# Лекция 8. Классы ч.2

Декораторы-функции

Декораторы-классы

Декораторы в классах: @classmethod, @staticmethod, @property

# Декораторы

Декораторы и их назначение.

Декораторы-функции и декораторы-классы

Декораторы в классах

# Декораторы

**Декораторы -** это функции-обертки, которые позволяют сделать **дополнение** к уже **существующей** функции.

Декоратор **принимает функцию**, добавляет новые возможности и **возвращает улучшенный вариант функции**.

*Внимание: поведение самой функции вы изменить не сможете. Но, например, сможете провести дальнейшую обработку результата, возвращаемого из функции, или добавить вывод информации для пользователя*

# Правила создания декоратора

- Создается функция-декоратор decorator, которая принимает параметр-внешнюю функцию fun
- Внутри нее создается обертка для внешней функции wrapper
- Функция-обертка возвращает внешнюю функцию fun
- Декоратор возвращает функцию-обертку wrapper
- Перед внешней функцией simple пишется название декоратора с @: @decorator

```
def decorator(fun):
  print("decorator")
  def wrapper():
    print("Работает функция wrapper")
    return eval(fun())
  return wrapper
```

```
@decorator
def simple():
  print("Работает функция simple с декоратором")
  return "1+1"

print(simple())
```

# Правила создания декоратора (продолжение)

- Создается несколько функций-декораторов decorator, decorator2 и т.д.
- Перед внешней функцией simple пишутся названия декораторов в обратном порядке
- Декораторы вызываются в порядке «кто ближе к функции»: decorator -> decorator2 ->... или decorator2(decorator(simple))

```
@decorator2
@decorator
def simple():
  print("Работает функция simple с несколькими декораторами")
  return "1+1"

print(simple())
```

# Декораторы для логирования

Можно подключить библиотеку логирования logging и выводить отладочную информацию о работе фукнции с результат работы в файл, а не на экран.

Обратите внимание: функция, декорированная логгером, ничего не пишет на экране! Она пишет результат в лог-файл с именем, соответствующим имени функции. А возвращает <function simple2 at 0x7f6abd399790>

```
@log
def simple2(*args):
  print("Работает функция simple с аргументами")
  return sum(args)

print(simple2(1,2,3,4,5))
```

```
import logging

def log(func):
  def wrap_log(*args, **kwargs):
    name = func.__name__
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)

    fh = logging.FileHandler("%s.log" % name)
    fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    formatter = logging.Formatter(fmt)
    fh.setFormatter(formatter)
    logger.addHandler(fh)

    logger.info("Вызов функции: %s" % name)
    # info("Вызов функции: %s" % name)
    result = func(*args, **kwargs)
    # logger.info("Результат: %s" % result)
    logger.info("Результат: %s" % result)
    return func

  return wrap_log
```

это интересно: https://pythonru.com/osnovy/rukovodstvo-po-dekoratoram-python

# Декораторы для оценки времени выполнения

Можно подключить библиотеки time и datetime и рассчитывать время выполнения.

Если сделать эту функцию декоратором, она будет рассчитывать время выполнения любой функции, которую ею декорировали

```python
from datetime import datetime
import time

def estimate_time(fun):
    def wrapper(*args):
        start = datetime.now()
        res = fun(*args)
        end = datetime.now()
        elapsed = (end - start).total_seconds() * 1000
        print(f'{fun.__name__} время выполнения (ms): {elapsed}')
        return res

    return wrapper
```

```python
@estimate_time
def simple(*args):
    print("Работает функция simple с оценкой времени выполнения")
    return sum(args)

print(simple(1,2,3,4,5))
```

