Kurs z zakresu Python PCAP i PCPP

Dr inż. Marcin Caryk

Logowanie w pythonie

Logging in Python

Logging in Python - wstęp

- Python Standard Library udostępnia przydatny moduł o nazwie logging, który rejestruje zdarzenia występujące w aplikacji.
- Logi są najczęściej używane do znalezienia przyczyny błędu.
- Domyślnie Python i jego moduły udostępniają wiele poziomós logów informujących o przyczynach błędów. Jednak dobrą praktyką jest tworzenie własnych logów, które mogą być przydatne dla Ciebie lub innych programistów.
- W Pythonie możesz przechowywać logi w różnych miejscach. Najczęściej ma postać pliku, ale może to być również strumień wyjściowy, a nawet usługa zewnętrzna.
- Aby rozpocząć logowanie musimy zaimportować odpowiedni moduł: `import logging`

Logging in Python - opis

- Standardowa biblioteka Pythona udostępnia moduł loggowania jako rozwiązanie do rejestrowania zdarzeń z aplikacji i bibliotek.
- Po skonfigurowaniu rejestratora staje się on częścią procesu interpretera języka Python, który uruchamia kod. Innymi słowy, ma charakter globalny.
- Podsystem logowania w języku Python można również skonfigurować przy użyciu zewnętrznego pliku konfiguracyjnego.
- Specyfikacje formatu konfiguracji logowania można znaleźć w standardowej bibliotece języka Python.
- Biblioteka logowania jest oparta na podejściu modułowym i obejmuje kategorie komponentów: **rejestratory, programy obsługi, filtry i formatery**.
    - **Rejestratory** ujawniają interfejs, z którego bezpośrednio korzysta kod aplikacji.
    - **Programy** obsługi wysyłają rekordy dziennika (utworzone przez rejestratory) do odpowiedniego miejsca docelowego.
    - **Filtry** zapewniają bardziej szczegółowe narzędzie do określania, które rekordy dziennika mają zostać wydrukowane.
    - **Formatery** określają układ rekordów dziennika w końcowym wyniku.

Logging in Python - opis

- Te liczne obiekty programu loggującego są zorganizowane w drzewo reprezentujące różne części systemu i różne zainstalowane biblioteki innych firm.
- Kiedy wysyłasz komunikat do jednego z programów loggujących, komunikat jest wysyłany do wszystkich programów obsługi tego logger przy użyciu programu formatującego, który jest dołączony do każdego modułu obsługi.
- Komunikat jest następnie propagowany w górę drzewa loggera, aż dotrze do głównego loggera lub loggera wyżej w drzewie, który jest skonfigurowany za pomocą propagate=False.

Logging in Python – nejlepsze praktyki

- Ustawianie nazw poziomów: Pomaga to w utrzymywaniu własnego słownika komunikatów dziennika i zmniejsza możliwość błędów literowych.
- logging.getLevelName(logging_level) zwraca tekstową reprezentację ważności o nazwie logging_level. Wstępnie zdefiniowane wartości obejmują, od najwyższej do najniższej istotności:

1. CRITICAL
2. ERROR
3. WARNING
4. INFO
5. DEBUG

- Logowanie z wielu modułów: jeśli masz różne moduły i musisz wykonać inicjalizację w każdym module przed logowaniem komunikatów, możesz użyć kaskadowego nazewnictwa rejestratora

logging.getLogger(“coralogix”)

logging.getLogger(“coralogix.database”)

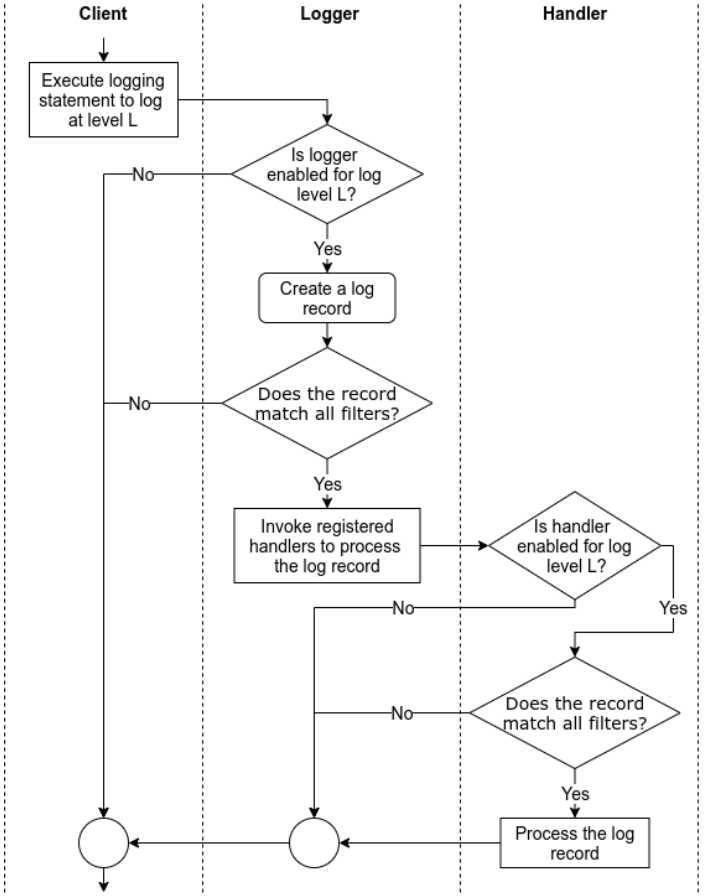
logging.getLogger(“coralogix.client”)

- Tworząc coralogix.client i coralogix.database jako potomków loggera coralogix i przekazując do niego ich komunikaty, umożliwia to łatwe logowanie wielomodułowe.
- Jest to jeden z pozytywnych skutków ubocznych nazwy w przypadku, gdy struktura bibliotek modułów odzwierciedla architekturę oprogramowania.

