.05 ( _DevB__precision )
```

Функция super()

Функция super()

- Встроенная функция super() используется внутри метода класса для вызова одноименного метода одного из родительских классов вверх по иерархии
- Вызов без параметров осуществляется для объекта self. Для поиска метода используется стандартный алгоритм начиная с класса, внутри метода которого она вызвана, но не включая его.

```
def prec(self):
  super().prec()
```

- Вызов с двумя параметрами осуществляется для объекта *obj*. Для поиска метода используется стандартный алгоритм начиная с класса *cls*, но не включая его.

```
def prec(self):
   super(cls, obj).prec()
```

Вызов super() эквивалентен вызову super(объемлющий_класс, self)

Класс super

- Функция super возвращает *объект класса super*
- Объект класса super имеет доступ к ближайшему в иерархии наследования классу и делегирует ему вызываемые методы
- При вызове функции super без параметров или с двумя параметрами результатом будет объект класса super связанный с объектом self в первом случае или с явно указанным объектом во втором

```
super() # => super(объемлющий_класс, self)
super(cls, obj)
```

- При вызове функции super с одним параметром, создается *несвязанный* объект класса super. На практике не используется.

Назначение функции super

- Функция super предназначена для вызова методов родительских классов
- Если метод дочернего класса не изменяет, а дополняет и расширяет действие метода родительского класса, имеет смысл вызвать метод родительского класса из метода класса дочернего
- Если разработчики классов будут следовать этому правилу, то в методе родительского класса будет вызван метод его родительского класса и так далее вверх по иерархии классов. Таким образом одним вызовом метода с использованием функции super будут вызваны *все* одноименные методы классов в иерархии наследования.
- Функция super почти всегда вызывается без параметров, ее результат используется для вызова метода с тем же именем, что и у метода, внутри которого она вызвана

```
def prepare_data(self, *pargs, **nargs):
  super().prepare_data(*pargs, **nargs)
```

Атрибут __mro__

- Атрибут __mro__ (Method Resolution Order) это атрибут-кортеж, содержащий последовательность классов, в которой производится поиск атрибутов
- Элементы помещенные в атрибут __mro__ определяются алгоритмом C3-линеаризации
- Функция super() использует атрибут __mro__ класса объекта, который был ей передан как *второй* параметр. Поиск в кортеже начинается с класса, переданного в функцию super() как *первый* параметр, но не включая его.

```
super(cls, obj)
```

- Вторым параметром функции super() может быть класс, в этом случае он используется "как есть", то есть используется атрибут __mro__ этого класса
- Атрибут __mro__ это атрибут класса. Функция dir *не показывает* этот атрибут.

Вызов метода __init__

- Для вызова метода __init__ родительских классов используется следующая конструкция:

```
class C(A, B):
  def __init__(self, *pargs, **nargs):
     # сначала создаются свойства родительских классов
     super().__init__(*pargs, **nargs)
     # затем инструкции инициализации
     # специфичные для объекта класса C

x = C()
```

- В Питоне версии 3 принято все вызовы метода __init__ родительских классов оформлять таким образом

Пример вызова метода __init__

```
class A:
  def __init__(self, *pargs, **nargs):
    # Вызов super(A, x)
    super().__init__(*pargs, **nargs) # Будет вызван B.__init__()

class B:
  def __init__(self, *pargs, **nargs):
    # Вызов super(B, x)
    super().__init__() # Будет вызван object.__init__()

class C(A, B):
  def __init__(self, *pargs, **nargs):
    # Вызов super(C, x)
    super().__init__(*pargs, **nargs) # Будет вызван A.__init__()

x = C() # Кортеж C.__mro__ == (C, A, B, object)
```

- Функция super() ищет класс, находящийся не *выше* в иерархии наследования, а *дальше* в списке наследования

Связанные методы

Связанные методы

- Связанный метод это объект содержащий ссылку на сам метод и на объект, которому этот метод принадлежит
- Связанный метод обладает свойством "быть вызванным" (callable) и вызывается как обычная функция
- Связанный метод позволяет скрыть объектную природу метода и использовать его в контексте традиционного процедурного программирования
  - Пример: использование связанного метода в качестве callback-функции
  - Механизм callback предполагает связывание произвольной функции с определенным событием в системе

Связанные методы, пример

