Урок 6

Декораторы и продолжение работы с сетью

Декоратор. Декоратор с параметром. Сетевое программирование (продолжение).

На этом занятии продолжим изучать работу в сети: более подробно рассмотрим модуль **socket**. Перед этим разберем создание пользовательских декораторов, исключим подход «это все магия» относительно конструкций **@property**.

Декораторы

В рамках курса «Python 1» мы познакомились с декораторами **@property**, **@classmethod**, **@staticmethod**. Вспомним основные тезисы.

Декоратор — это функция, основное назначение которой — служить оберткой для другой функции или класса с целью изменить или расширить возможности этого объекта. Синтаксически декораторы оформляются добавлением специального символа @ к имени, как показано ниже:

```
@trace
def square(x):
    return x*x
```

Предыдущий фрагмент является сокращенной версией этого:

```
def square(x):
    return x*x

square = trace(square)
```

В этом примере объявляется функция **square()**. Но сразу же ее объект передается функции **trace()**, а возвращаемый ею — замещает оригинальный **square**.

Рассмотрим реализацию функции **trace**, чтобы выяснить, что полезного она делает:

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log", "w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Вызов %s: %s, %s\n" % (func.__name__, args,
kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s вернула %s\n" % (func.__name__, r))
            return r
       return callf
    else:
        return func
```

В этом фрагменте **trace()** создает функцию-обертку, которая записывает отладочную информацию в файл и затем вызывает оригинальный объект функции. Если теперь вызвать функцию **square()**, в файле **debug.log** можно будет увидеть результат вызова метода **write()** в функции-обертке.

Декоратор может быть реализован в виде класса с магическим методом **__call__**. Он вызывается, когда происходит обращение к экземпляру класса как к функции. (Файл **examples/01_decorators/01_deco.py**):

```
class Log():
    def __init__(self):
        pass

    # Магический метод __call__ позволяет обращаться к объекту класса, как к
функции
    def __call__(self, func):
        def decorated(*args, **kwargs):
            res = func(*args, **kwargs)
            print('log: {}({}, {}) = {}'.format(func.__name__, args, kwargs,
res))
            return res

        return decorated
```

Применение такого декоратора будет выглядеть следующим образом:

```
@Log()
def square(x):
    return x * x
```

При использовании декораторы должны помещаться в отдельной строке, непосредственно перед объявлением функции или класса. Кроме того, допускается указывать более одного декоратора. Например:

```
@foo
@bar
@spam
def grok(x):
    pass
```

В этом случае декораторы применяются в порядке следования. Результат получается тот же самый:

```
def grok(x):
    pass

grok = foo(bar(spam(grok)))
```

Декораторы могут также применяться к определениям классов. Например:

```
@foo
```

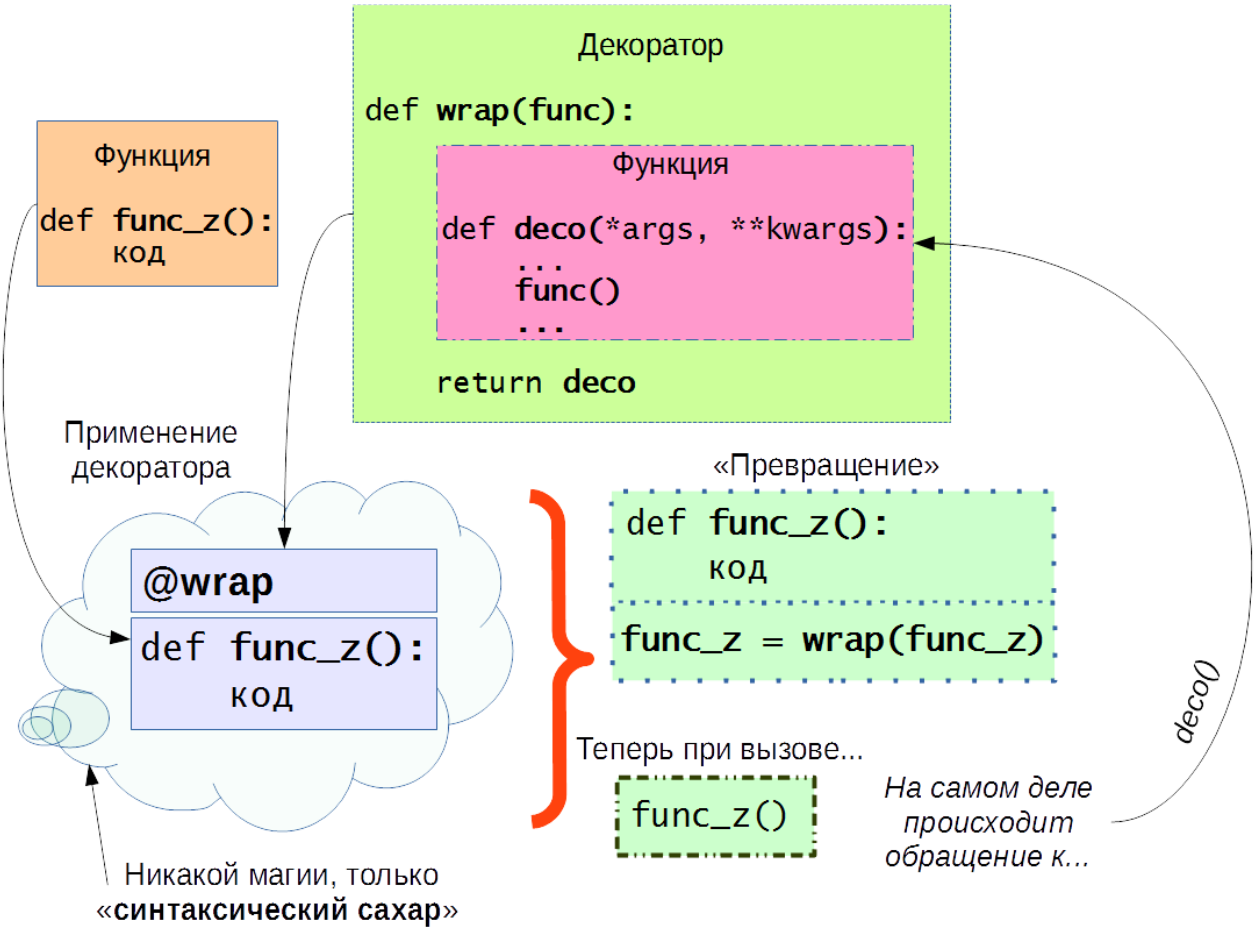
```
class Bar(object):
    def __init__(self,x):
        self.x = x