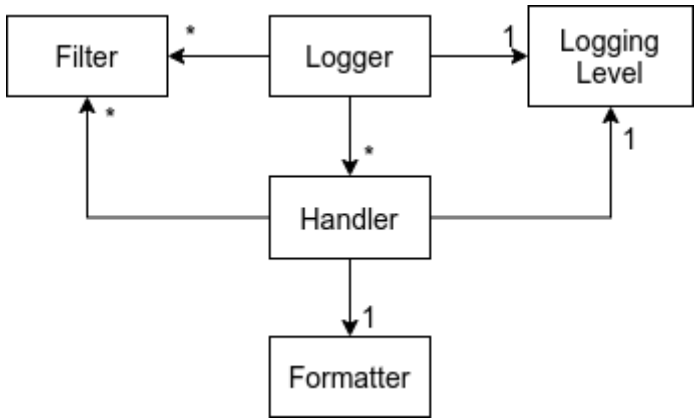
Logging in Python – bazowy concept

Gdy korzystamy z biblioteki logowania, wykonujemy/uruchamiamy następujące typowe zadania, korzystając z powiązanych pojęć:

- Klient wysyła żądanie logów (**log request**), wykonując instrukcję logowania (**logging statement**). Często takie instrukcje rejestrowania wywołują funkcję/metodę w interfejsie API logowania (biblioteki) (**logging (library) API**), dostarczając dane loggera (**log data**) i poziom loggera (**logging level**) jako argumenty. Poziom rejestrowania określa ważność żądania loggera. Dane loggera (**log message**) to często komunikat loggera, który jest ciągiem znaków wraz z dodatkowymi danymi do zarejestrowania. Często interfejs API rejestrowania jest udostępniany za pośrednictwem obiektów loggera.
- Aby umożliwić przetwarzanie żądania przechodzącego przez bibliotekę loggera, biblioteka loggera tworzy rekord dziennika, który reprezentuje żądanie dziennika i przechwytuje odpowiednie dane loggera
- W zależności od konfiguracji biblioteki loggowania (poprzez konfigurację logowania) biblioteka logowania filtruje żądania/rekordy dziennika. To filtrowanie polega na porównaniu żądanego poziomu logowania z progowym poziomem rejestrowania i przepuszczaniu rekordów dziennika przez filtry udostępniane przez użytkownika.
- Programy obsługi (**Handlers**) przetwarzają przefiltrowane rekordy loggera w celu zapisania danych loggera (np. zapisania danych loggera do pliku) lub wykonania innych czynności związanych z danymi loggera (np. wysłania wiadomości e-mail z danymi loggera). W niektórych bibliotekach loggowania, przed przetworzeniem rekordów loggów, program obsługi może ponownie filtrować zapisy loggera w oparciu o poziom loggowania programu obsługi i filtry (**filters**) specyficzne dla programu dostarczone przez użytkownika. Ponadto, w razie potrzeby, procedury obsługi często polegają na formaterach (**formatters**) dostarczonych przez użytkownika, aby sformatować rekordy dziennika w ciągi, tj. wpisy logów (**log entries**).



- Standardowa biblioteka Pythona oferuje wsparcie dla logowania poprzez moduły **logging, logging.config** i **logging.handlers**.
- **logging** zapewnia główny interfejs API skierowany do klienta.
- moduł **logging.config** zapewnia interfejs API do konfigurowania logowania w kliencie.
- Moduł **logging.handlers** zapewnia różne procedury obsługi, które obejmują typowe sposoby przetwarzania i przechowywania rekordów dziennika.
- Zbiorowo określamy te moduły jako bibliotekę loggowania Pythona.



Logging in Python – bazowy concept

- W języku Python obsługuje pięć poziomów rejestrowania: critical, error, warning, info i debug. Poziomy te są oznaczone stałymi o tej samej nazwie w module logowania, tj. logging.CRITICAL, logging.ERROR, logging.WARNING, logging.INFO i logging.DEBUG. Wartości tych stałych wynoszą odpowiednio 50, 40, 30, 20 i 10.
- W czasie wykonywania wartość liczbowa poziomu logowania określa znaczenie poziomu logowania. W związku z tym klienci mogą wprowadzać nowe poziomy logowania, używając jako poziomów logowania wartości liczbowych większych niż 0 i nierównych wstępnie zdefiniowanym poziomom logowania.
- Poziomy logowania mogą mieć nazwy. Gdy dostępne są nazwy, poziomy logowania są wyświetlane według nazw we wpisach dziennika. Każdy predefiniowany poziom logowania ma taką samą nazwę jak nazwa odpowiadającej mu stałej, dlatego pojawiają się one pod nazwami we wpisach dziennika, np. logowanie.WARNING, a 30 poziomów pojawia się jako „WARNING”.
- Niestandardowe poziomy logowania są domyślnie nienazwane. Tak więc nienazwany niestandardowy poziom logowania z wartością liczbową n pojawia się we wpisach dziennika jako „Poziom n”, co skutkuje niespójnymi i nieprzyjaznymi dla człowieka wpisami dziennika. Aby temu zaradzić, można nazwać niestandardowy poziom logowania za pomocą funkcji na poziomie modułu logging.addLevelName(level, levelName). Na przykład przy użyciu funkcji logging.addLevelName(33, „CUSTOM1”) poziom 33 zostanie zapisany jako „CUSTOM1”.

Logging in Python – bazowy concept

- **Debug**: użyj logging.DEBUG, aby logować szczegółowe informacje, zwykle przydatne tylko podczas diagnozowania problemów, np. podczas uruchamiania aplikacji.
- **Info**: użyj logging.INFO, aby potwierdzić, że oprogramowanie działa zgodnie z oczekiwaniami, np. po pomyślnym zainicjowaniu aplikacji.
- **Warning**: Użyj logging.WARNING, aby zgłosić nieoczekiwane zachowania lub wskazujące na przyszłe problemy, ale nie mające wpływu na bieżące działanie oprogramowania, np. gdy aplikacja wykryje małą ilość pamięci, co może wpłynąć na przyszłe działanie aplikacji.
- **Error**: użyj logging.ERROR, aby zgłosić, że oprogramowanie nie wykonało jakiejś funkcji, np. gdy aplikacja nie może zapisać danych z powodu niewystarczających uprawnień.
- **Critical**: Użyj logging.CRITICAL, aby zgłosić poważne błędy, które mogą uniemożliwić dalsze działanie oprogramowania, np. gdy aplikacja nie może przydzielić pamięci.

