Продвинутый курс Python

Урок 2

Декораторы. Менеджеры контекста. Сети (продолжение). Логгирование событий

Замыкания (closures). Декоратор. Декоратор с параметром. Менеджеры контекста и оператор with. Сетевое программирование, модули select и socket. Журналирование событий и модуль logging.

Введение

На этом занятии будет продолжено изучение работы в сети - будут рассмотрены модули select и socket, будет рассмотрена возможность обслуживания нескольких подключений. Перед этим будут рассмотрены вопросы создания пользовательских декораторов и менеджеров контекста, что позволит исключить подход “это всё магия” относительно конструкций @property и оператора with. Дополнительно будет рассмотрен вопрос журналирования (логгирования) событий с использованием модуля logging.

Декораторы

В рамках курса “Python-1” было знакомство с такими декораторами как @property, @classmethod, @staticmethod. На прошлом занятии при работе с PyTest использовались декоторы этой библиотеки. Возможно отдельные студенты изучали вопрос создания собственных декораторов. Для того, чтобы уравнять знания слушателей в этом вопросе, необходимо раскрыть тему декораторов, чтобы дальнейшее их использование не вызывало непонимания. Для этого сначала будет раскрыта тема замыканий.

Замыкания (closures)

Функции в языке Python – объекты первого класса. Это означает, что они могут передаваться другим функциям в виде аргументов, сохраняться в структурах данных и возвращаться функциями в виде результата. Ниже приводится пример функции, которая на входе принимает другую функцию и вызывает ее:

```
# foo.py
def callf(func):
    return func()
```

А это пример использования функции, объявленной выше:

```
>>> import foo
>>> def helloworld():
...     return 'Привет, Мир!'
...
>>> foo.callf(helloworld)   # Передача функции в виде аргумента
'Привет, Мир!'
>>>
```

Когда функция интерпретируется как данные, она неявно несет информацию об окружении, в котором была объявлена функция, что оказывает влияние на связывание свободных переменных в функции. В качестве примера рассмотрим модифицированную версию файла foo.py, в который были добавлены переменные:

```
# foo.py
x = 42
def callf(func):
    return func()
```

Теперь исследуем следующий пример:

```
>>> import foo
>>> x = 37
>>> def helloworld():
...     return "Привет, Мир! x = %d" % x
...
>>> foo.callf(helloworld)       # Передача функции в виде
аргумента
'Привет, Мир! x = 37'
>>>
```

Обратите внимание, как функция helloworld() в этом примере использует значение переменной x, которая была определена в том же окружении, что и сама функция helloworld(). Однако хотя переменная x определена в файле foo.py и именно там, фактически, вызывается функция helloworld(), при исполнении функцией helloworld() используется не это значение переменной x.

Когда инструкции, составляющие функцию, упаковываются вместе с окружением, в котором они выполняются, получившийся объект называют замыканием (closure). Такое поведение предыдущего примера объясняется наличием у каждой функции атрибута __globals__, ссылающегося на глобальное пространство имен, в котором функция была определена. Это пространство имен всегда соответствует модулю, в котором была объявлена функция. Для предыдущего примера атрибут __globals__ содержит следующее:

```
>>> helloworld.__globals__
{'__builtins__': <module '__builtin__' (built-in)>,
'helloworld': <function helloworld at 0x7bb30>,
'x': 37, '__name__': '__main__', '__doc__': None
'foo': <module 'foo' from 'foo.py'>}
>>>
```

Когда функция используется как вложенная, в замыкание включается все ее окружение, необходимое для работы внутренней функции. Например:

```
import foo

def bar():
    x = 13

def helloworld():
    return "Привет, Мир! x = %d" % x

foo.callf(helloworld)           # Вернет 'Привет, Мир! x = 13'
```

Замыкания и вложенные функции особенно удобны, когда требуется написать программный код, реализующий концепцию отложенных вычислений. Рассмотрим еще один пример:

```
from urllib.request import urlopen
def page(url):
    def get():
        return urlopen(url).read()
    return get
```

Функция page() в этом примере не выполняет никаких вычислений. Она просто создает и возвращает функцию get(), которая при вызове будет извлекать содержимое веб-страницы. То есть вычисления, которые производятся в функции get(), в действительности откладываются до момента, когда фактически будет вызвана функция get(). Например:

```
>>> python = page("http://www.python.org")
>>> jython = page("http://www.jython.org")
>>> python
<function get at 0x95d5f0>
>>> jython
<function get at 0x9735f0>
>>> pydata = python()           # Извлечет страницу http://www.python.org
>>> jydata = jython()           # Извлечет страницу http://www.jython.org
>>>
```

Две переменные, python и jython, объявленные в этом примере, в действительности являются двумя различными версиями функции get(). Хотя функция page(), которая создала эти значения, больше не выполняется, тем не менее обе версии функции get() неявно несут в себе значения внешних переменных на момент создания функции get(). То есть при выполнении функция get() вызовет urlopen(url) со значением url, которое было передано функции page(). Взглянув на атрибуты объектов python и jython, можно увидеть, какие значения переменных были включены в замыкания. Например:

```
>>> python.__closure__
(<cell at 0x67f50: str object at 0x69230>,)
>>> python.__closure__[0].cell_contents
'http://www.python.org'
>>> jython.__closure__[0].cell_contents
'http://www.jython.org'
>>>
```

Замыкание может быть весьма эффективным способом сохранения информации о состоянии между вызовами функции. Например, рассмотрим следующий пример, в котором реализован простой счетчик:

```
def countdown(n):
    def next():
        nonlocal n
        r = n
        n -= 1
        return r
    return next

# Пример использования
next = countdown(10)
while True:
    v = next() # Получить следующее значение
    if not v: break
```

В этом примере для хранения значения внутреннего счетчика n используется замыкание. Вложенная функция next() при каждом вызове уменьшает значение счетчика и возвращает его предыдущее значение. Программисты, незнакомые с замыканиями, скорее всего реализовали бы такой счетчик с помощью класса, например:

```
class Countdown:
    def __init__(self,n):
        self.n = n

    def next(self):
        r = self.n
        self.n -= 1
        return r

# Пример использования
c = Countdown(10)
while True:
    v = c.next() # Получить следующее значение
    if not v: break
```

Однако, если увеличить начальное значение обратного счетчика и произвести простейшие измерения производительности, можно обнаружить, что версия, основанная на замыканиях, выполняется значительно быстрее (прирост скорости может достигать 50%).

Тот факт, что замыкания сохраняют в себе окружение вложенных функций, делает их удобным инструментом, когда требуется обернуть существующую функцию с целью расширить ее возможности - это является объектом темы “Декораторы”.

Декораторы

Декоратор – это функция, основное назначение которой состоит в том, чтобы служить оберткой для другой функции или класса. Главная цель такого обертывания – изменить или расширить возможности обертываемого объекта. Синтаксически декораторы оформляются добавлением специального символа @ к имени, как показано ниже:

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

Предыдущий фрагмент является сокращенной версией следующего фрагмента:

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

square = trace(square)
```

В этом примере объявляется функция square(). Однако сразу же вслед за объявлением объект функции передается функции trace(), а возвращаемый ею объект замещает оригинальный объект square.

Теперь 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 функция trace() создает функцию-обертку, которая записывает некоторую отладочную информацию в файл и затем вызывает оригинальный объект функции. То есть, если теперь вызвать функцию square(), в файле debug.log можно будет увидеть результат вызова метода write() в функции-обертке. Функция callf, которая возвращается функцией trace(), – это замыкание, которым замещается оригинальная функция. Но самое интересное в этом фрагменте заключается в том, что возможность трассировки включается только с помощью глобальной переменной enable_tracing. Если этой переменной присвоить значение False, декоратор trace() просто вернет оригинальную функцию, без каких-либо изменений. То есть, когда трассировка отключена, использование декоратора не влечет за собой снижения производительности.

Декоратор может быть реализован в виде класса с магическим методом __call__ (этот метод вызывается, когда происходит обращение к экземпляру класса как к функции):