```
class DevA:
  def __init__(self):
    self.precision = 0.02
  def prec(self):
    print('DevA:', self.precision); return self.precision
class DevB:
  def __init__(self):
    self.precision = 0.05
  def prec(self):
    print('DevB:', self.precision); return self.precision

da = DevA()
db = DevB()
da.prec() # традиционный вызов "от объекта"
mu = DevA.prec # несвязанный метод
mu(da) # для вызова нужен параметр-объект
mu(db) # можно передать объект другого совместимого класса
mba = da.prec # связанный метод, объект класса DevA
mbb = db.prec # связанный метод, объект класса DevB
mba() # вызов без параметра, использован объект da
mbb() # вызов без параметра, использован объект db
```

Связанные методы, реализация

- Связанный метод представляет собой объект класса *method*
- При создании связанного метода присваиванием

```
mba = da.prec
```

  - создается объект mba класса method
  - в атрибут mba.__self__ помещается объект da
  - в атрибут mba.__func__ помещается метод prec
  - вызов связанного метода реализован как метод __call__() в классе method:

```
def __call__(self, *pargs, **nargs):
  self.__func__(self.__self__, *pargs, **nargs)
```

Таким образом любой объект можно связать с любым доступным ему методом и получить функцию, пригодную к использованию в парадигме процедурного программирования

Статические методы

и методы класса

Статические методы и методы класса

- Статические методы
  - Статический метод создается встроенной функцией staticmethod()
  - У статического метода нет аргумента self, его первый аргумент, если он есть, это обычный аргумент
  - Статический метод имеет доступ к атрибутам класса через явно указанное имя класса
- Методы класса (*старый термин в новом значении*)
  - Метод класса создается встроенной функцией classmethod()
  - Первый аргумент метода класса это класс
  - Метод класса имеет доступ к атрибутам класса через свой первый аргумент
- Статический метод, как и метод класса может быть вызван как атрибут объекта, так и атрибут класса

Пример реализации методов

```
class Dev:
  _num_instances = 0
  def __init__(self):
    self.precision = 0.02
    Dev._num_instances += 1

  # Обычный метод (метод объекта), аргумент self это объект
  def instance_method(self):
    print('Total ', Dev._num_instances)

  # Метод класса, один аргумент, аргумент cls это класс
  def class_method(cls):
    print('Total ', cls._num_instances)
  class_method = classmethod(class_method)

  # Статический метод, нет аргумента self
  def static_method():
    print('Total ', Dev._num_instances)
  static_method = staticmethod(static_method)
```

Пример вызова методов

```
# Обычный метод вызывается как атрибут объекта
xa = Dev()
xa.instance_method() # => 1
# или объект нужно передать как аргумент
# при вызове в виде атрибута класса
 Dev.instance_method(xa) # => 1

# Метод класса вызывается как атрибут объекта,
# так и как атрибут класса
xb = Dev()
xb.class_method() # => 2
Dev.class_method() # => 2

# Статический метод вызывается как атрибут объекта,
# так и как атрибут класса
xc = Dev()
xc.static_method() # => 3
Dev.static_method() # => 3
```

Статический метод и метод класса используют только атрибуты класса. Удаление объекта из параметров передаваемых методу позволяет избежать ошибочных обращений к атрибутам объекта.

Декораторы

Декораторы функций

- Функцию, используемую подобно функциям staticmethod() и classmethod() называют функцией-оберткой или метафункцией
- Для применения функций-оберток введен специальный синтаксис, называемый декоратором. Следующие два варианта записи кода идентичны.
  - Присваивание:

```
def class_method(cls):
  print('Total ', cls._num_instances)
class_method = classmethod(class_method)
```

  - Декоратор:

```
@classmethod
def class_method(cls):
  print('Total ', cls._num_instances)
```

- Декоратор это еще один способ модификации свойств объекта, но с использованием процедурного подхода

Расшифровка синтаксиса

```
# Запись

@decorator
def function()
  pass

# Эквивалентна записи

def function()
  pass
function = decorator(function)
```

- Функция decorator. Вызывается функция с аргументом function и ее возвращаемое значение присваивается имени function. Функция decorator возвращает функцию.
- Класс decorator. Создается объект класса decorator и присваивается имени function. Объект может быть вызван как функция (имеет свойство callable)

Декораторы виртуальных атрибутов

```
# Функциональная запись
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")

# Запись с декораторами
@property
def v(self):
  "Voltage property"
  return get_voltage_from_remote_device() if self.v_exists else error()
@v.setter
def v(self, v):
  set_voltage_on_remote_device(v) if self.v_exists else error()
@v.deleter
def v(self):
  self.v_exists = False
```

Пример декоратора в виде функции

