Клиент-серверные приложения на Python

Урок 5

Логирование

Журналирование событий и модуль logging.

Логирование событий

Логирование может применяться для журналирования обращения к функциям. В реальных проектах оно позволяет сэкономить много сил и времени разработчика при отладке кода и устранении ошибок, а также оперативной технической поддержке. В Python, в отличие от других языков программирования, для логирования есть стандартный модуль **logging**.

Модуль **logging**

Позволяет гибко журналировать события, ошибки, предупреждения и отладочную информацию в приложениях. Эти сведения могут собираться, фильтроваться, записываться в файлы, отправляться в системный журнал и даже передаваться по сети на удаленные машины. Рассмотрим основные аспекты использования этого модуля в наиболее типичных ситуациях.

Уровни журналирования

Основная задача модуля **logging** — получать и обрабатывать сообщения. Они состоят из 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 — основа для функций и методов в модуле **logging**. Есть методы, которые различают уровни важности сообщений и выполняют фильтрацию: блокируют запись сообщений, уровень важности которых не соответствует заданному пороговому значению.

Базовая настройка

Перед использованием функций из модуля **logging** необходимо выполнить базовую настройку корневого регистратора (**Logger**). Он содержит настройки по умолчанию: уровень журналирования, поток вывода, формат сообщений и другие параметры. Настраивают **Logger** через функцию **basicConfig([**kwargs])**. Эта функция должна вызываться первой 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 указанному или выше его. Сообщения с более низким уровнем 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 **filename**.

Назначение большинства этих аргументов понятно по их названиям. **format** определяет формат журналируемых сообщений с дополнительной контекстной информацией — именами файлов, уровнями важности, номерами строк. Аргумент **datefmt** определяет формат вывода дат, совместимый с функцией **time.strftime()**. Если он 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, когда была выполнена запись в журнал, в формате **ASCII**

%(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 (файл **examples/01_logging/01_logging_basic.py**):

```
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** так:

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

Объекты класса Logger

Чтобы выводить сообщения в журнал, необходимо получить объект класса **Logger**. Разберемся, как создавать, настраивать и использовать его.

Создание экземпляра класса *Logger*

Создать новый объект класса **Logger** можно с помощью следующей функции:

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

Она возвращает экземпляр класса **Logger** с именем **logname**. Если объект с таким именем не существует, 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 **Logger** создаются иначе, чем у большинства классов в других библиотечных модулях. При добавлении 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 вернет 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 вызвать функцию **getLogger()**, чтобы получить ссылку на соответствующий объект **Logger**.

Пример создания экземпляра класса **Logging** (файл **examples/01_logging/01_logging_basic.py**):

```
log = logging.getLogger('basic')
```

Выбор имен

При использовании функции **getLogger()** желательно всегда выбирать говорящие имена. 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:

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

Добавление имен модулей упрощает выборочное отключение или перенастройку механизма журналирования для каждого модуля в отдельности.

Запись сообщений в журнал

Если переменная **log** является экземпляром класса **Logger**, 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]]])

Пример записи сообщений в журнал (файл **examples/01_logging/01_logging_basic.py**):

```
log.info('Hello, World!')
log.warning('It seems to be a bug...')
log.critical('Critical bug in app! Hello, World!')
```

Параметры логирования:

- 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, оператор форматирования получит их в виде кортежа. Если в качестве единственного аргумента передается словарь, имена его ключей 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 при вызове **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 (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 примере, приведенном выше, значения для спецификаторов формата просто передаются модулю **logging** и используются, только когда сообщение действительно будет выводиться. Это тонкое отличие, но так как в большинстве приложений задействуется механизм фильтрации сообщений или они 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. Метод **log.setLevel(level)** используется для выполнения простой фильтрации на основе числового значения уровня важности сообщений. Он устанавливает этот уровень в объекте **log** в соответствии со значением аргумента **level**. Обрабатываться будут только сообщения с уровнем важности, равным значению **level** или выше его. Все остальные сообщения 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** есть множество 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 собственные обработчики сообщений добавляются в регистратор, чтобы переопределить поведение корневого регистратора. Поэтому в примере был отключен механизм распространения сообщений (то есть регистратор **app** сам будет обрабатывать их все).

Объекты класса Handler

Модуль **logging** предоставляет коллекцию предопределенных обработчиков для сообщений. Они добавляются в объекты класса **Logger** с помощью метода **addHandler()**. Кроме того, для каждого обработчика можно установить уровень важности и фильтры.

Встроенные обработчики

Ниже перечислены встроенные объекты обработчиков. Некоторые из них определяются в подмодуле **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** ненадежен, 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 **delay** — логический флаг; если он имеет значение **True**, открытие файла журнала откладывается до появления первого сообщения. По умолчанию у него 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** задает 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 его размер 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 равен 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** – самую старую. Режим **mode** определяет механизм 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 **toaddrs** — получателя, а в **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 уходят в том же виде, в каком их отправляет обработчик **DatagramHandler**. В отличие от него, **SocketHandler** обеспечивает надежную доставку сообщений;

- **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 **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 **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**. В сообщение можно добавить дополнительную информацию — например, время, имя файла, номер строки. Рассмотрим, как можно реализовать это автоматически.

Объекты форматирования

Прежде чем изменить формат сообщения, необходимо создать объект класса **Formatter** с помощью **Formatter([fmt [, datefmt]])**. 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, даты форматируются в соответствии со стандартом ISO8601.

Чтобы задействовать объект класса **Formatter**, его необходимо подключить к обработчику. Метод **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 (файл **examples/01_logging/02_logging_formatter.py**):

```
# 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 регистратор
log = logging.getLogger('basic')