```
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, что и ниже:

```
def grok(x):
    pass

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

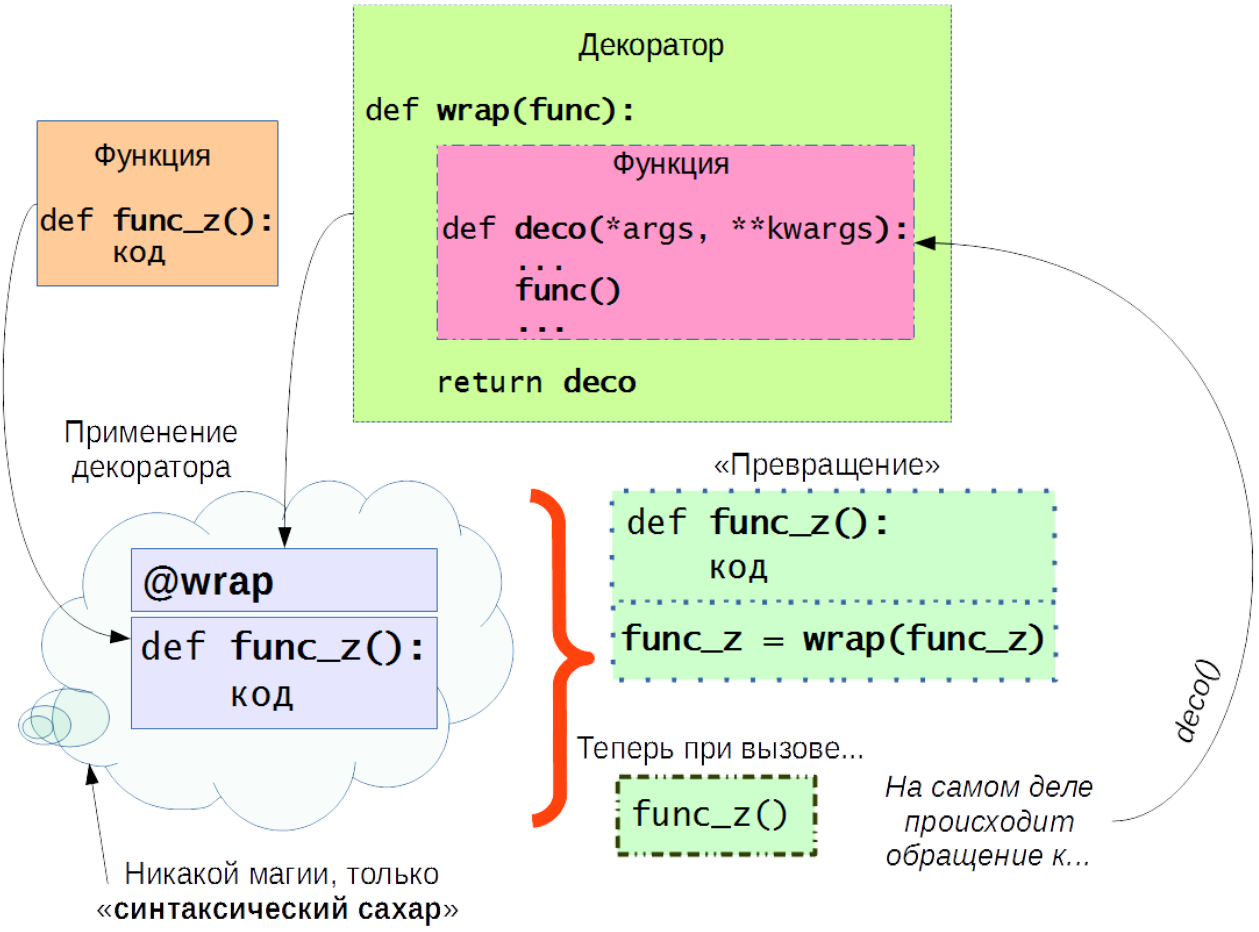
Декораторы могут также применяться к определениям классов. Например:

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

    def spam(self):
        # инструкции
        pass
```

Функция-декоратор, применяемая к классу, всегда должна возвращать объект класса. Программному коду, действующему с оригинальным классом, может потребоваться напрямую обращаться к методам и атрибутам класса, например: Bar.spam. Это будет невозможно, если функция-декоратор foo() будет возвращать функцию. Подробнее декораторы классов будут рассмотрены на занятии, посвященном ООП (метапрограммирование).

Обобщенно создание и использование декоратора можно представить на диаграмме:



Атрибуты функции

На практике часто можно увидеть на месте первой инструкции функции строку документирования, описывающую порядок использования этой функции. Например:

```
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, который часто используется интегрированными средами разработки для предоставления интерактивной справки.

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)
>>>
```

То есть интерпретатор вернул строки документации для функции call. Чтобы исправить эту проблему, 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 достаточно типичной проблемы в модуле functools имеется функция wraps, которая автоматически копирует эти атрибуты. Неудивительно, что 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 атрибута __dict__ функции.

В основном атрибуты функций используются в узкоспециализированных приложениях, таких как генераторы парсеров и прикладные фреймворки, которые часто пользуются возможностью присоединения дополнительной информации к объектам функций.

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

```
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 - то, что они выполняются сразу после определения декорируемой функции. Обычно на этапе импорта (т.е. когда Python загружает модуль). Рассмотрим пример (файл 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 аргумент декорируемый объект функцию, например, <function fl at 0xl00631bf8>.

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

Если же данный код импортируется другим модулем (а не запускается как скрипт), то 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, то получим:

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

Основная цель данного примера - подчеркнуть, что декораторы функций выполняются сразу после импорта модуля, но сами декорируемые функции - только в результате явного вызова. В этом проявляется различие между этапом импорта и этапом выполнения в Python.

По сравнению с типичным применением декораторов в реальных программах пример выше необычен в двух отношениях:

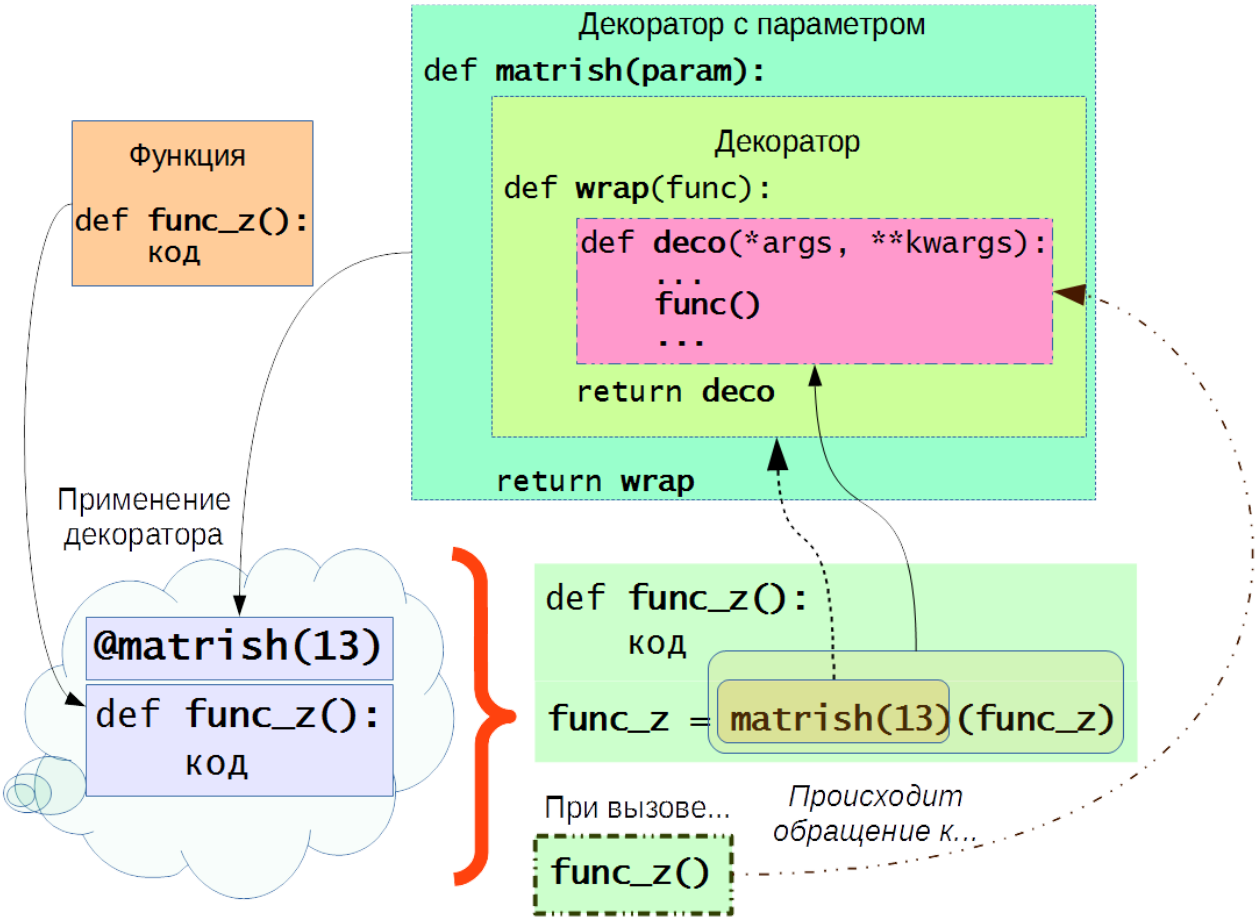
- Функция-декоратор определена в том же модуле, что и декорируемые функции. Настоящий декоратор обычно определяется в одном модуле, а применяется к функциям из других модулей.
- Декоратор register возвращает ту же функцию, что была передана в качестве аргумента. На практике декоратор обычно определяет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

Декоратор с параметрами

Как можно увидеть на примере функции functools.wraps, декораторы могут принимать аргументы. При надлежащем использовании эти аргументы могут использоваться, как параметры настройки, которые могут принимать индивидуальные значения для каждой декорируемой функции.

Для реализации декоратора 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:

```
import time
from functools import wraps

def sleep(timeout):
    def decorator(func):
        @wraps(func)
        def decorated(*args, **kwargs):
            ''' Д е к о р а т о р  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 применения данного декоратора вызов декорированной функции будет отложен 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
```

Это связано с тем, что декоратор применён к функции, которая использует рекурсию.

Декоратор и рекурсия

Нужно внимательно отслеживать использование совместно 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 какое-либо отношение к управлению системными ресурсами, такими как механизмы синхронизации или блокировки, от рекурсии лучше воздержаться.

Примеры декораторов

Приведём примеры декораторов в стандартной библиотеке Python и некоторых популярных фреймворках.

- @functools.lru_cache(maxsize=128, typed=False) сохраняет указанное количество результатов вызовов декорируемой функции (мемоизация).
- @functools.singledispatch позволяет реализовать функционал одиночной диспечеризации (single dispatch) - когда работа функции зависит от типа единственного аргумента, вместо создания большого ветвления if..elif..else, можно зарегистрировать несколько коротких функций, каждая из которых обрабатывает свой тип аргумента.
- @contextlib.contextmanager формирует менеджер контекста из декорируемой функции (рассматривается ниже в данном уроке).
- @abc.abstractmethod служит для указания абстрактного метода (будет рассмотрен на занятии, посвященном 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.
- Уже знакомые Вам декораторы библиотеке PyTest: @pytest.fixture, @pytest.yield_fixture, @pytest.mark.paramertize.

Резюме

- Замыкание - это функция, которая запоминает привязки свободных переменных на момент определения функции, так что их можно использовать впоследствии при вызове функции, когда область видимости, в которой она была определена, уже не существует.
- Декоратор - это функция, дополняющая другую функцию или класс.

- Декоратор может быть реализован через функцию, используя механизм замыкания, или через 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.

Менеджеры контекста и инструкция with

Протокол менеджера контекста

Надлежащее управление системными ресурсами, такими как файлы, блокировки и соединения, часто является достаточно сложной задачей, особенно в соединении с обработкой исключений. Например, возникшее исключение может заставить программу пропустить инструкции, отвечающие за освобождение критических ресурсов, таких как блокировки.

Инструкция with позволяет организовать выполнение последовательности инструкций внутри контекста, управляемого объектом, который играет роль менеджера контекста. Это позволяет предотвратить ошибки и уменьшить объем стереотипного кода, одновременно сделав API безопаснее и проще в использовании. Например:

```
# Оператор with при работе с файлами
with open("debuglog", "a") as f:
    f.write("Отладка\n")
    # инструкции
    f.write("Конец\n")

# Оператор with при работе с мьютексом
import threading

lock = threading.Lock()
with lock:
    # Начало критического блока
    # ... инструкции ...
    # Конец критического блока
```

В первом примере инструкция with автоматически закроет открытый файл, когда поток управления покинет блок инструкций, следующий ниже. Во втором примере инструкция with автоматически захватит блокировку, когда поток управления войдет в блок инструкций, следующий ниже, и освободит ее при выходе.

Программисты на Python находят много применений блокам with помимо автоматического закрытия файлов.

Объекты контекстных менеджеров служат для управления предложением with, точно так же, как итераторы управляют предложением for. Предложение with было задумано, для того чтобы упростить конструкцию try/finally, гарантирующую, что некоторая операция будет выполнена после блока, даже если этот блок прерван в результате исключения, предложения return или вызова sys.exit(). Код внутри

части finally обычно освобождает критически важный ресурс или восстанавливает временно измененное состояние.

Инструкция with obj позволяет объекту obj управлять происходящим, когда поток управления входит и покидает блок инструкций, следующий ниже. Когда интерпретатор встречает инструкцию with obj, он вызывает метод obj.__enter__(), чтобы известить объект о том, что был выполнен вход в новый контекст. Когда поток управления покидает контекст, вызывается метод obj.__exit__(type, value, traceback). Если при выполнении инструкций в блоке не возникло никаких исключений, во всех трех аргументах методу __exit__() передается значение None. В противном случае в них записываются тип исключения, его значение и трассировочная информация об исключении, вынудившем поток управления покинуть контекст. Метод __exit__() возвращает True или False, чтобы показать, было обработано исключение или нет (если возвращается значение False, возникшее исключение продолжит свое движение вверх за пределами контекста).

Инструкция with obj может принимать дополнительный спецификатор as var. В этом случае значение, возвращаемое методом obj.__enter__(), записывается в переменную var. Важно отметить, что в переменную `var` не обязательно будет записано значение `obj`.

Инструкция with способна работать только с объектами, поддерживающими протокол управления контекстом (методы __enter__() и __exit__()). Пользовательские классы также могут определять эти методы, чтобы реализовать собственный способ управления контекстом. Ниже приводится простой пример такого класса:

```
class ListTransaction:
    def __init__(self, thelist):
        self.thelist = thelist

    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy

    def __exit__(self, type, value, tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

Этот класс позволяет выполнять серию модификаций в существующем списке. При этом модификации вступят в силу, только если в процессе их выполнения не возникло исключения. В противном случае список останется в первоначальном состоянии. Например:

```
items = [1,2,3]
with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items)              # Выведет [1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("Немного смошенничаем!")
except RuntimeError:
```

```
    pass
print(items)              # Выведет [1,2,3,4,5]
```

Менеджер и объект контекста

Часть as в предложении with необязательна. В случае функции open она необходима, чтобы получить ссылку на файл, но некоторые контекстные менеджеры возвращают None за неимением чего-то полезного.

В следующем примере показана работа тестового контекстного менеджера, единственный смысл которого - подчеркнуть различие между самим менеджером и объектом, который возвращает его метод __enter__:

```
class LookingGlass:
    def __enter__(self):
        ''' Ради забавы заменим стандартный поток
вывода на "реверсивный" '''
        import sys
        self.original_write = sys.stdout.write
        sys.stdout.write = self.reverse_write
        return 'JABBER_WOCKY'

    def reverse_write(self, text):
        ''' В оригинальный поток вывода передается
перевёрнутая строка '''
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback):
        ''' При выходе из контекста вернём стандартный
поток вывода '''
        import sys
        sys.stdout.write = self.original_write
        if exc_type is ZeroDivisionError:
            print('Пожалуйста, НЕ НАДО делить на нуль!')
        return True

with LookingGlass() as what:
    print('Мы находимся в контексте')
    print(what)

print('Код за пределами контекста')
print(what)
```

В результате будет выведено:

```
етскетнок в ясмидохан ыМ
YKCOW_REBBAJ
Код за пределами контекста
JABBER_WOCKY
```

Таким образом, действие контекста - это замена потока вывода, а результат метода __enter__ - просто строка. После выхода из контекста подменённый поток вывода восстановлен на оригинальный. Это показывает также, что действие контекста может выходить за рамки некоего локального участка кода.

Модуль contextlib

Модуль contextlib упрощает реализацию собственных менеджеров контекста за счет обертывания функций-генераторов. Например:

```
from contextlib import contextmanager

@contextmanager
def ListTransaction(thelist):
      workingcopy = list(thelist)
      yield workingcopy
      # Изменить оригинальный список, только если не
возникло ошибок
      thelist[:] = workingcopy
```

В этом примере значение, передаваемое инструкции yield, используется, как возвращаемое значение метода __enter__(). После вызова метода __exit__() выполнение будет продолжено с инструкции, следующей за инструкцией yield. Если в контексте возникло исключение, оно проявится как исключение в функции-генераторе. При необходимости функция может перехватить и обработать исключение, но в данном случае оно продолжит свое распространение за пределы генератора и, возможно, будет обработано где-то в другом месте.

Примеры менеджеров контекста

Контекстные менеджеры появились сравнительно недавно, однако сообщество Python медленно, но верно находит им все новые изобретательные применения.

Приведем несколько примеров из стандартной библиотеки.

- Управление транзакциями в модуле sqlite3 (https://docs.python.org/3.6/library/sqlite3.html#using-the-connection-as-a-context-manager).
- Хранение блокировок, условных переменных и семафоров в модуле threading (https://docs.python.org/3.6/library/threading.html#using-locks-conditions-and-semaphores-in-the-with-statement).
- Настройка среды для арифметических операций с объектами Decimal (https://docs.python.org/3.6/library/decimal.html?highlight=decimal%20localcontex#decimal.localcontext).
- Внесение временных изменений в объекты для тестирования модуля unittest (https://docs.python.org/3.6/library/unittest.mock.html?highlight=unittest%20mock%20patch#unittest.mock.patch).

В стандартную библиотеку входят также утилиты contextiib, содержащий помимо рассмотренного выше декоратора следующие функции:

- contextlib.closing(thing) - возвращает менеджер контекста, который закрывает объект thing при выходе из блока. Например:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

- contextlib.suppress(*exceptions) - возвращает менеджер контекста, который ждёт появления какого-либо исключения из списка exceptions;
- contextlib.redirect_stdout(new_target) - менеджер контекста для временного перенаправления потока вывода sys.stdout в файло-подобный объект (или просто файл).

Работа с сетью (продолжение)

Модуль select

Модуль select предоставляет доступ к системным вызовам select() и poll(). Системный вызов select() обычно используется для реализации опроса, или мультиплексирования, обработки нескольких потоков ввода-вывода, без использования потоков управления или дочерних процессов. В системах UNIX эти вызовы можно использовать для работы с файлами, сокетами, каналами и со многими другими типами файлов. В Windows их можно использовать только для работы с сокетами.