    def spam(self):
        # Инструкции
```

```
    pass
```

Функция-декоратор, применяемая к классу, всегда должна возвращать его объект. Программному коду, действующему с оригинальным классом, может потребоваться напрямую обращаться к его методам и атрибутам — например, **Bar.spam**. Это будет невозможно, если функция-декоратор **foo()** будет возвращать функцию.

Обобщенно создание и использование декоратора представлено на диаграмме:



Атрибуты функции

На практике часто можно увидеть на месте первой инструкции функции строку документирования, описывающую порядок использования этой функции. Например (файл **examples/01_decorators/02_deco_attrs.py**):

```
def factorial(n):
    """ Вычисляет факториал числа n. Например:
        >>> factorial(6)
       120
    """
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)
```

Строка документирования сохраняется в атрибуте **__doc__** функции. Он часто используется интегрированными средами разработки, чтобы предоставлять интерактивную справку.

Однако обертывание функций с помощью декораторов может препятствовать работе справочной системы, связанной со строками документирования. Рассмотрим в качестве примера следующий фрагмент:

```
def wrap(func):
    def call(*args, **kwargs):
        return func(*args, **kwargs)
    return call

@wrap
def factorial(n):
    """ Вычисляет факториал числа n """
    ...
```

Если теперь запросить справочную информацию для этой версии функции **factorial()**, интерпретатор вернет довольно странное пояснение:

```
>>> help(factorial)
Help on function call in module __main__:
call(*args, **kwargs)
(END)
>>>
```

Интерпретатор вернул строки документации для функции **call**. Чтобы это исправить, декоратор функций должен скопировать имя и строку документирования оригинальной функции в соответствующие атрибуты декорированной версии. Например:

```
def wrap(func):
    def call(*args,**kwargs):
        return func(*args,**kwargs)

    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    return call
```

Для решения этой типичной проблемы в модуле **functools** есть функция **wraps**, которая автоматически копирует эти атрибуты. Она тоже является декоратором:

```
from functools import wraps

def wrap(func):
      @wraps(func)
      def call(*args, **kwargs):
      return func(*args, **kwargs)
    return call
```

Декоратор **@wraps(func)**, объявленный в модуле **functools**, копирует атрибуты функции **func** в атрибуты обернутой версии функции.

Следует иметь в виду, что функции допускают возможность присоединения к ним любых атрибутов. Например:

```
def foo():
    # Инструкции
    ...

foo.secure = 1
foo.private = 1
```

Атрибуты функции сохраняются в словаре, доступном в виде **__dict__**.

В основном атрибуты функций используются в узкоспециализированных приложениях, таких как генераторы парсеров и прикладные фреймворки. Они часто пользуются возможностью присоединить дополнительную информацию к объектам функций.

Как и в случае со строками документирования, атрибуты функций требуют особого внимания при использовании декораторов. Если функция обернута декоратором, попытки обращения к ее атрибутам фактически будут переадресовываться к декорированной версии. Это может быть как желательно, так и нет — в зависимости от потребностей приложения. Чтобы скопировать уже определенные атрибуты функции в атрибуты декорированной версии, можно воспользоваться декоратором **functools.wraps()**, как было показано в предыдущем примере или использовать приведенный далее подход:

```
def wrap(func):
    def call(*args,**kwargs):
        return func(*args,**kwargs)

    call.__doc__ = func.__doc__
    call.__name__ = func.__name__
    call.__dict__.update(func.__dict__)
    return call
```

Когда выполняется декоратор

Главное свойство декораторов — они выполняются сразу после определения декорируемой функции, обычно на этапе импорта (когда Python загружает модуль). Рассмотрим пример (файл **examples/01_decorators/when_deco_is_called.py**):

```
registry = []       # Список-реестр «зарегистрированных» функций

def register(func):
    ''' Декоратор, регистрирующий функции в неком "реестре" '''
    print(' running register ( %s ) ' % func)
    registry.append(func)
    return func
# Функции, которые могут быть декорированы:
@register
def f1() :
    print('running f1()')

@register
def f2() :
    print('running f2()')

# Намеренно декорируем не все функции
def f3():
    print('running f3()')

def main():
    print(' running main()' )
    # Выведем список «зарегистрированных» функций
    print (' registry ->' , registry)

    # Теперь просто выполним все функции
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main()
```

При запуске данного скрипта как программы будет следующий вывод:

```
 running register ( <function f1 at 0x00000000028EABF8> )
 running register ( <function f2 at 0x00000000028EAC80> )
```

```
 running main()
 registry -> [<function f1 at 0x00000000028EABF8>, <function f2 at
0x00000000028EAC80>]
running f1()
running f2()
running f3()
```

Отметим, что **register** выполняется дважды до любой другой функции в модуле. При вызове **register** получает в качестве аргумента декорируемый объект-функцию — например, **<function f1 at 0xl00631bf8>**.

После загрузки модуля в списке **registry** оказываются ссылки на две декорированные функции: **f1** и **f2**. Они, как и **f3**, выполняются только при явном вызове из функции **main**.

Если же данный код импортируется другим модулем (а не запускается как скрипт), вывод выглядит так:

```
>>> import when_deco_is_called
running register ( <function f1 at 0x00000000028BAEA0> )
running register ( <function f2 at 0x00000000028CC048> )
```

Если обратиться к списку **registry**, получим:

```
>>> when_deco_is_called.registry
[<function f1 at 0x00000000028BAEA0>, <function f2 at 0x00000000028CC048>]
```

Этим примером подчеркиваем, что декораторы функций выполняются сразу после импорта модуля, но сами декорируемые функции — только в результате явного вызова. В этом различие между этапом импорта и выполнения в Python (файл **examples/01_decorators/03_when_deco_is_called_import.py**).

```
import when_deco_is_called
```

Если сравнивать этот пример с типичным применением декораторов в реальных программах, он необычен в двух отношениях:

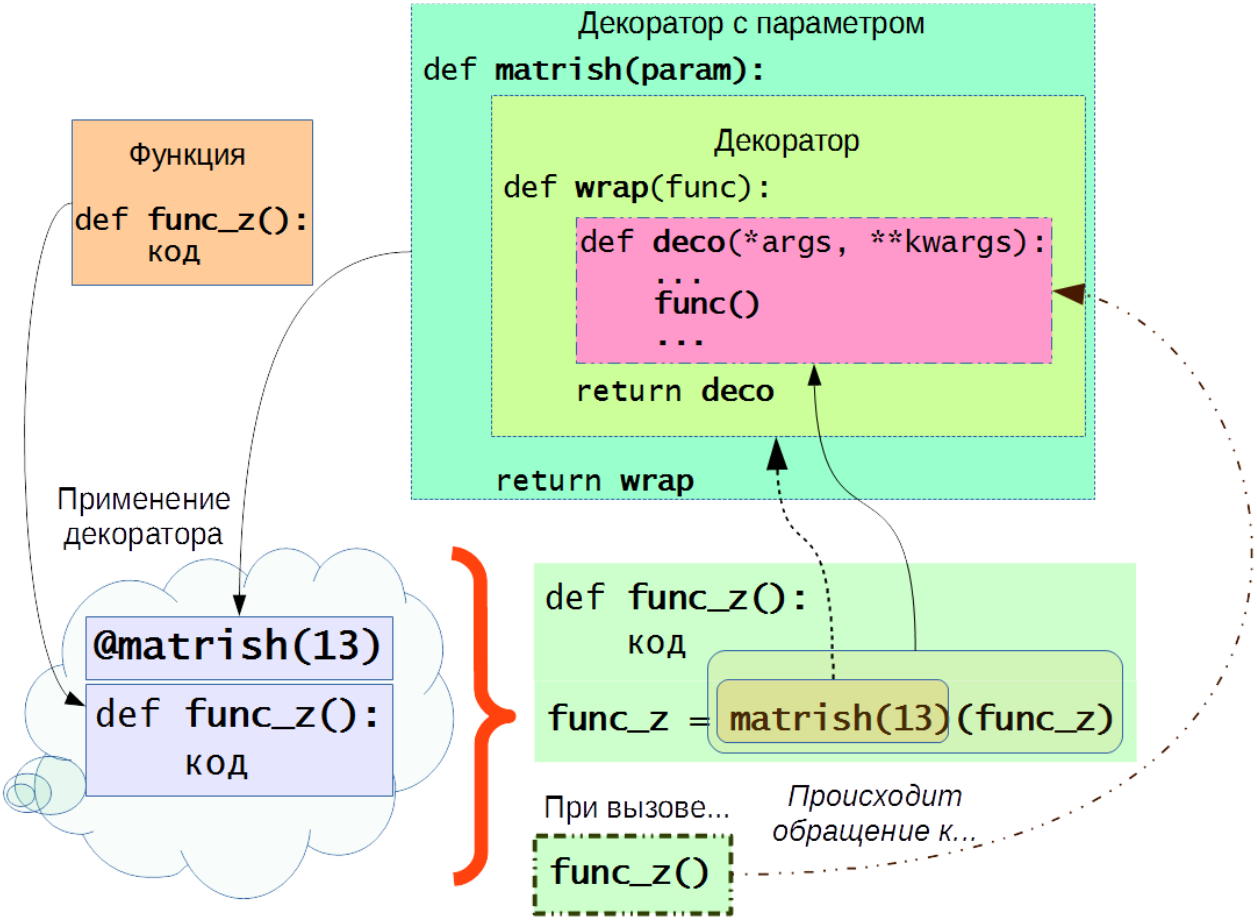
- Функция-декоратор определена в том же модуле, что и декорируемые функции. Настоящий декоратор обычно определяется в одном модуле, а применяется к функциям из других;
- Декоратор **register** возвращает ту же функцию, что была передана в качестве аргумента. На практике декоратор обычно определяет внутреннюю функцию и возвращает именно ее.

Декоратор с параметрами

Как видим на примере функции **functools.wraps**, декораторы могут принимать аргументы. При надлежащем использовании их можно применять как параметры настройки, которые принимают индивидуальные значения для каждой декорируемой функции.

Чтобы реализовать декоратор с параметрами, необходимо обычную функцию-декоратор обернуть функцией, которая будет принимать необходимые параметры.

Эту хитрую взаимосвязь можно проиллюстрировать следующей схемой.


Декоратор с параметром
def matrish(param):
Декоратор
def wrap(func):
def deco(*args, **kwargs):
...
func()
...
return deco
return wrap
Функция
def func_z():
код
Применение декоратора
@matrish(13)
def func_z():
код
def func_z():
код
func_z = matrish(13)(func_z)
При вызове...
func_z()
Происходит обращение к...

В качестве примера рассмотрим декоратор **sleep**, искусственно замедляющий выполнение декорируемой функции (файл **examples/01_decorators/04_deco_params.py**):