# Добавить обработчик к регистратору
```

```
log.addHandler(crit_hand)
# Передать сообщение обработчику
log.critical('Oghr! Kernel panic!')
```

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 для использования модуля **logging** обычно выполняется в несколько основных этапов:

1. С помощью функции **getLogger()** создается несколько объектов класса **Logger**. Соответствующим образом устанавливаются значения параметров — например, уровня важности.

2. Создаются объекты обработчиков различных типов (**FileHandler**, **StreamHandler**, **SocketHandler** и других), 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 поместить реализацию настройки механизма журналирования в одном и хорошо документированном месте. Можно создать файл с реализацией журналирования, который будет импортироваться основным модулем приложения:

```
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)
```

Чтобы было легче учесть все нюансы, когда при настройке потребуются изменения, всю процедуру следует реализовывать в одном месте. Этот специальный файл должен импортироваться только единожды и в одном месте программы. В других модулях, где потребуется выводить журналируемые сообщения, достаточно просто добавить следующие строки:

```
import logging

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

Еще один пример, в котором представлен программный код файла, реализующего журналирование (файл **examples/01_logging/log_config.py**):

```
# logging - стандартный модуль для организации логирования
import logging
# Можно выполнить более расширенную настройку логирования.

# Создаем объект-логгер с именем app.main:
logger = logging.getLogger('app.main')

# Создаем объект форматирования:
formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s ")

# Создаем файловый обработчик логирования (можно задать кодировку):
fh = logging.FileHandler("app.main.log", encoding='utf-8')
fh.setLevel(logging.DEBUG)
fh.setFormatter(formatter)

# Добавляем в логгер новый обработчик событий и устанавливаем уровень
логирования
logger.addHandler(fh)
logger.setLevel(logging.DEBUG)

if __name__ == '__main__':
    # Создаем потоковый обработчик логирования (по умолчанию sys.stderr):
    console = logging.StreamHandler()
    console.setLevel(logging.DEBUG)
    console.setFormatter(formatter)
    logger.addHandler(console)
    logger.info('Тестовый запуск логирования')
```

Этот файл может быть запущен как самостоятельное приложение. Результат его работы:

```
2018-06-06 17:34:21,790 - INFO - Тестовый запуск Логирования
```

Он также может быть импортирован (файл **examples/01_logging/03_use_log_config.py**):

```
import logging
import log_config

# Обратите внимание, логгер уже создан в модуле log_config,
# теперь нужно его просто получить
logger = logging.getLogger('app.main')

def main():
    ''' Тестовая главная функция
    '''
    logger.debug('Старт приложения')

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

Данный пример показывает, что сначала создается файл, реализующий логику журналирования (**log_config.py**), после чего он доступен из любого модуля программы. Для этого в модуле (одном) он импортируется. А для доступа к самому регистратору (**logger**) выполняется несложная инструкция:

```
logger = logging.getLogger('app.main')
```

Это позволяет легко отправить необходимое сообщение в журнал из текущего модуля:

```
logger.debug('Старт приложения')
```

Вопросы производительности

Добавив в приложение механизм журналирования, можем существенно ухудшить его производительность, если не отнесемся к этому с должным вниманием. Есть приемы, которые помогут избежать негативных последствий.

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

```
if __debug__: инструкции
```

Если модуль **logging** используется только для отладки, можно поместить все вызовы механизма журналирования в условные инструкции, которые автоматически будут удаляться при компиляции в оптимизированном режиме.

Второй прием — использовать «пустой» объект **Null** вместо объектов **Logger**, когда журналирование должно быть полностью отключено. Этот способ отличается от применения **None** тем, что основан на объектах, 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 управлять и с помощью декораторов и метаклассов. Они потребляют время только на этапе, когда Python интерпретирует определения функций, методов и классов, поэтому позволяют добавлять и удалять поддержку журналирования в различных частях программы и не терять в производительности, когда журналирование отключено.

Резюме

- Стандартная библиотека Python включает модуль, обеспечивающий логирование — нет необходимости изобретать что-то свое;
- Модуль **logging** имеет множество параметров настройки, которые не обсуждались на данном занятии. За дополнительными подробностями обращайтесь к официальной документации;
- Модуль **logging** может использоваться в многопоточных программах. Не надо окружать операциями блокировки программный код, который выводит журналируемые сообщения.

Логирование помогает быть в курсе событий, связанных с работой приложения. Это полезно как для пользователей программы, так и для разработчика. Логи позволяют отслеживать алгоритм работы программы, получать информацию об ошибках и данные, формируемые на конкретном этапе выполнения приложения. По сути, логирование протоколирует события при отладке, поиске ошибок, диагностике программы.

Практическое задание

Для проекта «Мессенджер» реализовать логирование с использованием модуля **logging**:

1. В директории проекта создать каталог **log**, в котором для клиентской и серверной сторон в отдельных модулях формата **client_log_config.py** и **server_log_config.py** создать логгеры;
2. В каждом модуле выполнить настройку соответствующего логгера по следующему алгоритму:
    a. Создание именованного логгера;
    b. Сообщения лога должны иметь следующий формат: **"<дата-время> <уровень_важности> <имя_модуля> <сообщение>"**;
    c. Журналирование должно производиться в лог-файл;
    d. На стороне сервера необходимо настроить ежедневную ротацию лог-файлов.
3. Реализовать применение созданных логгеров для решения двух задач:
    a. Журналирование обработки исключений **try/except**. Вместо функции **print()** использовать журналирование и обеспечить вывод служебных сообщений в лог-файл;

b. Журналирование функций, исполняемых на серверной и клиентской сторонах при работе мессенджера.

Дополнительные материалы

1. Logging Cookbook.
2. Логирование в Python.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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