- select(iwtd, owtd, ewtd [, timeout]) - запрашивает информацию о готовности к вводу, выводу и о наличии исключений для группы дескрипторов файлов. В первых трех аргументах передаются списки с целочисленными дескрипторами файлов или с объектами, обладающими методом fileno(), который возвращает дескриптор файла. Аргумент iwtd определяет список объектов, которые проверяются на готовность к вводу, owtd – список объектов, которые проверяются на готовность к выводу, и ewtd – список объектов, которые проверяются на наличие исключительных ситуаций. В любом из аргументов допускается передавать пустой список. В аргументе timeout передается число с плавающей точкой, определяющее предельное время ожидания в секундах. При вызове без аргумента timeout функция ожидает, пока хотя бы один из дескрипторов не окажется в требуемом состоянии. Если в этом аргументе передать число 0, функция просто выполнит опрос и тут же вернет управление. Возвращает кортеж списков с объектами, находящимися в требуемом состоянии. Эти списки включают подмножества объектов в первых трех аргументах. Если к моменту истечения предельного времени ожидания ни один из дескрипторов не находится в требуемом состоянии, возвращается три пустых списка. В случае ошибки возбуждается исключение select.error. В качестве значения исключения возвращается та же информация, что и в исключениях IOError и OSError.
- poll() - создает объект, выполняющий опрос с помощью системного вызова poll(). Эта функция доступна только в системах, поддерживающих системный вызов poll().

Рассмотрим пример эхо-сервера, обслуживающего одновременно нескольких клиентов, реализованный с использованием функции select:

```
import time
import select
from socket import socket, AF_INET, SOCK_STREAM

def new_listen_socket(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)
    sock.settimeout(0.2)   # Таймаут для операций с сокетом
    # Таймаут необходим, чтобы не ждать появления
данных в сокете
    return sock


def mainloop():
    ''' Основной цикл обработки запросов клиентов
    '''
    address = ('', 8888)
    clients = []
    sock = new_listen_socket(address)

    while True:
       try:
           conn, addr = sock.accept()  # Проверка подключений
      except OSError as e:
           pass                 # timeout вышел
      else:
          print("Получен запрос на соединение с %s" % str(addr))
           clients.append(conn)
      finally:
           # Проверить наличие событий ввода-вывода без
таймаута
           w = []
           try:
               r, w, e = select.select([], clients, [], 0)
           except Exception as e:
                        # Исключение произойдёт, если
какой-то клиент отключится
               pass       # Ничего не делать, если какой-то
клиент отключился

           # Обойти список клиентов, читающих из сокета
           for s_client in w:
           timestr = time.ctime(time.time()) + "\n"
           try:
               s_client.send(timestr.encode('ascii'))
           except:
               # Удаляем клиента, который отключился
               clients.remove(s_client)

print('Эхо-сервер запущен!')
```

```
mainloop()
```

Клиентское приложение теперь находится в постоянном опросе сервера:

```
from socket import *

s = socket(AF_INET, SOCK_STREAM) # Создать сокет TCP
s.connect(('localhost', 8888))   # Соединиться с сервером

while True:             # Постоянный опрос сервера
    tm = s.recv(1024)
    print("Текущее время: %s" % tm.decode('ascii'))

s.close()
```

Такой скрипт позволяет поэкспериментировать с запуском нескольких “клиентов”, чтобы увидеть как сервер будет обрабатывать клиентские подключения. Создадим служебный скрипт, запускающий несколько “клиентов” с использованием модуля subprocess:

```
from subprocess import Popen, CREATE_NEW_CONSOLE

p_list = []        # Список клиентских процессов

while True:
    user = input("Запустить 10 клиентов (s) / Закрыть
клиентов (x) / Выйти (q) ")

    if user == 'q':
        break
    elif user == 's':
        for _ in range(10):
            # Флаг CREATE_NEW_CONSOLE нужен для ОС Windows,
            # чтобы каждый процесс запускался в
отдельном окне консоли
            p_list.append(Popen('python time_client_random.py',
                                creationflags=CREATE_NEW_CONSOLE))

        print(' Запущено 10 клиентов')
    elif user == 'x':
        for p in p_list:
            p.kill()
        p_list.clear()
```

- Модуль subprocess будет рассмотрен на занятии, посвященном процессам и потокам.

Модуль socket

Знакомство с модулем socket было начато на прошлом занятии, но не были раскрыты подробности использования данного модуля.

Модуль socket предоставляет доступ к стандартному интерфейсу сокетов BSD. Хотя этот интерфейс разрабатывался для системы UNIX, тем не менее модуль socket может использоваться во всех платформах. Интерфейс сокетов разрабатывался для поддержки широкого разнообразия сетевых протоколов (IP, TIPC, Bluetooth и других). Однако наиболее часто используемым является протокол Интернета (Internet Protocol, IP), который включает в себя оба протокола, TCP и UDP. В языке Python поддерживаются обе версии протокола, IPv4 и IPv6, хотя версия IPv4 распространена намного шире.

Следует заметить, что этот модуль находится на достаточно низком уровне, обеспечивая непосредственный доступ к сетевым функциям, предоставляемым операционной системой. При разработке сетевых приложений может оказаться проще использовать специализированные модули, такие как ftplib, http, smtplib, urllib, xmlrpc, или модуль SocketServer, который будет рассмотрен на следующем занятии.

Семейства адресов

Некоторые функции из модуля socket требуют указывать семейство адресов. Семейство определяет, какой сетевой протокол будет использоваться. Ниже приводится 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 это встраиваемые системы). Семейства AF_NETLINK, AF_PACKET и AF_TIPC поддерживаются только в операционной системе Linux. Семейство AF_NETLINK используется для обеспечения скоростных взаимодействий между пользовательскими процессами и ядром Linux. Семейство AF_PACKET используется для прямого взаимодействия с уровнем передачи данных (например, для непосредственной работы с пакетами ethernet). Семейство AF_TIPC определяет протокол, используемый для обеспечения скоростных взаимодействий 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. Тип сокета определяет тип взаимодействий (потоки или пакеты) для использования с указанным семейством протоколов. Ниже приводится список констант, используемых для этой цели:

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

Наиболее часто в практике используются типы сокетов SOCK_STREAM и SOCK_DGRAM, потому что они соответствуют протоколам TCP и UDP в семействе протоколов Интернета (IP). Тип SOCK_RDM является разновидностью протокола UDP, обеспечивающей доставку дейтаграмм, но не гарантирующей порядок их доставки (дейтаграммы могут поступать получателю не в том порядке, в каком они отправлялись). Тип SOCK_SEQPACKET используется для отправки пакетов через соединения, ориентированные на передачу потока данных с сохранением порядка следования пакетов и их границ. Типы SOCK_RDM и SOCK_SEQPACKET не получили широкой поддержки, поэтому для обеспечения переносимости программ их лучше не использовать. Тип SOCK_RAW используется для низкоуровневого доступа к протоколу и может применяться для реализации специализированных операций, таких как отправка управляющих сообщений (например, сообщений ICMP). Обычно тип SOCK_RAW используется 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 тип SOCK_DGRAM или SOCK_RAW. Семейство AF_NETLINK поддерживает только тип SOCK_RAW.

Адресация

Чтобы обеспечить возможность взаимодействий через сокеты, 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). Ниже приводятся два примера 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 используется 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, таких как ‘www.python.org’, для их преобразования в IP-адреса будет использоваться служба доменных имен (DNS). В зависимости от настроек службы DNS программа может получать каждый раз различные IP-адреса. Чтобы избежать этого, в случае необходимости 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 адресах IPv4, за исключением того, что в числовой форме адрес IPv6 обычно задается строкой, состоящей из восьми шестнадцатеричных чисел, разделенных двоеточием, например: ‘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 одних 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:...’ и 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 едиый сайт (например, все компьютеры в подсети). В этом случае поле scopeid определяет числовой идентификатор сайта.

В случае отсутствия необходимости указывать значения полей flowinfo и scopeid адреса IPv6 можно указывать в виде кортежа (host, port), как и адреса IPv4.

Функции модуля *socket*

Ниже перечислены функции, которые определяются модулем 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, которые 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, которые передаются функции socket(). В поле canonname возвращается строка, представляющая каноническое имя хоста. В поле sockaddr возвращается кортеж с адресом сокета, как описывалось в предыдущем разделе, где описываются способы адресации. 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(). Например, 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() предназначена для широкого применения и 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. * 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, такой как ‘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 аргумент 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).

Для открытия соединения по протоколу UDP можно использовать 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 противном случае. Аргумент type может принимать только значение SOCK_DGRAM или SOCK_STREAM. Если в аргументе type передается значение SOCK_STREAM, создается объект, известный как 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 процесс получают возможность взаимодействовать друг с другом, используя эти сокеты. Также может использоваться для организации юнит-тестирования.

Экземпляры класса *socket.socket*

Сокет s класса socket обладает следующими методами:

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

- s.bind(address) - присваивает сокету указанный адрес address. Формат представления адреса зависит от семейства адресов. В большинстве случаев это кортеж вида (hostname, port). Для IP-адресов пустая строка представляет INADDR_ANY (любой адрес), а строка ‘<broadcast>’ представляет 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 используется, 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 вызывается также, 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. Формат адреса address зависит от семейства, к которому он относится, но обычно в этом аргументе передается кортеж (hostname, port). В случае ошибки возбуждает исключение socket.error. При подключении к серверу, выполняющемуся 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 возвращаемого значения зависит от используемого 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. Для получения параметров уровня сокета в нем передается значение SOL_SOCKET, а для получения параметров 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 аргумент buflen указан, он определяет максимальный размер буфера, куда должно быть записано значение параметра. Этот буфер возвращается методом в виде строки байтов. Интерпретация содержимого буфера с помощью модуля struct или других инструментов целиком возлагается на вызывающую программу.