```
import time
from functools import wraps

def sleep(timeout):
    def decorator(func):
        @wraps(func)
       def decorated(*args, **kwargs):
            ''' Декоратор sleep
            '''
            t1 = time.sleep(timeout)
            res = func(*args, **kwargs)
            print('Function {} was sleeping'.format(func.__name__))
            return res
         return decorated
    return decorator

@sleep(2)
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

В результате вызов декорированной функции откладывается на указанный таймаут.

Стоит обратить внимание на вывод в консоли:

```
Function factorial was sleeping
Function factorial was sleeping
Function factorial was sleeping
Function factorial was sleeping
Function factorial was sleeping
```

Дело в том, что декоратор применен к функции, которая использует рекурсию.

Декоратор с параметрами может быть реализован не только в виде функции, но и отдельного класса. При этом декорируемая функция аналогично оборачивается декоратором (экземпляром класса) и в него передается значение параметра.

Пример декоратора с параметрами с реализацией через класс (файл **examples/01_decorators/04_deco_params.py**):

```
from functools import wraps
import time

# ---- Декоратор с параметрами, реализованный через класс ----

class Sleep():
    ''' Фабрика декораторов-замедлителей
    '''
    def __init__(self, timeout):
        self.timeout = timeout

    def __call__(self, func):
        ''' Декоратор-замедлитель
        '''
        @wraps(func)
        def decorated(*args, **kwargs):
            ''' Декорированная функция
            '''
            time.sleep(self.timeout)
            res = func(*args, **kwargs)
            print('Function {} was sleeping in class'.format(func.__name__))
            return res
        return decorated


                # Применение декоратора, основанного на классе,
@Sleep(3)       # заключается в создании объекта данного класса
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n-1)

print(' -- Использован декоратор, реализованный через класс --')
print('!!! Обратите внимание на то, сколько раз будет вызван декоратор
(рекурсия) !!!')
print(factorial(5))
```

Результат:

```
 -- Использован декоратор, реализованный через класс --
!!! Обратите внимание на то, сколько раз будет вызван декоратор (рекурсия) !!!
Function factorial was sleeping in class
Function factorial was sleeping in class
Function factorial was sleeping in class
Function factorial was sleeping in class
Function factorial was sleeping in class
120
```

Декоратор и рекурсия

Нужно внимательно отслеживать совместное использование рекурсивных функций и декораторов. Если применить декоратор к рекурсивной функции, все внутренние рекурсивные вызовы будут обращаться к декорированной версии. Например:

```
@locked
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)    # <-- Вызов декорированной версии
factorial
```

Если декоратор имеет отношение к управлению системными ресурсами, такими как механизмы синхронизации или блокировки, от рекурсии лучше отказаться.

Примеры декораторов

Приведем примеры декораторов в стандартной библиотеке Python и популярных фреймворках.

- **@functools.lru_cache(maxsize=128, typed=False)** сохраняет указанное количество результатов вызовов декорируемой функции (мемоизация);
- **@functools.singledispatch** позволяет реализовать функционал одиночной диспетчеризации (**single dispatch**) — когда работа функции зависит от типа единственного аргумента, можно зарегистрировать несколько коротких функций, каждая из которых обрабатывает свой тип аргумента, а не создавать большое ветвление **if..elif..else**;
- **@contextlib.contextmanager** формирует менеджер контекста из декорируемой функции;
- **@abc.abstractmethod** служит для указания абстрактного метода (рассмотрим на занятии по ООП);
- **@atexit.register(func, *args, **kargs)** регистрирует функцию, которая должна быть вызвана при завершении приложения;
- **@login_required** в библиотеке Django (модуль **django.contrib.auth.decorators**) позволяет указать, какие представления (**view**) должны быть доступны только авторизованным пользователям;
- **@app.route(“/”)** в библиотеке **Flask** регистрирует функцию представления для обработки указанного **url**;
- декораторы библиотеке **PyTest**: **@pytest.fixture**, **@pytest.yield_fixture**, **@pytest.mark.paramertize**.

Резюме

- Декоратор — это функция, дополняющая другую функцию или класс;
- Декоратор можно реализовать через функцию или класс, используя магический метод **__call__**;
- Декоратор заменяет атрибуты исходной функции;
- Декоратор выполняется уже на этапе импорта модуля;

- Декоратор может принимать аргументы, тогда он станет «фабрикой декораторов». Его реализация чуть сложнее, чем обычного декоратора;
- Будьте внимательны при использовании декораторов с рекурсивными функциями.