Logging in Python – Python Logger - tworzenie

- Obiekty logging.Logger oferują podstawowy interfejs do biblioteki rejestrowania. Obiekty te udostępniają metody rejestrowania do wydawania żądań dziennika wraz z metodami wykonywania zapytań i modyfikowania ich stanu.
- Fabryczna funkcja logging.getLogger(nazwa) jest zwykle używana do tworzenia loggerów. Korzystając z funkcji fabrycznej, klienci mogą polegać na bibliotece w celu zarządzania loggerami i uzyskiwania dostępu do loggerów za pomocą ich nazw zamiast przechowywania i przekazywania odniesień do loggera.
- Argumentem name w funkcji fabrycznej jest zazwyczaj hierarchiczna nazwa oddzielona kropkami, np. a.b.c. Ta konwencja nazewnictwa umożliwia bibliotece utrzymywanie hierarchii loggerów. W szczególności, gdy funkcja fabryczna tworzy logger, biblioteka zapewnia istnienie loggera dla każdego poziomu hierarchii określonego przez nazwę, a każdy logger w hierarchii jest połączony z jego nadrzędnymi i podrzędnymi loggerem.

Logging in Python – Python Logger – elemety loggera

- Każdy logger ma progowy poziom loggowania, który określa, czy żądanie dziennika powinno zostać przetworzone. Rejestrator przetwarza żądanie dziennika, jeśli wartość liczbowa żądanego poziomu loggera jest większa lub równa wartości liczbowej progowego poziomu rejestrowania loggera.
- Klienci mogą pobierać i zmieniać progowy poziom rejestrowania loggera za pomocą odpowiednio metod **Logger.getEffectiveLevel()** i **Logger.setLevel(level).** Kiedy funkcja fabryczna jest używana do tworzenia loggera, funkcja ustawia progowy poziom rejestrowania loggera na progowy poziom rejestrowania jego loggera nadrzędnego określony przez jego nazwę.

  Logger.critical(msg, *args, **kwargs)

  Logger.error(msg, *args, **kwargs)

  Logger.debug(msg, *args, **kwargs)

  Logger.info(msg, *args, **kwargs)

  Logger.warn(msg, *args, **kwargs)

- Oprócz powyższych metod, rejestratory oferują również dwie następujące metody:

  Logger.log(level, msg, *args, **kwargs) wysyła żądania loggera z jawnie określonymi poziomami logowania. Ta metoda jest przydatna w przypadku korzystania z niestandardowych poziomów logowania.

  Logger.exception(msg, *args, **kwargs) wysyła żądania loggera z poziomem rejestrowania ERROR, które przechwytują bieżący wyjątek jako część wpisów dziennika. W związku z tym klienci powinni wywoływać tę metodę tylko z programu obsługi wyjątków.

- Argumenty msg i args w powyższych metodach są łączone w celu utworzenia komunikatów loggera przechwytywanych przez wpisy dziennika. Wszystkie powyższe metody obsługują argument słowo kluczowe exc_info w celu dodania informacji o wyjątkach do wpisów logów oraz informacje o stosie i poziom stosu w celu dodania informacji o stosie wywołań do wpisów dziennika.
- Obsługują również dodatkowy argument słowa kluczowego, który jest słownikiem, aby przekazywać wartości istotne dla filtrów, programów obsługi i formaterów.

Logging in Python – Python Logger - elemety loggera

- Poza poziomami logowania filtry zapewniają dokładniejszy sposób filtrowania żądań logów na podstawie informacji zawartych w rekordzie dziennika, np. ignorują żądania dziennika wydawane w określonej klasie.
- Klienci mogą dodawać i usuwać filtry do/z rejestratorów, używając odpowiednio metod Logger.addFilter(filter) i Logger.removeFilter(filter).

Logging Filters

- Dowolna funkcja lub funkcja wywoływalna, która akceptuje argument rekordu logów i zwraca zero, aby odrzucić rekord, i wartość różną od zera, aby zaakceptować rekord, może służyć jako filtr. Każdy obiekt, który oferuje metodę z filtrem podpisu (record: LogRecord) -> int może również służyć jako filtr.
- Podklasa logging.Filter(nazwa: str), która opcjonalnie zastępuje metodę logging.Filter.filter(record) może również służyć jako filtr. Bez przesłonięcia metody filtrowania, taki filtr będzie dopuszczał rekordy emitowane przez loggery, które mają taką samą nazwę jak filtr i są dziećmi filtra (na podstawie nazwy loggerów i filtra). Jeśli nazwa filtru jest pusta, filtr przepuszcza wszystkie rekordy. Jeśli metoda jest przesłonięta, to powinna zwrócić wartość zero, aby odrzucić rekord i wartość niezerową, aby przyjąć rekord.

Logging Handler

- Obiekty logging.Handler wykonują końcowe przetwarzanie rekordów dziennika, tj. Logowanie żądań dziennika. To końcowe przetwarzanie często przekłada się na przechowywanie zapisu dziennika, np. zapisanie go do logów systemowych lub plików. Może to również przetłumaczyć, aby przekazać dane rekordu loggera określonym podmiotom (np. wysłać wiadomość e-mail) lub przekazać zapis dziennika innym podmiotom w celu dalszego przetwarzania .
- Podobnie jak logger, programy obsługi mają próg rejestrowania, który można ustawić za pomocą metody theHandler.setLevel(level). Obsługują również filtry za pomocą metod Handler.addFilter(filter) i Handler.removeFilter(filter).
- Klienci mogą ustawić formater dla programu obsługi za pomocą metody Handler.setFormatter(formatter). Jeśli program obsługi nie ma programu formatującego, używa domyślnego programu formatującego dostarczonego przez bibliotekę.