```
def corrector(fun):
  def corrected_fun(channel=0):
    if channel < 0 or channel >= 4:
      channel = 0
    t = fun(channel)
    if t < 0: t = 0
    if t > 100: t = 100
    return t
  return corrected_fun

@corrector
def four_channels_thermometer(channel):
  if channel == 0: return 20
  elif channel == 1: return 35
  elif channel == 2: return 107
  elif channel == 3: return -3
  else:
    print('ERROR: Invalid channel', channel, end=', ')
    return None
```

Пример декоратора в виде класса

```
class corrector:
  def __init__(self, fun):
    self.fun = fun
  def __call__(self, channel=0):
    if channel < 0 or channel >= 4:
      channel = 0
    t = self.fun(channel)
    if t < 0: t = 0
    if t > 100: t = 100
    return t

@corrector
def four_channels_thermometer(channel):
  if channel == 0: return 20
  elif channel == 1: return 35
  elif channel == 2: return 107
  elif channel == 3: return -3
  else:
    print('ERROR: Invalid channel', channel, end=', ')
    return None
```

Декораторы методов

- Метод определенный в классе это обычная функция, первый аргумент которой имеет специальное значение
- С методами могут быть использованы декораторы, реализованные как функции
- Использование декораторов в виде класса с методами невозможно, так как при вызове __call__() в качестве аргумента self будет передан объект-декоратор вместо экземпляра класса, метод которого декорируется (пример 12)

```
class decor:
   def __init__(self, f): ...
   def __call__(self): ...
class X:
  def fun(self):   # fun это метод класса X
    pass           # self это объект класса X
  fun = decor(fun) # теперь fun это объект класса decor
m = X()
m.fun() # => decor.__call__(X.fun) # параметр это объект класса decor
```

Декораторы с параметрами

- Функция-декоратор или метод __init__() класса-декоратора могут иметь дополнительные параметры
- Дополнительные параметры не передаются декорируемой функции, они оказывают влияние на процесс декорирования
- Декоратор с параметрами "двухслойный", первый вызов - получение параметров, второй вызов - декорирование

```
class Button:
  def __init__(self):
    self.color = 'White'
    self.__callback = None

  def set_callback(self, callback):
    self.__callback = callback

  def click(self):
    print(self.color, 'button clicked: ', end='')
    if self.__callback:
      self.__callback()
```

продолжение примера

```
button_open = Button()
button_test = Button()

def callback_decorator(button, color):
  def real_decorator(fun):
    button.color = color
    button.set_callback(fun)
    return fun
  return real_decorator

@callback_decorator(button_open, 'Green')
def open_message():
  print('Open file')

# Запись @callback_decorator(button_open, 'Green') перед функцией
# эквивалентна записи
# open_message = callback_decorator(button_open, 'Green')(open_message)
# после функции

button_open.click() # => Green button clicked: Open file
button_test.click() # => White button clicked:
```

Декораторы классов

- Подобно функциям классы также могут быть подвергнуты декорированию:

```
def wrapper(cls):
  class WrappingClass(cls):
    def prec(self):
      print('Customized message for ', end='')
      cls.prec(self)
  return WrappingClass

@wrapper                # => Dev = wrapper(Dev)
class Dev:
  def __init__(self):
    self.precision = 0.02
  def prec(self):
    print('Dev:', self.precision)

d = Dev()
d.prec() # => Customized message for Dev: 0.02
```

- Декораторами классов могут быть как функции так и классы

Вложенные декораторы

Запись

```
@dec1
@dec2
@dec3
def function()
  pass
```

эквивалентна записи

```
def function()
  pass
function = dec1(dec2(dec3(function)))
```

- Вложенные декораторы вызываются в порядке обратном их записи; первым будет вызван декоратор ближайший к определению функции

Применимость декораторов

- Декоратор полезен когда:
  - необходимо сделать "привязку" функции или класса к другим объектам программы
  - большое количество функций или классов должно быть модифицировано одинаковым образом,
  - вносимые изменения носят временный характер, например для отладки,
  - в рамках процедурного программирования возникает необходимость использования техники наследования
- Примеры:
  - в класс вносятся статические атрибуты, например для подсчета созданных объектов
  - в функцию вносится отладочная печать, протоколирующая переданные параметры и возвращаемое значение