Работа с сетью (продолжение)

Модуль socket

Модуль **socket** предоставляет доступ к стандартному интерфейсу сокетов **BSD**. Он разрабатывался для системы UNIX, но может использоваться во всех платформах. Интерфейс сокетов предназначался для поддержки разнообразных сетевых протоколов (**IP**, **TIPC**, **Bluetooth** и других). Но наиболее часто используется протокол интернета (Internet Protocol, IP), который включает **TCP** и **UDP**. В Python поддерживаются обе версии протокола, **IPv4** и **IPv6**, хотя первая распространена намного шире.

Этот модуль находится на низком уровне и обеспечивает непосредственный доступ к сетевым функциям, предоставляемым операционной системой. При разработке сетевых приложений может оказаться проще использовать специализированные модули: **ftplib, http, smtplib, urllib, xmlrpc** или **SocketServer**, который рассмотрим на следующем занятии.

Семейства адресов

Некоторые функции из модуля **socket** требуют указывать семейство адресов. Оно определяет, какой сетевой протокол будет использоваться. Список констант, которые определены для этой цели:

Константа	Описание
AF_BLUETOOTH	Протокол Bluetooth
AF_INET	Протокол IPv4 (TCP, UDP)
AF_INET6	Протокол IPv6 (TCP, UDP)
AF_NETLINK	Протокол Netlink взаимодействия процессов
AF_PACKET	Пакеты канального уровня
AF_TIPC	Прозрачный протокол взаимодействия процессов (Transparent Inter-Process Communication protocol, TIPC)
AF_UNIX	Протоколы домена UNIX

Из них наиболее часто употребляются семейства **AF_INET** и **AF_INET6**, потому что они представляют стандартные сетевые соединения. Семейство **AF_BLUETOOTH** доступно только в системах, поддерживающих его (обычно они встраиваемые). Семейства **AF_NETLINK**, **AF_PACKET** и **AF_TIPC** поддерживаются только в операционной системе Linux. **AF_NETLINK** используется для скоростных взаимодействий между пользовательскими процессами и ядром Linux. **AF_PACKET** применяют для прямого взаимодействия с уровнем передачи данных (например, для непосредственной работы с пакетами **ethernet**). Семейство **AF_TIPC** определяет протокол, обеспечивающий скоростные

взаимодействия процессов в кластерах, действующих под управлением операционной системы Linux (http://tipc.sourceforge.net/).

Типы сокетов

Некоторые функции в модуле **socket** дополнительно требуют указывать тип сокета. Он определяет тип взаимодействий (потоки или пакеты) для использования с указанным семейством протоколов. Список констант, предназначенных для этого:

Константа	Описание
SOCK_STREAM	Поток байтов с поддержкой логического соединения, обеспечивающего надежность передачи данных (TCP)
SOCK_DGRAM	Дейтаграммы (UDP)
SOCK_RAW	Простой сокет
SOCK_RDM	Дейтаграммы с надежной доставкой
SOCK_SEQPACKET	Обеспечивает последовательную передачу записей с поддержкой логических соединений

На практике чаще используются типы сокетов **SOCK_STREAM** и **SOCK_DGRAM**, потому что они соответствуют протоколам **TCP** и **UDP** в семействе протоколов интернета (IP). Тип **SOCK_RDM** — это разновидность протокола **UDP**, обеспечивает доставку дейтаграмм, но не гарантирует ее порядок (дейтаграммы могут поступать получателю не в том порядке, в каком отправлялись). Тип **SOCK_SEQPACKET** применяют для отправки пакетов через соединения, ориентированные на передачу потока данных с сохранением порядка следования пакетов и их границ. **SOCK_RDM** и **SOCK_SEQPACKET** не получили широкой поддержки — чтобы обеспечить переносимость программ, их лучше не использовать. **SOCK_RAW** используется для низкоуровневого доступа к протоколу и может применяться для реализации специализированных операций, таких как отправка управляющих сообщений (например, **ICMP**). Обычно тип **SOCK_RAW** встречается только в программах, которые предполагается выполнять с привилегиями суперпользователя или администратора.

Не все типы сокетов поддерживаются всеми семействами протоколов. Например, используя семейство **AF_PACKET** для перехвата пакетов **Ethernet** в системе Linux, нельзя установить соединение для приема потока байтов, указав тип **SOCK_STREAM**. Вместо этого придется использовать **SOCK_DGRAM** или **SOCK_RAW**. Семейство **AF_NETLINK** поддерживает только тип **SOCK_RAW**.

Адресация

Чтобы взаимодействия через сокеты были возможны, необходимо указать адрес назначения. Форма адреса зависит от используемого семейства протоколов.

AF_INET (IPv4)

Для интернет-приложений, использующих протокол **IPv4**, адреса определяются в виде кортежа (**host**, **port**). Примеры определения адресов:

- **('www.python.org', 80)**;

- **(‘66.113.130.182’, 25)**.

Если в поле **host** передается пустая строка, это равносильно использованию константы **INADDR_ANY**, которая соответствует любому адресу. Обычно такой способ адресации применяется на стороне сервера, когда создается сокет, принимающий соединения от любых клиентов. Если в поле **host** передается значение **‘<broadcast>’**, это равносильно использованию константы **INADDR_BROADCAST** в API сокетов.

Следует помнить, что при использовании имен хостов вроде **‘www.python.org’** для их преобразования в IP-адреса будет задействована служба доменных имен (**DNS**). В зависимости от настроек службы **DNS** программа может получать каждый раз различные IP-адреса. Чтобы избежать этого, можно использовать обычные адреса, такие как ‘**66.113.130.182**’.