Logging in Python – Python Logger - elemety loggera

- Moduł logging.handler zapewnia bogatą kolekcję 15 użytecznych procedur obsługi, które obejmują wiele typowych przypadków użycia. Tak więc tworzenie instancji i konfigurowanie tych procedur obsługi jest wystarczające w wielu sytuacjach.

  https://docs.python.org/3/howto/logging.html#useful-handlers

- W sytuacjach wymagających obsługi niestandardowej programiści mogą rozszerzyć klasę Handler lub jedną z predefiniowanych klas Handler, implementując metodę Handler.emit(record) w celu zarejestrowania podanego rekordu logera.

Logging Formatter

- Programy obsługi używają obiektów logging.Formatter do formatowania rekordu dziennika na wpis dziennika oparty na ciągu znaków.
- Program formatujący działa poprzez łączenie pól/danych w rekordzie loggó z ciągiem formatu określonym przez użytkownika.
- W przeciwieństwie do programów obsługi, biblioteka rejestrowania udostępnia jedynie podstawowy program formatujący, który rejestruje żądany poziom logów, nazwę loggera i komunikat dziennika. Tak więc, poza prostymi przypadkami użycia, klienci muszą tworzyć nowe formatery, tworząc obiekty logging.Formatter z niezbędnymi ciągami formatującymi.
- Formatery obsługują trzy style ciągów formatujących:

  printf, e.g., '%(levelname)s:%(name)s:%(message)s'

  str.format(), e.g., '{levelname}:{name}:{message}'

  str.template, e.g., '$levelname:$name:$message'

- Ciąg formatujący formatera może odnosić się do dowolnego pola obiektów LogRecord, w tym pól opartych na kluczach dodatkowego argumentu metody logowania.
- Przed sformatowaniem rekordu loggera program formatujący używa metody LogRecord.getMessage() do skonstruowania komunikatu dziennika przez połączenie argumentów msg i args metody logowania (przechowywanej w rekordzie loggera) przy użyciu operatora formatowania ciągu znaków (%).
- Następnie program formatujący łączy wynikowy komunikat dziennika z danymi w rekordzie dziennika, używając określonego ciągu formatu, aby utworzyć wpis dziennika.

Logging in Python – Python Logger - elemety loggera

Logging Module

- Aby zachować hierarchię loggerów, gdy klient korzysta z biblioteki loggowania, biblioteka tworzy główny logger, który służy jako element główny hierarchii rejestratorów. Domyślnym progowym poziomem logowania głównego programu logującego jest logging.WARNING.
- Moduł oferuje wszystkie metody logowania oferowane przez klasę Logger jako funkcje na poziomie modułu o identycznych nazwach i podpisach, np. logging.debug(msg, *args, **kwargs).
- Jeśli główny program loggujący nie ma żadnych procedur obsługi podczas obsługi żądań loggera wysyłanych za pośrednictwem tych metod, biblioteka logowania dodaje instancję logging.StreamHandler opartą na strumieniu sys.stderr jako procedurę obsługi do głównego programu logującego.
- Gdy rlogger bez programów obsługi odbierają żądania logów, biblioteka logowania kieruje takie żądania dziennika do ostatniego programu obsługi, którym jest instancja logging.StreamHandler oparta na strumieniu sys.stderr. Ta procedura obsługi jest dostępna za pośrednictwem atrybutu logging.lastResort.

Logging in Python - zagnieżdzenie

- Jedna aplikacja może mieć kilka loggerów stworzonych zarówno przez nas, jak i przez programistów modułów.
- Jeśli aplikacja jest prosta, można użyć root loggera. W tym celu wywołaj funkcję getLogger bez podawania nazwy. Główny logger znajduje się w najwyższym punkcie hierarchii.
- Jego miejsce w hierarchii jest przydzielane na podstawie nazw przekazywanych do funkcji getLogger.
- Nazwy loggerów są podobne do nazw modułów Pythona, w których używany jest separator kropek. Ich format jest następujący:
- hello – tworzy logger, który jest dzieckiem roota loggera
- hello.world – tworzy logger, który jest dzieckiem hello loggera.
- Jeśli chcesz wykonać kolejne zagnieżdżenie, należy użyć separatora kropek.
- Funkcja getLogger zwraca obiekt Logger
- Zalecamy wywołanie funkcji getLogger z argumentem __name__, który jest zastępowany nazwą bieżącego modułu. Pozwala to w łatwy sposób określić źródło logowanej wiadomości.

```
import logging

logger = logging.getLogger()
hello_logger = logging.getLogger('hello')
hello_world_logger = logging.getLogger('hello.world')
recommended_logger = logging.getLogger(__name__)
```

Logging in Python – poziomy logowania

- Obiekt Logger umożliwia tworzenie logów o różnych poziomach rejestrowania, które pomagają odróżnić mniej ważne logi od tych, które zgłaszają poważny błąd. Domyślnie zdefiniowane są następujące poziomy logowania:

Nazwa	Wartość
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

- Każdy poziom ma nazwę i wartość liczbową. Można też zdefiniować własny poziom, ale te oferowane przez moduł logowania są w zupełności wystarczające. Obiekt Logger ma metody, które ustawiają poziom rejestrowania.
- Wszystkie powyższe metody wymagają podania komunikatu, który będzie widoczny w logach. Domyślny format dziennika obejmuje poziom, nazwę rejestratora i zdefiniowaną wiadomość. Należy zauważyć, że wszystkie te wartości są oddzielone dwukropkiem.
- Komunikaty z poziomu INFO i DEBUG nie są wyświetlane, wynika to z domyślnej konfiguracji (basicConfig)

```
import logging

logging.basicConfig()

logger = logging.getLogger()

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