# Декораторы для создания «безопасных» функций

```python
def error_handler(fun):
    def wrapper(*args, **kwargs):
        res = 0
        try:
            res = fun(*args, **kwargs)
        except Exception as e:
            print(f"Ошибка {e} при вызове {fun.__name__}")
        return res
    return wrapper
```

```python
# потенциально "опасная" функция
def not_safe(*args):
    if len(args) == 0:
        return 0
    div_res = args[0]
    for arg in args:
        div_res/= arg
    return div_res
```

```python
@error_handler
def simple(*args):
    print('Работает "безопасная" функция simple')
    return not_safe(*args)

print(simple(1,2,3,4,5))
print(simple(1,2,0,4,5))
```

```
Работает "безопасная" функция simple
0.008333333333333333
Работает "безопасная" функция simple
Ошибка float division by zero при вызове simple
0
```

# Декораторы-классы

```python
# декоратор-класс. в этом случае в классе создается функция __call__
from datetime import datetime
class DecoratorArgs:
  def __init__(self, dec_name):
    print(f"Входные аргументы __init__: {dec_name}")

  # метод-декоратор из класса. обратите внимание: аргументы идут напрямую в декоратор,
  #а не в метод __init__ класса
  def __call__(self, fun):
    def wrapper(*args):
      print(f"Входные аргументы wrapper: {args}")
      start = datetime.now()
      res = fun(*args)
      end = datetime.now()
      elapsed = (end - start).total_seconds() * 1000
      print(f'{fun.__name__} время выполнения (ms): {elapsed}')
      return res
    return wrapper

@DecoratorArgs("класс-декоратор")
def simple(*args):
  print(f"Работает функция simple с оценкой времени выполнения")
  return sum(args)

print(simple(1,2,3,4,5))
```

**Зачем?**
Для реализации **наследования**: из «базового» декоратора можно сделать декоратор с дополнительными функциями.

Например, декоратор-класс может просто записывать лог ошибки в файл. А его наследник будет еще и отсылать оповещение сисадмину, см. https://pavel-karateev.gitbook.io/intermediate-python/dekoratory/decorators

# Декораторы в классах

# Декораторы @classmethod, @staticmethod

Пока нет наследования, декораторы @classmethod и @staticmethod не различаются: они делают так, что декорированную ими функцию можно вызывать как из экземпляра класса, так и статически из самого класса

```
class Aircraft:
  def __init__(self, model, weight):
    self.weight = weight
    self.model = model

  @staticmethod
  def get_spare(weight, model):
    return Aircraft(weight, model)

  @classmethod
  def get_spare(self, weight, model):
    return self(weight, model)
```

# Декораторы @classmethod, @staticmethod, @property (продолжение)

При наследовании @classmethod будет относиться к классу-наследнику, а @staticmethod - к базовому (родительскому) классу

```
class Aircraft:
  def __init__(self, model, weight):
    self.weight = weight
    self.model = model

  @staticmethod
  def get_spare(weight, model):
    return Aircraft(weight, model)
```

```
class Multirotor(Aircraft):
  def __init__(self, model, weight, rotors=4):
    self.weight = weight
    self.model = model
    self.rotors = rotors
```

```
print("===Работает экземпляр класса Multirotor===")
aircraft2 = Multirotor("DJI Mavic 2 Pro", 907, 4)
spare = aircraft2.get_spare("DJI Mavic 2 Pro", 907)
print(type(spare))
```

вернет:

===Работает экземпляр класса Multirotor===

<class '__main__.Aircraft'>

# Декоратор @property

@property заменает собой прямое название getter и setter, которое используется в других языках программирования

```
def get_regnum(self):
    return self.__regnum

def set_regnum(self, val):
    self.__regnum = val

regnum = property(get_regnum,
set_regnum)
```

```
@property
def regnum(self):
    return self.__regnum

@regnum.setter
def regnum(self, num):
    self.__regnum = num
```

# Декоратор @property (продолжение)

```
class Aircraft:
  def __init__(self, model, weight):
    self.__weight = weight
    self.__model = model
    self.__regnum = ""

  @property
  def regnum(self):
    return self.__regnum

  @regnum.setter
  def regnum(self, num):
    self.__regnum = num
```

```
print("===Работает экземпляр класса Aircraft===")
aircraft1 = Aircraft("DJI Mavic 2 Pro", 907)
print(aircraft1.regnum)
aircraft1.regnum = "a0rdk9s"
print(aircraft1.regnum)
```