AF_INET6 (IPv6)

Для протокола **IPv6** адреса указываются в виде кортежа из 4 элементов (**host**, **port**, **flowinfo**, **scopeid**).

- В адресах **IPv6** поля **host** и **port** имеют тот же смысл, что и в **IPv4**. Но в числовой форме адрес **IPv6** обычно задается строкой, состоящей из восьми шестнадцатеричных чисел, разделенных двоеточием: ‘**FEDC:BA98:7654:3210:FEDC:BA98:7654:3210**’ или ‘**080A::4:1**’. В последнем случае два двоеточия, следующие подряд, соответствуют недостающим группам чисел, состоящих из нулей;

- В поле **flowinfo** указывается 32-битное число, содержащее 24-битный идентификатор потока (младшие 24 бита) и 4-битное значение приоритета (следующие 4 бита). Старшие 4 бита зарезервированы. Идентификатор потока обычно используется, только когда отправителю необходимо разрешить специальные виды обработки трафика маршрутизаторами. В противном случае в поле **flowinfo** передается значение 0;

- В поле **scopeid** указывается 32-битное число, которое требуется только при работе с адресами локальной сети и локального сайта. Адреса локальной сети всегда начинаются с префикса **‘FE80:...’**. Они используются для взаимодействий между компьютерами, находящимися в одной локальной сети (маршрутизаторы не передают локальные пакеты за пределы сети). В этом случае поле **scopeid** определяет индекс интерфейса, идентифицирующий конкретный сетевой интерфейс хоста. Эту информацию можно увидеть, воспользовавшись командой ‘**ifconfig**’ в UNIX или ‘**ipv6 if**’ в Windows. Локальные адреса сайта всегда начинаются с префикса **‘FEC0:...’** и используются для взаимодействий между компьютерами, составляющими единый сайт (например, все компьютеры в подсети). В этом случае поле **scopeid** определяет числовой идентификатор сайта.

Если нет необходимости указывать значения полей **flowinfo** и **scopeid,** адреса **IPv6** можно указывать в виде кортежа (**host, port**), как и **IPv4**.

Функции модуля *socket*

- **create_connection(address [, timeout])** — устанавливает соединение типа **SOCK_STREAM** с адресом **address** и возвращает уже подключенный объект сокета. В аргументе **address** передается кортеж вида (**host**, **port**). Необязательный аргумент **timeout** определяет предельное время ожидания. Эта функция сначала вызывает функцию **getaddrinfo()** и затем пытается соединиться с каждым из возвращаемых ею адресов;

- **fromfd(fd, family, socktype [, proto])** — на основе целочисленного дескриптора файла **fd** создает объект сокета. В остальных аргументах предаются семейство адресов, тип сокета и номер протокола. Они могут принимать те же значения, что и в функции **socket()**. Дескриптор файла должен ссылаться на ранее созданный объект сокета. Возвращает экземпляр класса **SocketType**;

- **getaddrinfo(host, port [, family [, socktype [, proto [, flags]]]])** — используя информацию **host** и **port** о хосте, эта функция возвращает список кортежей с информацией, необходимой для открытия соединения. В аргументе **host** передается строка с именем хоста или IP-адрес в числовой форме. В аргументе **port** передается число или строка, представляющая имя службы (например, **http**, **ftp**, **smtp**). Каждый из возвращаемых кортежей содержит пять элементов (**family**, **socktype**, **proto**, **canonname**, **sockaddr**). Поля **family**, **socktype** и **proto** могут принимать те же значения, что передаются функции **socket()**. В поле **canonname** возвращается строка, представляющая каноническое имя хоста. В поле **sockaddr** возвращается кортеж с адресом сокета. Например:

```
 >>> getaddrinfo('www.python.org', 80)
[(2,2,17,'',('194.109.137.226',80)), (2,1,6,'',('194.109.137.226'),80))]
```

  В этом примере функция **getaddrinfo()** вернула информацию о двух возможных соединениях. Первый кортеж в списке соответствует соединению по протоколу **UDP (proto=17)**, а второй — соединению по протоколу **TCP (proto=6)**. Чтобы сузить набор доступных адресов, можно использовать дополнительный аргумент **flags** функции **getaddrinfo()**. В следующем примере возвращается информация об установленном соединении по протоколу **IPv4 TCP**:

```
>>> getaddrinfo('www.python.org',80,AF_INET,SOCK_STREAM)
[(2,1,6,'',('194.109.137.226',80))]
```

  Функция **getaddrinfo()** может применяться ко всем поддерживаемым сетевым протоколам (**IPv4**, **IPv6** и так далее). Используйте эту функцию, если вас беспокоит проблема совместимости и поддержки протоколов, которые могут появиться в будущем. Особенно если предполагается, что программа будет поддерживать протокол **IPv6**. Функция **getdefaulttimeout()** возвращает предельное время ожидания в секундах для сокетов. Значение **None** указывает, что предельное время ожидания не было определено. **gethostbyname(hostname)** преобразует имя хоста (такое как '**www.python.org**'), в адрес **IPv4**. IP-адрес возвращается в виде строки вида '**132.151.1.90**'. Эта функция не поддерживает адреса **IPv6**.