```
CRITICAL:root:Your CRITICAL message
ERROR:root:Your ERROR message
WARNING:root:Your WARNING message
```

Logging in Python – podstawowa konfiguracja

- Podstawowa konfiguracja logowania odbywa się za pomocą metody basicConfig. Wywołanie metody basicConfig (bez podania argumentów) tworzy obiekt StreamHandler, który przetwarza logi, a następnie wyświetla je w konsoli.
- Obiekt StreamHandler jest tworzony przez domyślny obiekt Formatter odpowiedzialny za format logu. Domyślny format składa się z nazwy poziomu, nazwy rejestratora i zdefiniowanego komunikatu.
- Na koniec nowo utworzony program obsługi jest dodawany do głównego programu rejestrującego.
- Korzystając z metody basicConfig, możesz zmienić poziom logowania (tak samo jak przy użyciu metody setLevel), a nawet lokalizację logów.
- W przykładzie metoda basicConfig przyjmuje trzy argumenty. Pierwszym z nich jest poziom logowania równy CRITICAL, co oznacza, że przetwarzane będą tylko komunikaty z tym poziomem
- Przekazanie nazwy pliku do drugiego argumentu tworzy obiekt FileHandler (zamiast obiektu StreamHandler). Po ustawieniu argumentu nazwa pliku wszystkie logi będą kierowane do podanego pliku.
- Dodatkowo przekazanie ostatniego argumentu filemode wartością 'a' (jest to tryb domyślny) oznacza, że do tego pliku zostaną dołączone nowe logi. Jeśli chcemy zmienić ten tryb, można użyć innych trybów, które są analogiczne do tych używanych we wbudowanej funkcji open.
- Metoda basicConfig zmienia konfigurację głównego programu rejestrującego i jego elementów podrzędnych, które nie mają zdefiniowanego własnego modułu obsługi.

```
import logging

logging.basicConfig(level=logging.CRITICAL, filename='prod.log',
filemode='a')

logger = logging.getLogger()

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Wynik jest w pliku `prod.log`

```
CRITICAL:root:Your CRITICAL message
```

Logging in Python – podstawowa konfiguracja

- Przedstawiona wcześniej metoda basicConfig może również służyć do zmiany domyślnego formatowania loggera. Odbywa się to za pomocą argumentu format, który można zdefiniować za pomocą dowolnych znaków lub atrybutów obiektu LogRecord.
- Definiowany przez nas format jest tworzony przez połączenie atrybutów obiektu LogRecord oddzielonych dwukropkiem. Obiekt LogRecord jest automatycznie tworzony przez rejestrator podczas logowania. Zawiera wiele atrybutów, takich jak nazwa loggera, poziom logowania, a nawet numer linii, w której wywoływana jest metoda logowania.
- Wiecej można przeczytać na https://docs.python.org/3/library/logging.html#logrecord-attributes
- %(name)s – ten wzorzec zostanie zastąpiony nazwą loggera wywołującego metodę logowania. W naszym przypadku jest to główny rejestrator;
- %(levelname)s – ten wzorzec zostanie zastąpiony ustawionym poziomem logowania. W naszym przypadku jest to poziom CRITICAL;
- %(asctime)s — ten wzorzec zostanie zastąpiony czytelnym dla człowieka formatem daty, który wskazuje, kiedy utworzono obiekt LogRecord. Wartość dziesiętna jest wyrażona w milisekundach;
- %(message)s – ten wzorzec zostanie zastąpiony zdefiniowanym komunikatem. W naszym przypadku jest to „Your CRITICAL message”\
- Ogólnie schemat użycia argumentu obiektu LogRecord w argumencie format wygląda następująco: `(LOG_RECORD_ATTRIBUTE_NAME)s`

```
import logging

FORMAT =
'%(name)s:%(levelname)s:%(asctime)s:%(message)s'

logging.basicConfig(level=logging.CRITICAL, filename='prod.log',
filemode='a', format=FORMAT)

logger = logging.getLogger()

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Wynik jest w pliku `prod.log`

```
root:CRITICAL:2023-01-08 09:40:03,962:Your
CRITICAL message
```

Logging in Python – własny handler

- Każdy rejestrator może zapisywać logi w różnych lokalizacjach, a także w różnych formatach. Aby to zrobić, musisz zdefiniować własny moduł obsługi i formater.
- W większości przypadków zapisujemy swoje loggi w pliku. Moduł logowania posiada klasę FileHandler, która ułatwia to zadanie.
- Podczas tworzenia obiektu FileHandler należy podać nazwę pliku, w którym będą zapisywane dzienniki.
- Dodatkowo można przekazać tryb pliku z argumentem mode, np. mode='a'. W kolejnym kroku należy ustawić poziom logowania, który będzie przetwarzany przez handler. Domyślnie nowo utworzony handler jest ustawiony na poziom NOTSET. Możesz to zmienić za pomocą metody setLevel.
- Na koniec musisz dodać utworzony moduł obsługi do swojego rejestratora za pomocą metody addHandler.
- Do każdego rejestratora można dodać kilka programów obsługi. Jeden program obsługi może zapisywać dzienniki w pliku, a inny może wysyłać je do usługi zewnętrznej. Aby móc przetwarzać komunikaty o poziomie niższym niż WARNING przez dodane handlery, konieczne jest ustawienie progu tego poziomu w root loggerze.

```
import logging

logger = logging.getLogger(__name__)

handler = logging.FileHandler('prod.log', mode='w')
handler.setLevel(logging.CRITICAL)

logger.addHandler(handler)

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Wynik jest w pliku `prod.log`

```
Your CRITICAL message
```

Logging in Python – własny formater

- W pierwszym kroku tworzymy obiekt Formatter, przekazując zdefiniowany format do jego konstruktora.
- Następnym krokiem jest ustawienie formatera w obiekcie obsługi. Odbywa się to za pomocą metody setFormatter.
- Po wykonaniu tej czynności można przeanalizować swoje loggi w pliku prod.log

```
import logging

FORMAT =
'%(name)s:%(levelname)s:%(asctime)s:%(message)s'

logger = logging.getLogger(__name__)

handler = logging.FileHandler('prod.log', mode='w')
handler.setLevel(logging.CRITICAL)

formatter = logging.Formatter(FORMAT)
handler.setFormatter(formatter)

logger.addHandler(handler)

logger.critical('Your CRITICAL message')
logger.error('Your ERROR message')
logger.warning('Your WARNING message')
logger.info('Your INFO message')
logger.debug('Your DEBUG message')
```

Wynik jest w pliku `prod.log`

```
__main__:CRITICAL:2019-10-10 20:40:05,119:Your
CRITICAL message
```

Logging in Python – przykład 1

1. Utwórz rejestrator o nazwie „app”
2. Ustaw progowy poziom logowania logera na INFO.
3. Utwórz procedurę obsługi opartą na strumieniu, która zapisuje wpisy loggera w standardowym strumieniu błędów.
4. Ustaw progowy poziom loggowania programu obsługi na INFO.
5. Utwórz formatter do przechwytywania

czas żądania dziennika jako liczba sekund od epoki,

poziom logowania żądania,

nazwa loggera,

nazwę modułu wystawiającego żądanie logowania,

komunikat dziennika.