В следующей таблице перечислены параметры сокета, поддерживаемые в языке Python. Большая часть этих параметров 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, ассоциированным со значением и используемым 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 возможность присваивания того же адреса и порта другому сокету. Этот параметр отключает действие параметра SO_REUSEADDR.
SO_KEEPALIVE	0, 1	Разрешает или запрещает периодическую передачу служебных сообщений другой стороне, для поддержания соединения в активном состоянии.
SO_LINGER	*linger*	Откладывает вызов метода *close()*, если в буфере передачи имеются данные. Значение типа linger представляет собой упакованную двоичную строку, содержащую 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 представляет собой упакованную двоичную строку, содержащую два 32-битных целых числа без знака (seconds, microseconds).
SO_REUSEADDR	0, 1	Разрешает или запрещает повторное использование локальных адресов.
SO_REUSEPORT	0, 1	Разрешает или запрещает нескольким процессам присваивать сокетам один и тот же адрес, при условии, что этот параметр установлено 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.

- s.gettimeout() - возвращает текущее значение предельного времени ожидания в секундах, если установлено. Возвращает число с плавающей точкой или None, если предельное время ожидания не было установлено.

- s.listen(backlog) - переводит сокет в режим ожидания входящих соединений. Аргумент backlog определяет максимальное количество запросов на соединение, ожидающих обработки, которые 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 возбуждает исключение.

- s.sendall(string [, flags]) - посылает данные через сетевое соединение; при этом, прежде чем вернуть управление, пытается отправить все данные. В случае успеха возвращает None; в случае ошибки возбуждает 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 противном случае устанавливается блокирующий режим (по умолчанию). При работе в неблокирующем режиме, если вызов метода recv() не находит никаких данных или вызов метода send() не может немедленно отправить данные, возбуждается 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. По истечении указанного предельного времени ожидания операции будут возбуждать исключение socket.timeout. Предельное время ожидания должно устанавливаться сразу же после создания сокета, так как оно также применяется к операциям, связанным с созданием логического соединения (таким как connect()).

- s.shutdown(how) - отключает одну или обе стороны соединения. Если в аргументе how передается значение 0, дальнейший прием данных будет запрещен. Если в аргументе how

передается значение 1, будет запрещена дальнейшая передача данных. Если в аргументе how передается значение 2, будут запрещены и прием, и передача данных.

На прошлом занятии приводился простой пример открытия TCP-соединения. Ниже приводится пример реализации простого эхо-сервера, возвращающего полученные данные, использующего протокол UDP:

```
# Сервер сообщений, действующий по протоколу UDP
# Принимает небольшие пакеты и отправляет их
 обратно
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('',10000))
while True:
    data, address = s.recvfrom(256)
    print('Получены данные с адреса %s' % str(address))
    s.sendto(b'echo:' + data, address)
```

Ниже приводится пример клиента, отправляющего сообщения серверу, реализация которого приводится выше:

```
# Клиент сообщений, действующий по протоколу UDP
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b'Hello World', ('', 10000))
resp, addr = s.recvfrom(256)
print(resp)
s.sendto(b'Spam', ('', 10000))
resp, addr = s.recvfrom(256)
print(resp)
s.close()
```

- Обратите внимание. Между операциями, выполняющимися в неблокирующем режиме, и операциями, предусматривающими выход по истечении предельного ожидания, существует важное отличие. Когда метод сокета выполняется в неблокирующем режиме, он возвращает управление немедленно, с признаком ошибки, если операция может быть заблокирована. Когда определено предельное время ожидания, методы возвращают признак ошибки, только если операция не была выполнена в течение указанного интервала времени.

Рассмотрим пример реализации эхо-сервера, обслуживающего нескольких клиентов с использованием функции select:

```
import select
from socket import socket, AF_INET, SOCK_STREAM

def read_requests(clients):
    ''' Чтение запросов из списка клиентов '''
    requests = {}  # Словарь ответов сервера вида {сокет:
 запрос}
```

```
    for sock in clients:
        try:
            data = sock.recv(1024).decode('ascii')
            responses[sock] = data
        except:
            print('Клиент {} {} отключился'.format(sock.fileno(),
sock.getpeername()))
            clients.remove(sock)
    return responses

def write_responses(requests, clients):
    ''' Эхо-ответ сервера клиентам, от которых были
запросы '''
    for sock in clients:
        if sock in requests:
            try:
                # Подготовить и отправить ответ сервера
                resp = requests[sock].encode('ascii')
                test_len = sock.send(resp.upper())
            except:             # Сокет недоступен, клиент
отключился
                print('Клиент {} отключился'.format(sock.getpeername()))
                sock.close()
                clients.remove(sock)

def mainloop():
    ''' Основной цикл обработки запросов клиентов '''
    address = ('', 10000)
    clients = []

    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    s.settimeout(0.2)   # Таймаут для операций с сокетом
    while True:
        try:
            conn, addr = s.accept()  # Проверка подключений
        except OSError as e:
            pass                          # timeout вышел
        else:
            print("Получен запрос на соединение с %s" %
str(addr))
            clients.append(conn)
        finally:
            # Проверить наличие событий ввода-вывода
            wait = 0
            r = []
            w = []
            try:
                r, w, e = select.select(clients, clients, [], wait)
            except:
                pass           # Ничего не делать, если какой-то
```

```
клиент отключился

            requests = read_requests(r)       # Сохраним запросы
клиентов
            write_responses(requests, w)      # Выполним отправку
ответов клиентам


print('Эхо-сервер запущен!')
mainloop()
```

Пояснения к коду:

1. Функции read_requests и write_responses достаточно просты:

· в read_requests осуществляется обход списка сокетов клиентов на чтение и формирование словаря запросов в формате {сокет: запрос};

· в write_responses осуществляется похожий обход списка сокетов на запись, отсылаемые ответы сервера формируются на основании словаря запросов requests.

2. Функция mainloop - основная функция сервера:

· открывает сокет на прослушивание и устанавливает таймаут для всех операций с сокетом;

· далее в цикле проверяет наличие новых подключений функцией accept() (при установленном таймауте accept() создаст исключение по истечении таймаута при отсутствии подключения) и добавляет новое подключение в список clients;

· посредством функции select() проверяет возможность чтения/записи из/в сокеты (список clients), select() возвращает кортеж из трёх списков - дескрипторы на чтение, на запись и имеющие исключительную ситуацию.

Код эхо-клиента, который отправляет “запрос” и сразу же читает ответ от сервера:

```
from socket import *
from select import select
import sys

ADDRESS = ('localhost', 10000)

def echo_client():
    # Начиная с Python 3.2 сокеты имеют протокол
менеджера контекста
    # При выходе из оператора with сокет будет
автоматически закрыт
    with socket(AF_INET, SOCK_STREAM) as sock: # Создать сокет TCP
        sock.connect(ADDRESS)   # Соединиться с сервером
        while True:
            msg = input('Ваше сообщение: ')
            if msg == 'exit':
                break
            sock.send(msg.encode('ascii'))    # Отправить!
```

```
            data = sock.recv(1024).decode('ascii')
            print('О т в е т :', data)

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

PyTest и сетевые приложения

После первого домашнего задания наверняка у многих слушателей возникали вопросы: до какой степени нужно тестировать код? нужно ли тестировать отправку данных в сокет (ведь в этом случае придётся написать полную копию клиентского и серверного приложений)?

С точки зрения тестирования могут быть выделены 2 основных момента:

1. модульное тестирование - тестирование на уровне функций и модулей, здесь нет необходимости в реализации полноценного сетевого окружения.
2. интеграционное тестирование - тестирование на уровне приложений и сервисов - здесь не обойтись без реализации простого, но полного сетевого окружения.

Для реализации модульного тестирования сетевых функций с использованием PyTest можно выделить несколько вариантов:

- использование фикстур для подмены класса socket;
- использование mock-объектов (monkeypatching) для подмены сокетов;
- использование модуля pytest-socket (необходимо установить отдельно), который блокирует обращение к объектам на основе типа socket, создавая исключение SocketBlockedError при любом обращении к сокетам.

Рассмотрим способ замены класса socket с использованием фикстур:

```
import pytest
import socket

def my_send(sock, msg):
    ''' Функция отправки сообщения в сокет
        Данная функция будет подвергнута тестированию
    '''
    msg = 'message: {}'.format(msg)
    msg = msg.encode('ascii')
    sock.send(msg)

class MySocket():
    ''' Класс-заглушка для операций с сокетом '''
    def __init__(self, sock_type=socket.AF_INET, sock_family=socket.SOCK_STREAM):
        self.data = b''
    def send(self, data):
        self.data = data
        return len(data)
    def recv(self, n):
        return b'Hello'