- **socket(family, type [, proto])** — по заданным значениям семейства адресов, типа сокета и номера протокола создает новый сокет. Аргумент **family** определяет семейство адресов, а **type** — тип сокета. Аргумент с номером протокола **proto**, как правило, не передается (и по умолчанию принимается значение 0). Обычно он используется только при создании простых сокетов (**SOCK_RAW**) и может принимать значение, зависящее от используемого семейства адресов.

  Для открытия соединения по протоколу **TCP** можно использовать вызов **socket(AF_INET, SOCK_STREAM)**. По протоколу **UDP** — вызов **socket(AF_INET, SOCK_DGRAM)**.

  Возвращает экземпляр класса **SocketType** (описывается ниже).

- **socketpair([family [, type [, proto]]])** — создает пару объектов сокетов, использующих указанное семейство адресов **family**, тип **type** и протокол **proto**. Аргументы имеют тот же смысл, что и в функции **socket()**. Начиная с Python версии 3.5, поддерживается в Windows. Аргумент **family** имеет значение по умолчанию **AF_UNIX**, если это поддерживается платформой, или **AF_INET** в ином случае. Аргумент **type** может принимать только значение **SOCK_DGRAM** или **SOCK_STREAM**. Если в аргументе **type** передается значение **SOCK_STREAM**, создается канал потока. В аргументе **proto** обычно передается значение 0 (по умолчанию). В основном эта функция используется для организации канала взаимодействия с процессами, которые создаются функцией **os.fork()**. Например, родительский процесс может с помощью функции **socketpair()** создать пару сокетов и вызвать

функцию **os.fork()**. После этого родительский и дочерний процессы могут взаимодействовать друг с другом, используя эти сокеты. Также может применяться для организации юнит-тестирования.

Экземпляры класса *socket.socket*

Сокет **s** класса **socket** обладает следующими методами:

- **s.accept()** — принимает соединение и возвращает кортеж (**conn**, **address**). В поле **conn** возвращается новый объект сокета, который может использоваться для приема и передачи данных через соединение. В поле **address** возвращается адрес сокета с другой стороны соединения;

- **s.bind(address)** — присваивает сокету указанный адрес **address**. Формат его представления зависит от семейства. В большинстве случаев это кортеж вида (**hostname**, **port**). Для IP-адресов пустая строка представляет **INADDR_ANY** (любой адрес), а строка '**<broadcast>**' — широковещательный адрес **INADDR_BROADCAST**. Значение **INADDR_ANY** (пустая строка) в поле **hostname** используется, чтобы показать, что сервер может принимать соединения на любом сетевом интерфейсе. Это значение часто применяется, когда сервер имеет несколько сетевых интерфейсов. Значение **INADDR_BROADCAST** (**'<broadcast>'**) в поле **hostname** применяется, когда сокет предполагается использовать для рассылки широковещательных сообщений;

- **s.close()** — закрывает сокет. Этот метод также вызывается, когда объект сокета утилизируется сборщиком мусора. Начиная с Python версии 3.2, в объект **socket** была добавлена поддержка протокола менеджера контекста. Выход из менеджера контекста аналогичен вызову **close()**;

- **s.connect(address)** — устанавливает соединение с удаленным узлом, имеющим адрес **address**. Его формат зависит от семейства, к которому он относится, но обычно в этом аргументе передается кортеж (**hostname**, **port**). В случае ошибки вызывает исключение **socket.error**. При подключении к серверу, запущенному на том же компьютере, в поле **hostname** можно передавать имя хоста '**localhost**';

- **s.fileno()** — возвращает файловый дескриптор сокета;

- **s.getpeername()** — возвращает адрес удаленного узла, с которым установлено соединение. Обычно возвращает кортеж (**ipaddr**, **port**), но вообще формат этого значения зависит от семейства адресов. Этот метод поддерживается не во всех системах;

- **s.getsockname()** — возвращает собственный адрес сокета. Обычно возвращаемым значением является кортеж (**ipaddr**, **port**).

- **s.getsockopt(level, optname [, buflen])** — возвращает значение параметра сокета. Аргумент **level** — уровень параметра. Чтобы получить параметры уровня сокета, в нем передается значение **SOL_SOCKET**, а параметры уровня протокола — номер протокола, такой как **IPPROTO_IP**. Аргумент **optname** определяет имя параметра. Если аргумент **buflen** отсутствует, предполагается, что параметр имеет целочисленное значение, которое и возвращается методом. Если **buflen** указан, он определяет максимальный размер буфера, куда должно быть записано значение параметра. Этот буфер возвращается методом в виде строки байтов. Интерпретация его содержимого с помощью модуля **struct** или других инструментов целиком возлагается на вызывающую программу.

- **s.gettimeout()** — возвращает текущее значение предельного времени ожидания в секундах, если оно установлено. Если нет — возвращает число с плавающей точкой или **None**;
- **s.listen(backlog)** — переводит сокет в режим ожидания входящих соединений. Аргумент **backlog** определяет максимальное количество запросов на соединение, ожидающих

обработки. Они могут быть приняты, прежде чем новые запросы начнут отвергаться. Этот аргумент должен иметь значение не меньше 1, а значения 5 вполне достаточно для большинства применений;

- **s.recv(bufsize [, flags])** — принимает данные из сокета. Данные возвращаются в виде строки. Максимальный объем принимаемых данных определяется аргументом **bufsize**. В аргументе **flags** может передаваться дополнительная информация о сообщении, но обычно он не используется (в этом случае он по умолчанию принимает значение 0);