6. Ustaw utworzony formater jako format handlera.
7. Dodaj utworzony handler do tego loggera.

```
import logging
import sys
import os

def _init_logger():
    logger = logging.getLogger('app')  #1
    logger.setLevel(logging.INFO)  #2
    handler = logging.StreamHandler(sys.stderr)  #3
    handler.setLevel(logging.INFO)  #4
    formatter = logging.Formatter(
            '%(created)f:%(levelname)s:%(name)s:%(module)s:%(message)s') #5
    handler.setFormatter(formatter)  #6
    logger.addHandler(handler)  #7

_init_logger()
_logger = logging.getLogger('app')


_logger.info('App started in %s', os.getcwd())
```

Logging in Python – przykład 2

```
import logging
import datetime

class _LoggingHandler(object):

    def __init__(self):
        #super().__init__()
        # Init Logger
        self.logger = logging.getLogger(__name__)
        # add new levels
        self.add_log_levels()
        # setup Logger
        self.__setup_logger()

    def add_log_levels(self):
        self.__add_logger_level_debug()
        self.__add_logger_level_info()
        self.__add_logger_level_warning()
        self.__add_logger_level_critical()
        self.__add_logger_level_error()
        self.__add_logger_level_note()
        self.__add_logger_level_trace()
        self.__add_logger_level_pass()
        self.__add_logger_level_fail()
```

```
def __add_logger_level_debug(self):
    logging.addLevelName(logging.DEBUG, '%-8s' %
logging.getLevelName(logging.DEBUG))

def __add_logger_level_info(self):
    logging.addLevelName(logging.INFO, '%-8s' %
logging.getLevelName(logging.INFO))

def __add_logger_level_warning(self):
    logging.addLevelName(logging.WARNING, '%-8s' %
logging.getLevelName(logging.WARNING))

def __add_logger_level_error(self):
    logging.addLevelName(logging.ERROR, '%-8s' %
logging.getLevelName(logging.ERROR))

def __add_logger_level_critical(self):
    logging.addLevelName(logging.CRITICAL, '%-8s' %
logging.getLevelName(logging.CRITICAL))
```

Logging in Python – przykład 2

```
def __add_logger_level_note(self):
    logging.NOTE = 11
    logging.addLevelName(logging.NOTE, '%-8s' % 'NOTE')
    self.logger.note = lambda msg, *args: self.logger._log(logging.NOTE,
msg, args)

def __add_logger_level_trace(self):
    logging.TRACE = 12
    logging.addLevelName(logging.TRACE, '%-8s' % 'TRACE')
    self.logger.trace = lambda msg, *args: self.logger._log(logging.TRACE,
msg, args)

def __add_logger_level_pass(self):
    logging.OK = 13
    logging.addLevelName(logging.OK, '%-8s' % 'OK')
    self.logger.ok = lambda msg, *args: self.logger._log(logging.OK, msg,
args)

def __add_logger_level_fail(self):
    logging.FAIL = 14
    logging.addLevelName(logging.FAIL, '%-8s' % 'FAIL')
    self.logger.fail = lambda msg, *args: self.logger._log(logging.FAIL,
msg, args)
```

```
def set_logger_formatter(self):
    # setup formatter
    self.formatter = logging.Formatter('%(levelname)s %(asctime)-8s -
%(message)s',
                                       "%Y-%m-%d %H:%M:%S")

def set_sys_stream_handler(self):
    # create console handler
    handler = logging.StreamHandler()
    # set handler level info
    handler.setLevel(logging.DEBUG)
    # add formatter to handler
    handler.setFormatter(self.formatter)
    # add handler to logger
    self.logger.addHandler(handler)
```

Logging in Python – przykład 2

```
def  set_file_handler(self, mfilename = 'robot'):
    dfilename = datetime.datetime.now().strftime("%Y%m%d-
%H%M%S")
    # create file handler
    filename = '{}.{}'.format(mfilename,'log')
    fhandler = logging.FileHandler(filename)
    # set handler level info
    fhandler.setLevel(logging.DEBUG)
    # add formatter to handler
    fhandler.setFormatter(self.formatter)
    # add handler to logger
    self.logger.addHandler(fhandler)

def __setup_logger(self):
    self.set_logger_formatter()
    # setup Level
    self.logger.setLevel(logging.DEBUG)
    self.set_sys_stream_handler()
    self.set_file_handler()
```

Logging in Python – przykład 2

```
log_handler = _LoggingHandler()

class RobotLogHandler():

    @staticmethod
    def debug(msg):
        log_handler.logger.debug(msg)

    @staticmethod
    def info(msg):
        log_handler.logger.info(msg)

    @staticmethod
    def warning(msg):
        log_handler.logger.warning(msg)

    @staticmethod
    def critical(msg):
        log_handler.logger.critical(msg)

    @staticmethod
    def error(msg):
        log_handler.logger.error(msg)

    @staticmethod
    def note(msg):
        log_handler.logger.note(msg)
```

```
    @staticmethod
    def trace(msg):
        log_handler.logger.trace(msg)

    @staticmethod
    def ok(msg):
        log_handler.logger.ok(msg)

    @staticmethod
    def fail(msg):
        log_handler.logger.fail(msg)

    @staticmethod
    def set_log_file_name(filename):
        log_handler.set_file_handler(filename)
```