# Создадим фикстуру, которая заменяет класс socket
@pytest.yield_fixture
def my_socket():
    orig_sock = socket.socket
    socket.socket = MySocket
    yield
    socket.socket = orig_sock

def test_send(my_socket):
    ''' Тестирование отправки сообщения в сокет '''
    sock = socket.socket()
    my_send(sock, 'Hello')
    assert sock.data == b'message: Hello'
```

В данном примере ключевыми моментами оказываются класс-заглушка MySocket для замены сокетов, а также фикстура my_socket, которая реализует замену сокета по месту применения. Класс MySocket реализует две самые простые операции работы с сокетом - recv и send, реализует их так, как это было бы удобно для тестов.

Monkeypatching-подход в PyTest позволяет выполнять подмену любых объектов (функций, классов) на время работы функции (потоки ввода/вывода, сокеты). Так, предыдущий пример может быть реализован без использования фикстуры:

```
def test_send_data(monkeypatch):
      monkeypatch.setattr("socket.socket", MySocket)
      sock = socket.socket()
      my_send(sock, 'Test Data')
      assert sock.data == b'message: Test Data'
```

monkeypatch является стандартной фикстурой (pytest находит её автоматически) и создан специально для безопасной установки/удаления атрибута, элемента словаря, переменной окружения или изменения sys.path для импорта.

Так, например, при необходимости тестирования только функции отправки можно обойтись без класса MySocket и заменить напрямую функцию socket.send:

```
def test_send_data_2(monkeypatch):
      test_data = b''
      def monkey_send(self, data):
      nonlocal test_data
      test_data = data
      return len(data)

      monkeypatch.setattr("socket.socket.send", monkey_send)
      sock = socket.socket()
      my_send(sock, 'Test Data')
      assert test_data == b'message: Test Data'
```

В последующих занятиях будет продолжено изучение сетевого программирования в Python, а также работа с модулем PyTest.

Логгирование событий

В теме декораторов рассматривались примеры функций, обеспечивающих логгирование (журналирование) обращения к функциям. В реальных проектах наличие логгирования событий позволяет сэкономить много сил и времени разработчика при отладке, в случае необходимости устранения ошибок, а также оперативной технической поддержки. В Python в отличие от других языков программирования имеется стандартный модуль logging, обеспечивающий задачи логгирования событий.

Следует уделить внимание рассмотрению возможностей данного модуля.

Модуль *logging*

Модуль logging предоставляет гибкую возможность реализовать в приложениях журналирование событий, ошибок, предупреждений и отладочной информации. Эта информация может собираться, фильтроваться, записываться в файлы, отправляться в системный журнал и даже передаваться по сети на удаленные машины. Рассмотрим самые основные сведения об использовании этого модуля в наиболее типичных ситуациях.

Уровни журналирования

Основной задачей модуля logging является получение и обработка сообщений. Каждое сообщение состоит из некоторого текста и ассоциированного с ним уровня, определяющего его важность. Уровни имеют как символические, так и числовые обозначения:

Уровень	Значение	Описание
CRITICAL	50	Критические ошибки/сообщения
ERROR	40	Ошибки
WARNING	30	Предупреждения
INFO	20	Информационные сообщения
DEBUG	10	Отладочная информация
NOTSET	0	Уровень не установлен

Эти уровни являются основой для различных функций и методов в модуле logging. Например, существуют методы, которые различают уровни важности сообщений и выполняют фильтрацию, блокируя запись сообщений, уровень важности которых не соответствует заданному пороговому значению.

Базовая настройка

Перед использованием функций из модуля logging необходимо выполнить базовую настройку специального объекта, известного как корневой регистратор (Logger). Корневой регистратор содержит настройки по умолчанию, включая уровень журналирования, поток вывода, формат сообщений и другие параметры. Для выполнения настройки используется следующая функция:

* `basicConfig([**kwargs])` - выполняет базовую настройку корневого регистратора. Эта функция должна вызываться перед вызовом других функций из модуля `logging`. Она принимает множество именованных аргументов:

Именованный аргумент	Описание
filename	Журналируемые сообщения будут добавляться в файл с указанным именем.
filemode	Определяет режим открытия файла. По умолчанию используется режим *'a'* (добавление в конец).
format	Строка формата для формирования сообщений.
datefmt	Строка формата для вывода даты и времени.
level	Устанавливает уровень важности корневого регистратора. Обрабатываться будут сообщения с уровнем важности, равным или выше указанного. Сообщения с более низким уровнем просто будут игнорироваться.
stream	Определяет объект открытого файла, куда будут записываться журналируемые сообщения. По умолчанию используется поток *std.stderr*. Этот аргумент не может использоваться одновременно с аргументом *filename*.

Назначение большинства этих аргументов понятно по их названиям. Аргумент format используется для определения формата журналируемых сообщений с дополнительной контекстной информацией, такой как имена файлов, уровни важности, номера строк и так далее. Аргумент datefmt определяет формат вывода дат, совместимый с функцией time.strftime(). Если не определен, даты форматируются в соответствии со стандартом ISO8601.

В аргументе format допускается использовать следующие символы подстановки:

Формат	Описание
%(name)s	Имя регистратора.
%(levelno)s	Числовой уровень важности.
%(levelname)s	Символическое имя уровня важности.
%(pathname)s	Путь к исходному файлу, откуда была выполнена запись в журнал.
%(filename)s	Имя исходного файла, откуда была выполнена запись в журнал.
%(funcName)s	Имя функции, выполнившей запись в журнал.
%(module)s	Имя модуля, откуда была выполнена запись в журнал.
%(lineno)d	Номер строки, откуда была выполнена запись в журнал.
%(created)f	Время, когда была выполнена запись в журнал. Значением должно быть число, такое как возвращаемое функцией *time.time()*
%(asctime)s	Время в формате ASCII, когда была выполнена запись в журнал.
%(msecs)s	Миллисекунда, когда была выполнена запись в журнал.
%(thread)d	Числовой идентификатор потока выполнения.
%(threadName)s	Имя потока выполнения.
%(process)d	Числовой идентификатор процесса.
%(message)s	Текст журналируемого сообщения (определяется пользователем).

Рассмотрим пример, иллюстрирующий настройки для записи в журнал сообщений с уровнем INFO или выше:

```
import logging

logging.basicConfig(
       filename = "app.log",
       format = "%(levelname)-10s %(asctime)s %(message)s",
       level = logging.INFO
)
```

При таких настройках вывод сообщения “Hello World” с уровнем важности CRITICAL будет выглядеть в файле журнала ‘app.log’, как показано ниже:

```
CRITICAL   2017-09-15 17:33:18,080 Hello, World!
```

Объекты класса Logger

Чтобы выводить сообщения в журнал, необходимо получить объект класса Logger. В этом разделе описывается процесс создания, настройки и использования этих объектов.

Создание экземпляра класса *Logger*

Создать новый объект класса Logger можно с помощью следующей функции:

```
getLogger([logname])
```

- Возвращает экземпляр класса Logger с именем logname. Если объект с таким именем не существует, то создается и возвращается новый экземпляр класса Logger. В аргументе logname передается строка, определяющая имя или последовательность имен, разделенных точками (например, ‘app’ или ‘app.net’). При вызове без аргумента logname вернет объект Logger корневого регистратора.

Экземпляры класса Logger создаются иначе, чем экземпляры большинства классов в других библиотечных модулях. При создании объекта Logger с помощью функции getLogger() ей всегда необходимо передавать аргумент logname. За кулисами функция getLogger() хранит кэш экземпляров класса Logger вместе с их именами. Если в какой-либо части программы будет запрошен регистратор с тем же именем, она возвратит экземпляр, созданный ранее. Это существенно упрощает обработку журналируемых сообщений в крупных приложениях, потому что не приходится заботиться о способах передачи экземпляров класса Logger из одного модуля программы в другой. Вместо этого в каждом модуле, где возникает необходимость журналирования сообщений, достаточно просто вызвать функцию getLogger(), чтобы получить ссылку на соответствующий объект Logger.

Выбор имен

По причинам, которые станут очевидными чуть ниже, при использовании функции getLogger() желательно всегда выбирать говорящие имена. Например, если приложение называется ‘app’, тогда как минимум следует использовать getLogger(‘app’) в начале каждого модуля, составляющего приложение. Например:

```
import logging
log = logging.getLogger('app')
```

Можно также добавить имя модуля, например getLogger(‘app.net’) или getLogger(‘app.user’), чтобы более четко указать источник сообщений. Реализовать это можно с помощью инструкций, как показано ниже:

```
import logging
log = logging.getLogger('app.' + __name__)
```

Добавление имен модулей упрощает выборочное отключение или перенастройку механизма журналирования для каждого модуля в отдельности, как будет описано ниже.

Запись сообщений в журнал

Если переменная log является экземпляром класса Logger, то для записи сообщений с разными уровнями важности можно использовать следующие методы:

Уровень важности	Метод
CRITICAL	*log.critical(fmt [, *args [, exc_info [, extra]]])*
ERROR	*log.error(fmt [, *args [, exc_info [, extra]]])*
WARNING	*log.warning(fmt [, *args [, exc_info [, extra]]])*
INFO	*log.info(fmt [, *args [, exc_info [, extra]]])*
DEBUG	*log.debug(fmt [, *args [, exc_info [, extra]]])*

Параметры логгирования:

- Аргумент fmt - строка формата вывода сообщения в журнал.
- Аргументы в args будут служить параметрами спецификаторов формата в строке fmt. Для формирования окончательного сообщения из этих аргументов используется оператор форматирования строк %. Если передается несколько аргументов, они будут переданы оператору форматирования в виде кортежа. Если в качестве единственного аргумента передается словарь, имена ключей этого словаря можно использовать в строке формата.

Например:

```
log = logging.getLogger("app")

# Записать сообщение, используя позиционные
 аргументы форматирования
log.critical("Can't connect to %s at port %d", host, port)

# Записать сообщение, используя словарь значений
parms = { 'host' : 'www.python.org',
       'port' : 80
       }
log.critical("Can't connect to %(host)s at port %(port)d", parms)
```

- Именованный аргумент exc_info (True/False) определяет, добавлять ли в сообщение информацию об исключении, полученную вызовом sys.exc_info().
- Именованный аргумент extra определяет словарь с дополнительными значениями для использования в строке формата.

Выполняя вывод журналируемых сообщений, не следует использовать возможности форматирования строк при вызове функции (то есть когда сообщение сначала форматируется, а затем передается модулю logging). Например:

```
log.critical("Can't connect to %s at port %d" % (host, port))
```

В этом примере оператор форматирования строки всегда будет выполняться перед вызовом самой функции log.critical(), потому что аргументы должны передаваться функции или методу уже полностью вычисленными. Однако в более раннем примере значения для спецификаторов формата просто

передаются модулю logging и используются, только когда сообщение действительно будет выводиться. Это очень тонкое отличие, но так как в большинстве приложений задействуется механизм фильтрации сообщений или сообщения выводятся только в процессе отладки, первый подход обеспечивает более высокую производительность, когда журналирование отключено.

Фильтрование журналируемых сообщений

Каждый объект log класса Logger имеет свой уровень и обладает внутренним механизмом фильтрации, с помощью которого определяет, какие сообщения следует обрабатывать. Следующий метод используется для выполнения простой фильтрации на основе числового значения уровня важности сообщений:

* `log.setLevel(level)` - устанавливает уровень важности в объекте `log`, в соответствии со значением аргумента `level`. Обрабатываться будут только сообщения с уровнем важности равным или выше значения `level`. Все остальные сообщения попросту игнорируются. По умолчанию аргумент `level` получает значение `logging.NOTSET`, при котором обрабатываются все сообщения.

Обработка сообщений

Обычно сообщения обрабатываются корневым регистратором. Однако любой объект класса Logger может иметь свои специальные обработчики, принимающие и обрабатывающие сообщения. Реализовать это можно с помощью следующих методов экземпляра log класса Logger:

* `log.addHandler(handler)` - добавляет объект класса `Handler` в регистратор.

* `log.removeHandler(handler)`- удаляет объект класса `Handler` из регистратора.

В модуле logging имеется множество различных предопределенных обработчиков, выполняющих запись сообщений в файлы, потоки, в системный журнал и так далее. Следующий пример демонстрирует, как подключать обработчики к регистраторам с помощью указанных методов:

```
import logging
import sys

# Создать регистратор верхнего уровня с именем 'app'
app_log = logging.getLogger('app')
app_log.setLevel(logging.INFO)
app_log.propagate = False

# Добавить несколько обработчиков в регистратор 'app'
app_log.addHandler(logging.FileHandler('app.log'))
app_log.addHandler(logging.StreamHandler(sys.stderr))

# Отправить несколько сообщений. Они попадут в файл
app.log
# и будут выведены в поток sys.stderr
app_log.critical('Creeping death detected!')
app_log.info('FYI')
```

Чаще всего добавление собственных обработчиков сообщений в регистратор выполняется с целью переопределить поведение корневого регистратора. Именно по этой причине в примере был отключен

механизм распространения сообщений (то есть регистратор ‘app’ сам будет обрабатывать все сообщения).

Объекты класса Handler

Модуль logging предоставляет целую коллекцию предопределенных обработчиков для обработки сообщений различными способами. Эти обработчики добавляются в объекты класса Logger с помощью метода addHandler(). Кроме того, для каждого обработчика можно установить свой уровень важности и фильтры.

Встроенные обработчики

Ниже перечислены встроенные объекты обработчиков. Некоторые из них определяются в подмодулей logging.handlers, который должен импортироваться отдельно.

- handlers.DatagramHandler(host, port) - отправляет сообщения по протоколу UDP на сервер с именем host и в порт port. Сообщения кодируются с применением соответствующего объекта словаря LogRecord и переводятся в последовательную форму с помощью модуля pickle. Сообщение, передаваемое в сеть, состоит из 4-байтового значения длины (с прямым порядком следования байтов), за которым следует упакованная запись с данными. Чтобы реконструировать сообщение, приемник должен отбросить заголовок с длиной, прочитать сообщение, распаковать его содержимое с помощью модуля pickle и вызвать функцию logging.makeLogRecord(). Так как протокол UDP является ненадежным, ошибки в сети могут привести к потере сообщений.

- FileHandler(filename [, mode [, encoding [, delay]]]) - выводит сообщения в файл с именем filename. Аргумент mode определяет режим открытия файла и по умолчанию имеет значение ‘a’. В аргументе encoding передается кодировка. В аргументе delay передается логический флаг; если он имеет значение True, открытие файла журнала откладывается до появления первого сообщения. По умолчанию имеет значение False.

- handlers.HTTPHandler(host, url [, method]) - выгружает сообщения на сервер HTTP, используя метод HTTP GET или POST. Аргумент host определяет имя хоста, url – используемый адрес URL, а method – метод HTTP, который может принимать значение ‘GET’`(по умолчанию) или `’POST’. Сообщения кодируются с применением соответствующего объекта словаря LogRecord и преобразуются в переменные строки запроса URL с помощью функции urllib.urlencode().

- handlers.MemoryHandler(capacity [, flushLevel [, target]]) - этот обработчик используется для сбора сообщений в памяти и периодической передачи другому обработчику, который определяется аргументом target. Аргумент capacity определяет размер буфера в байтах. В аргументе flushLevel передается числовое значение уровня важности. Когда появляется сообщение с указанным уровнем или выше, это вынуждает обработчик передать содержимое буфера дальше. По умолчанию используется значение ERROR. В аргументе target передается объект класса Handler, принимающий сообщения. Если аргумент target опущен, вам придется определить объект-обработчик с помощью метода setTarget(), чтобы он мог выполнять обработку.

- handlers.RotatingFileHandler(filename [, mode [, maxBytes [, backupCount [,encoding [, delay]]]]]) - выводит сообщение в файл filename. Если размер этого файла превысит значение в аргументе maxBytes, он будет переименован в filename.1 и будет открыт новый файл с именем filename. Аргумент backupCount определяет максимальное количество резервных копий файла (по умолчанию имеет значение 0). При любом ненулевом значении будет выполняться циклическое переименование последовательности filename.1, filename.2, ..., filename.N, где filename.1 всегда представляет последнюю резервную копию, а filename.N – всегда самую старую. Режим mode определяет режим открытия файла журнала. По умолчанию аргумент mode имеет значение ‘a’.

Если в аргументе maxBytes передается значение 0 (по умолчанию), резервные копии файла журнала не создаются и размер его никак не будет ограничиваться. Аргументы encoding и delay имеют тот же смысл, что и в обработчике FileHandler.

- handlers.SMTPHandler(mailhost, fromaddr, toaddrs, subject [, credentials]) - отправляет сообщение по электронной почте. В аргументе mailhost передается адрес сервера SMTP, который сможет принять сообщение. Адрес может быть простым именем хоста, указанным в виде строки, или кортежем (host, port). В аргументе fromaddr передается адрес отправителя, в аргументе toaddrs – адрес получателя и в аргументе subject – тема сообщения. В аргументе credentials передается кортеж (username, password) с именем пользователя и паролем.

- handlers.SocketHandler(host, port) - отправляет сообщение удаленному хосту по протоколу TCP. Аргументы host и port определяют адрес получателя. Сообщения отправляются в том же виде, в каком их отправляет обработчик DatagramHandler. В отличие от DatagramHandler, этот обработчик обеспечивает надежную доставку сообщений.

- StreamHandler([fileobj]) - выводит сообщение в уже открытый объект файла fileobj. При вызове без аргумента сообщение выводится в поток sys.stderr.

- StreamHandler([fileobj]) Выводит сообщение в уже открытый объект файла fileobj. При вызове без аргумента сообщение выводится в поток sys.stderr.

- handlers.SysLogHandler([address [, facility]]) - передает сообщение демону системного журнала в системе UNIX. В аргументе address передается адрес хоста назначения в виде (host, port). Если этот аргумент опущен, используется адрес (‘localhost’, 514). В аргументе facility передается целочисленный код типа источника сообщения; аргумент по умолчанию принимает значение SysLogHandler.LOG_USER. Полный список кодов источников сообщений можно найти в определении обработчика SysLogHandler.

- handlers.TimedRotatingFileHandler(filename [, when [, interval [, backupCount [, encoding [, delay [, utc]]]]]]) - то же, что и RotatingFileHandler, но циклическое переименование файлов происходит через определенные интервалы времени, а не по достижении файлом заданного размера. В аргументе interval передается число, определяющее величину интервала в единицах, а в аргументе when – строка, определяющая единицы измерения. Допустимыми значениями для аргумента when являются: ‘S’ (секунды), ‘M’ (минуты), ‘H’ (часы), ‘D’ (дни), ‘W’ (недели) и ‘midnight’ (ротация выполняется в полночь). Например, если в аргументе interval передать число 3, а в аргументе when – строку ‘D’, ротация файла журнала будет выполняться каждые три дня. Аргумент backupCount определяет максимальное число хранимых резервных копий. В аргументе utc передается логический флаг, который определяет, должно ли использоваться локальное время (по умолчанию) или время по Гринвичу (UTC).

Форматирование сообщений

По умолчанию объекты класса Handler выводят сообщения в том виде, в каком они передаются функциям модуля logging. Однако иногда бывает необходимо добавить в сообщение дополнительную информацию, например время, имя файла, номер строки и так далее. Рассмотрим как можно реализовать автоматическое добавление этой информации в сообщения.

Объекты форматирования

Прежде чем изменить формат сообщения, необходимо создать объект класса Formatter: * Formatter([fmt [, datefmt]]) - создает новый объект класса Formatter. В аргументе fmt передается строка формата сообщения. В строке fmt допускается использовать любые символы подстановки, перечисленные в описании функции basicConfig(). В аргументе datefmt передается строка форматирования дат в виде,

совместимом с функцией time.strftime(). Если этот аргумент опущен, форматирование дат осуществляется в соответствии со стандартом ISO8601.

Чтобы задействовать объект класса Formatter, его необходимо подключить к обработчику. Делается это с помощью метода h.setFormatter() экземпляра h класса Handler.

- h.setFormatter(format) - подключает объект форматирования, который будет использоваться экземпляром h класса Handler при создании сообщений. В аргументе format должен передаваться объект класса Formatter.

Рассмотрим пример настройки форматирования сообщений в обработчике:

```
import logging
import sys

# Определить формат сообщений
_format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# Создать обработчик, который выводит сообщения с
 уровнем CRITICAL в поток stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(_format)
```

В этом примере нестандартный объект форматирования подключается к обработчику crit_hand. Если этому обработчику передать сообщение, такое как ‘Oghr! Kernel panic!’, он выведет следующий текст:

```
CRITICAL   2017-09-16 18:16:55,267 Oghr! Kernel panic!
```

Настройка механизма журналирования

Настройка приложения на использование модуля logging обычно выполняется в несколько основных этапов:

1. С помощью функции getLogger() создается несколько объектов класса Logger. Соответствующим образом устанавливаются значения параметров, таких как уровень важности.
2. Создаются объекты обработчиков различных типов (таких как FileHandler, StreamHandler, SocketHandler и так далее) и устанавливаются соответствующие уровни важности.
3. Создаются объекты класса Formatter и подключаются к объектам Handler с помощью метода setFormatter().
4. С помощью метода addHandler() объекты Handler подключаются к объектам Logger.

Каждый этап может оказаться достаточно сложным, поэтому лучше всего поместить реализацию настройки механизма журналирования в одном, хорошо документированном, месте. Например, можно создать файл applogconfig.py, который будет импортироваться основным модулем приложения:

```
# applogconfig.py
import logging
import sys

# Определить формат сообщений
format = logging.Formatter('%(levelname)-10s %(asctime)s %(message)s')

# Создать обработчик, который выводит сообщения с
 уровнем CRITICAL в поток stderr
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)

# Создать обработчик, который выводит сообщения в
 файл
applog_hand = logging.FileHandler('app.log')
applog_hand.setFormatter(format)

# Создать регистратор верхнего уровня с именем 'app'
app_log = logging.getLogger('app')
app_log.setLevel(logging.INFO)
app_log.addHandler(applog_hand)
app_log.addHandler(crit_hand)

# Изменить уровень важности для регистратора 'app.net'
logging.getLogger('app.net').setLevel(logging.ERROR)
```

Если в какой-либо части процедуры настройки потребуется что-то изменить, легче будет учесть все нюансы, если вся процедура будет реализована в одном месте. Имейте в виду, что этот специальный файл должен импортироваться только один раз и только в одном месте программы. В других модулях программы, где потребуется выводить журналируемые сообщения, достаточно просто добавить следующие строки:

```
import logging
app_log = logging.getLogger('app')
...
app_log.critical('An error occurred')
```

Вопросы производительности

Добавление в приложение механизма журналирования может существенно ухудшить его производительность, если не отнестись к этому с должным вниманием. Однако существуют некоторые приемы, которые могут помочь уменьшить это отрицательное влияние.

Во-первых, при запуске в оптимизированном режиме (-O) удаляется весь программный код, который выполняется в условных инструкциях, таких как:

```
if __debug__: инструкции
```

Если отладка – единственная цель использования модуля logging, можно поместить все вызовы механизма журналирования в условные инструкции, которые автоматически будут удаляться при компиляции в оптимизированном режиме.

Второй прием заключается в использовании “пустого” объекта Null вместо объектов Logger, когда журналирование должно быть полностью отключено. Этот прием отличается от использования None тем, что основан на использовании объектов, которые просто пропускают все обращения к ним. Например:

```
class Null(object):
    def __init__(self, *args, **kwargs): pass
    def __call__(self, *args, **kwargs): return self
    def __getattribute__(self, name): return self
    def __setattr__(self, name, value): pass
    def __delattr__(self,name): pass

log = Null()
log.critical("An error occurred.")           # Ничего не делает
```

Журналированием можно также управлять с помощью декораторов и метаклассов. Поскольку эти две особенности потребляют время только на этапе, когда Python интерпретирует определения функций, методов и классов, они позволяют добавлять и удалять поддержку журналирования в различных частях программы и избавиться от потери производительности, когда журналирование выключено.

Резюме

- Стандартная библиотека Python включает модуль, обеспечивающий логгирование - нет необходимости изобретать что-то своё.
- Модуль logging имеет множество параметров настройки, которые не обсуждались на данном занятии. За дополнительными подробностями слушателям следует обращаться к официальной документации.
- Модуль logging может использоваться в многопоточных программах. В частности, нет никакой необходимости окружать операциями блокировки программный код, который выводит журналируемые сообщения.

Итоги

На данном занятии была рассмотрена тема декораторов, протокола менеджера контекстов, продолжено знакомство с сетевым программированием, а также затронуты вопросы обеспечения логгирования событий.

Для понимания механизма работы декораторов важно понимать различия между этапом импорта и этапом выполнения, а также разбираться в областях действия переменных и замыканиях. Свободное владение замыканиями и объявлением nonlocal важно не только при написании декораторов, но и при разработке событийно-ориентированных программ с графическим интерфейсом, а также для асинхронного ввода-вывода без обратных вызовов.

Впереди ещё много интересного! Продолжаем интенсивную работу!

Домашнее задание

Проект “Мессенджер”

Основное задание:

- Реализовать логгирование с использованием модуля logging:
    - Реализовать декоратор @log, фиксирующий обращение к декорируемой функции: сохраняет имя функции и её аргументы.
    - Настройку логгера выполнить в отдельном модуле log_config.py:
        - Создание именованного логгера.
        - Сообщения лога должны иметь следующий формат:

"<дата-время> <уровень_важности> <имя_модуля> <имя_функции> <сообщение>"

- Журналирование должно производиться в лог-файл.
- На стороне сервера необходимо настроить ежедневную ротацию лог-файлов
- Реализовать обработку нескольких клиентов на сервере с использованием функции select таким образом, что клиенты общаются в "общем чате", т.е. каждое сообщение каждого клиента отправляется всем клиентам, подключенным к серверу.
- Реализовать функции отправки/приёма данных на стороне клиента. Для упрощения разработки приложения на данном этапе пусть клиентское приложение будет либо только принимать, либо только отправлять сообщения в общий чат:
- запуск скрипта клиента должен осуществляться с параметром командной строки: -r (чтение чата) или -w (передача сообщений в чат).
- Для всех функций необходимо написать тесты.

Дополнительно:

- Реализовать скрипт, запускающий два клиентских приложения - одно на чтение чата, другое на запись в чат (уместно использовать модуль subprocess).
- Реализовать скрипт, запускающий указанное количество клиентских приложений.
- Для ОС UNIX реализовать обработку выбора ввода данных от пользователя с клавиатуры и чтение из сокета с использованием функции select.
- В декораторе @log реализовать фиксацию функции, из которой была вызвана декорированная функция. Т.е. если имеется код:

```
@log
   def func_z():
    pass

def main():
    func_z()
```

То в логе должна быть отражена информация:

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

Дополнительные материалы

1. Logging Cookbook
2. Python Decorators Library - This page is meant to be a central repository of decorator code pieces.
3. PEP 343 – The “with” Statement
4. Awesome Python Decorator - A curated list of awesome python decorator resources
5. Mocks and Monkeypatching in Python
6. Понимаем декораторы в Python’е, 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.
2. Лучано Ромальо. “Python. К вершинам мастерства”.
3. Дэвид Бизли. “Python. Подробный справочник”