- **s.send(string [, flags])** — посылает данные через сетевое соединение. Необязательный аргумент **flags** имеет тот же смысл, что и в методе **recv()**. Возвращает количество отправленных байтов. Это число может оказаться меньше количества байтов в строке **string**. В случае ошибки вызывает исключение;

- **s.sendall(string [, flags])** — посылает данные через сетевое соединение. Прежде чем вернуть управление, пытается отправить все данные. В случае успеха возвращает **None**; при ошибке — выдает исключение. Аргумент **flags** имеет тот же смысл, что и в методе **send()**;

- **s.setblocking(flag)** — если в аргументе **flag** передается ноль, сокет переводится в неблокирующий режим работы. В ином случае устанавливается блокирующий режим (по умолчанию). При работе в неблокирующем режиме, если вызов метода **recv()** не находит данных или вызов метода **send()** не может немедленно отправить их, появляется исключение **socket.error**. При работе в блокирующем режиме вызовы этих методов в подобных обстоятельствах блокируются, пока не появится возможность продолжить работу;

- **s.setsockopt(level, optname, value)** — устанавливает значение **value** для параметра **optname** сокета. Аргументы **level** и **optname** имеют тот же смысл, что и в методе **getsockopt()**. В аргументе **value** может передаваться целое число или строка, представляющая содержимое буфера. В последнем случае приложение должно гарантировать, что строка содержит допустимое значение;

- **s.settimeout(timeout)** — устанавливает предельное время ожидания для операций, выполняемых сокетом. В аргументе **timeout** передается число с плавающей точкой, определяющее интервал времени в секундах. Значение **None** означает отсутствие предельного времени ожидания. Если оно указано, по его истечении операции будут вызывать исключение **socket.timeout**. Предельное время ожидания должно устанавливаться сразу же после создания сокета — оно также применяется к операциям, связанным с созданием логического соединения (например, **connect()**).

- **s.shutdown(how)** — отключает одну или обе стороны соединения. Если в аргументе **how** передается значение 0, дальнейший прием данных будет запрещен. Если 1 — будет запрещена дальнейшая передача данных. Если 2 — будут запрещены и прием, и передача данных.

Рассмотрим параметры сокета, поддерживаемые в языке Python. Большая их часть относится к расширенному API сокетов и содержит низкоуровневую информацию о сети. Более подробное описание параметров можно найти в документации и в специализированной литературе. Имена типов, встречающиеся в столбце «Значение», соответствуют стандартным типам языка C. Они ассоциированы со значением и используются в стандартном программном интерфейсе сокетов. Не все параметры поддерживаются в каждой из систем.

Ниже перечислены наиболее часто используемые имена параметров уровня **SOL_SOCKET**:

Имя параметра	Значение	Описание

SO_ACCEPTCONN	0, 1	Разрешает или запрещает прием соединения
SO_BROADCAST	0, 1	Разрешает или запрещает передачу широковещательных сообщений
SO_DEBUG	0, 1	Разрешает или запрещает запись отладочной информации
SO_DONTROUTE	0, 1	Разрешает или запрещает передавать сообщения в обход таблицы маршрутизации
SO_ERROR	int	Возвращает признак ошибки
SO_EXCLUSIVEADDRUSE	0, 1	Разрешает или запрещает присваивать тот же адрес и порт другому сокету. Отключает действие параметра SO_REUSEADDR
SO_KEEPALIVE	0, 1	Разрешает или запрещает периодическую передачу служебных сообщений другой стороне, чтобы поддерживать соединение в активном состоянии
SO_LINGER	linger	Откладывает вызов метода **close()**, если в буфере передачи имеются данные. Значение типа **linger** — это упакованная двоичная строка, содержащая два 32-битных целых числа (**onoff, seconds**)
SO_RCVBUF	int	Размер приемного буфера (в байтах)
SO_RCVTIMEO	timeval	Максимальное время ожидания в секундах при приеме данных. Значение **timeval** — это упакованная двоичная строка, содержащая два 32-битных целых числа без знака (**seconds, microseconds**)
SO_REUSEADDR	0, 1	Разрешает или запрещает повторно использовать локальные адреса
SO_REUSEPORT	0, 1	Разрешает или запрещает нескольким процессам присваивать сокетам один и тот же адрес — при условии, что этот параметр установлен в значение 1 во всех процессах
SO_SNDBUF	int	Размер передающего буфера (в байтах)

SO_SNDTIMEO	timeval	Максимальное время ожидания в секундах при передаче данных.

На данном занятии мы рассмотрели декораторы и продолжили знакомство с сетевым программированием. Чтобы понимать механизм работы декораторов, важно знать различия между этапом импорта и выполнения, а также разбираться в областях действия переменных и замыканиях.

Домашнее задание

1. Продолжая задачу логирования, реализовать декоратор **@log**, фиксирующий обращение к декорируемой функции. Он сохраняет ее имя и аргументы.
2. В декораторе **@log** реализовать фиксацию функции, из которой была вызвана декорированная. Если имеется такой код:

```
@log
def func_z():
   pass

def main():
   func_z()
```

...в логе должна быть отражена информация:

```
"<дата-время> Функция func_z() вызвана из функции main"
```

Дополнительные материалы

1. Logging Cookbook.
2. Python Decorators Library.
3. Awesome Python Decorator – A curated list of awesome python decorator resources.
4. Понимаем декораторы в Python шаг за шагом. Шаг 1.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. Лучано Ромальо. Python. К вершинам мастерства (каталог «Дополнительные материалы»).
3. Дэвид Бизли. Python. Подробный справочник (каталог «Дополнительные материалы»).