Logging in Python – przykład 2

```python
if __name__ == '__main__':

    RobotLogHandler.debug('debug message')
    RobotLogHandler.info('info message')
    RobotLogHandler.warning('warn message')
    RobotLogHandler.error('error message')
    RobotLogHandler.critical('critical message')
    RobotLogHandler.note('note message')
    RobotLogHandler.trace('trace message')
    RobotLogHandler.ok('ok message')
    RobotLogHandler.fail('fail message')
```

```
DEBUG    2023-01-08 10:56:59 - debug message
INFO     2023-01-08 10:56:59 - info message
WARNING  2023-01-08 10:56:59 - warn message
ERROR    2023-01-08 10:56:59 - error message
CRITICAL 2023-01-08 10:56:59 - critical message
NOTE     2023-01-08 10:56:59 - note message
TRACE    2023-01-08 10:56:59 - trace message
OK       2023-01-08 10:56:59 - ok message
FAIL     2023-01-08 10:56:59 - fail message
```

robot.log

```
DEBUG    2023-01-08 11:07:31 - debug message
INFO     2023-01-08 11:07:31 - info message
WARNING  2023-01-08 11:07:31 - warn message
ERROR    2023-01-08 11:07:31 - error message
CRITICAL 2023-01-08 11:07:31 - critical message
NOTE     2023-01-08 11:07:31 - note message
TRACE    2023-01-08 11:07:31 - trace message
OK       2023-01-08 11:07:31 - ok message
FAIL     2023-01-08 11:07:31 - fail message
```

Logging in Python – przykład 3

- Do tworzenia logów wykorzystywane są LogRecord atrybuty
- Można o nich poczytać pod linkiem https://docs.python.org/3/library/logging.html#logrecord-attributes
- Można tworzyć również własne atrybuty

```
import logging

wlasne_dane = "Testowanie Loggera"

def main() -> None:
    logging.basicConfig(
        format="[%(asctime)s] [%(wlasne_dane)s] [%(name)s] %(levelname)s:
%(message)s", level=logging.DEBUG)

    old_factory = logging.getLogRecordFactory()

    def record_factory(*args: object, **kwargs: object) -> logging.LogRecord:
        global wlasne_dane
        record = old_factory(*args, **kwargs)

        record.wlasne_dane = wlasne_dane

        return record

    logging.setLogRecordFactory(record_factory)

    logger = logging.getLogger(__name__)
    logger.info("Wszystko ok dla testowania logRecord - Testowanie Loggera")
    global wlasne_dane
    wlasne_dane = "Some Text"
    logger.info("Wszystko ok dla testowania logRecord - Some Text")


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

Zadanie 3

Prawdopodobnie temperatura baterii telefonu może być dość wysoka. Sprawdź, czy to prawda. Napisz program, który będzie symulował rejestrację temperatury baterii w odstępie jednej minuty. Symulacja powinna zawierać 60 logów (z ostatniej godziny).Aby symulować temperatury, użyj jednej z dostępnych funkcji losowych w Pythonie. Temperatury należy narysować w zakresie 20–40 stopni Celsjusza, a następnie zapisać w następującym formacie:

```
LEVEL_NAME - TEMPERATURE_IN_CELSIUS UNIT => DEBUG - 20 C
```

Wylosowane temperatury należy przypisać do odpowiedniego poziomu w zależności od ich wartości:

```
DEBUG = TEMPERATURE_IN_CELSIUS < 20

WARNING = TEMPERATURE_IN_CELSIUS >= 30 AND TEMPERATURE_IN_CELSIUS <= 35

CRITICAL = TEMPERATURE_IN_CELSIUS > 35
```

Umieść wszystkie logi w pliku battery_temperature.log. Zadanie zostanie zakończone, gdy zaimplementujesz własny moduł obsługi i formater.

2

Programowanie zorientowane obiektowo

Object-Oriented Programming

Configparser – wstęp

- Obecnie wiele popularnych serwisów udostępnia API, które możemy wykorzystać w naszych aplikacjach. Integracja z tymi usługami wymaga uwierzytelnienia za pomocą danych takich jak login i hasło lub po prostu token dostępowy.
- Każda usługa może wymagać innych danych do uwierzytelnienia, ale jedno jest pewne – trzeba je gdzieś przechowywać w naszej aplikacji. Zakodowanie ich bezpośrednio w kodzie nie jest dobrym pomysłem.
- Lepszym rozwiązaniem jest użycie pliku konfiguracyjnego, który zostanie odczytany przez kod. W Pythonie jest to możliwe dzięki modułowi o nazwie configparser.
- Moduł configparser jest dostępny w standardowej bibliotece Pythona. Aby zacząć z niego korzystać musimy zaimportować odpowiedni moduł:

```
import configparser
```

- Struktura pliku konfiguracyjnego jest bardzo podobna do plików INI systemu Microsoft Windows.
- Składa się z sekcji identyfikowanych nazwami ujętymi w nawiasy kwadratowe.
- Sekcje zawierają elementy składające się z par klucz-wartość.
- Każda para jest oddzielona dwukropkiem : lub znakiem równości =.
- Co więcej, plik konfiguracyjny może zawierać komentarze poprzedzone średnikiem ; lub hash #.

Configparser – wstęp

```
[DEFAULT]
host = localhost # This is a comment.

[mariadb]
name = hello
user = user
password = password

[redis]
port = 6379
db = 0
```

- Plik konfiguracyjny zawiera sekcje DEFAULT, mariadb i redis.
- Sekcja DEFAULT jest nieco inna, ponieważ zawiera wartości domyślne, które można odczytać w innych sekcjach pliku. W naszym przypadku istnieje wspólny host dla wszystkich sekcji.
- Druga sekcja o nazwie mariadb przechowuje dane niezbędne do połączenia z bazą danych MariaDB. Są to nazwa bazy danych, nazwa użytkownika i hasło.
- Ostatnia sekcja zawiera dane konfiguracyjne Redis, składające się z portu i numeru bazy danych.
- Dodatkowo zarówno w tej sekcji jak i w sekcji mariadb mamy dostęp do opcji host zdefiniowanej w sekcji DEFAULT

Configparser – parsowanie pliku konfiguracyjnego

- Najpierw musimy utworzyć obiekt ConfigParser, który udostępnia wiele przydatnych metod analizowania danych.
- Jedną z nich jest metoda read, odpowiedzialna za odczyt i parsowanie pliku konfiguracyjnego.
- przekazujemy mu nazwę pliku config.ini, ale możliwe jest również przekazanie listy zawierającej kilka plików.
- Jeśli wszystko pójdzie dobrze, metoda read zwraca listę nazw plików, które zostały pomyślnie przeanalizowane.
- W przykładzie używamy metody sections, aby wyświetlić nazwy sekcji w pliku.
- Pamiętaj, że sekcja DEFAULT nie pojawia się na liście zwróconych sekcji.
- Dostęp do danych zawartych w pliku konfiguracyjnym jest analogiczny do sposobu, w jaki korzystamy ze słowników. Należy zauważyć, że w nazwach sekcji rozróżniana jest wielkość liter, a w kluczach nie.
- Pomimo tego, że sekcja DEFAULT jest pominięta w wyniku zastosowania metody sections, nadal mamy dostęp do jej opcji. Zarówno sekcje mariadb, jak i redis mogą odczytywać opcję hosta.
- Możliwy jest również dostęp do wartości przechowywanych w opcjach za pomocą metody get. Metoda get wymaga podania nazwy sekcji i klucza.

```
print('Host:', config.get('mariadb', 'host'))
```

```
import configparser

config = configparser.ConfigParser()
print(config.read('config.ini'))

print('Sections:', config.sections(),'\n')

print('mariadb section:')
print('Host:', config['mariadb']['host'])
print('Database:', config['mariadb']['name'])
print('Username:', config['mariadb']['user'])
print('Password:', config['mariadb']['password'], '\n')

print('redis section:')
print('Host:', config['redis']['host'])
print('Port:', int(config['redis']['port']))
print('Database number:', int(config['redis']['db']))
```

Configparser – czytanie configuracji z innych źródeł

- Moduł configparser umożliwia odczyt konfiguracji z różnych źródeł. Jednym z nich jest słownik, który możemy załadować za pomocą read_dict.
- Metoda read_dict akceptuje każdy słownik, którego kluczami są nazwy sekcji, natomiast wartości obejmują słowniki zawierające klucze i wartości. Wszystkie wartości odczytane ze słownika są konwertowane na łańcuchy znaków.
- Moduł configparser posiada również metody read_file i read_string, które umożliwiają odczytanie konfiguracji z otwartego pliku lub napisu.

```
import configparser

config = configparser.ConfigParser()

dict = {
    'DEFAULT': {
        'host': 'localhost'
    },
    'mariadb': {
        'name': 'hello',
        'user': 'root',
        'password': 'password'
    },
    'redis': {
        'port': 6379,
        'db': 0
    }
}

config.read_dict(dict)

print('Sections:', config.sections(),'\n')

print('mariadb section:')
print('Host:', config['mariadb']['host'])
print('Database:', config['mariadb']['name'])
print('Username:', config['mariadb']['user'])
print('Password:', config['mariadb']['password'], '\n')

print('redis section:')
print('Host:', config['redis']['host'])
print('Port:', int(config['redis']['port']))
print('Database number:', int(config['redis']['db']))
```

Configparser – tworzenie config

- Aby utworzyć plik konfiguracyjny, należy traktować obiekt ConfigParser jako słownik.
- Nazwa sekcji to klucze, a ich opcje są wymienione w oddzielnych słownikach.
- Konfiguracja zapisywana jest metodą write, która wymaga przekazania otwartego pliku w trybie tekstowym. W tym celu wykorzystywana jest wbudowana metoda open.
- Konfigurację załadowaną metodą read można również modyfikować. Aby zmienić pojedynczą opcję wystarczy ustawić nową wartość na odpowiedni klucz, a następnie zapisać plik metodą write

```
import configparser

config = configparser.ConfigParser()

config['DEFAULT'] = {'host': 'localhost'}
config['mariadb'] = {'name': 'hello',
                     'user': 'root',
                     'password': 'password'}
config['redis'] = {'port': 6379,
                   'db': 0}

with open('config2.ini', 'w') as configfile:
    config.write(configfile)
```

Configparser – interpolacja wartości

- Dużą zaletą pliku konfiguracyjnego jest możliwość zastosowania interpolacji.
- Pozwala na tworzenie wyrażeń składających się ze symbolu zastępczego, pod którym zostanie podstawiona odpowiednia wartość.
- Plik konfiguracyjny został rozszerzony o kolejną opcję o nazwie dsn. Jego wartość zawiera symbol zastępczy %(host)s, który należy zastąpić odpowiednią wartością.
- Umieszczenie dowolnego znaku między % a s informuje parser o konieczności interpolacji. Oczywiście cała praca jest wykonywana za nas, a my otrzymujemy tylko gotowe efekty.
- W przypadku opcji dsn będzie to następujący ciąg: redis://localhost. Zauważ, że symbol zastępczy %(host)s został zastąpiony wartością przechowywaną w opcji hosta.

```
[DEFAULT]
host = localhost
[mariadb]
name = hello
user = user
password = password
[redis]
port = 6379
db = 0
dsn = redis://%(host)s
```

```
import configparser

config = configparser.ConfigParser()
print(config.read('config3.ini'))

print('DSN:', str(config['redis']['dsn']))
```

Configparser – interpolacja

Basic Interpolation

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures
```

```
import configparser

config = configparser.ConfigParser()
print(config.read('basic_interpolation.ini'))

print('Pictures:',
str(config['Paths']['my_pictures']))
```

Extended Interpolation

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.9
path: ${Common:system_dir}/Library/Frameworks/

[Marcin]
nickname: MM
last_name: TT
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir:
${Frameworks:path}/Python/Versions/${Frameworks:
Python}
```

```
import configparser

config =
configparser.ConfigParser(interpolation=configparser
.ExtendedInterpolation())
print(config.read('extended_interpolation.ini'))

print('Python dir:', str(config['Marcin']['python_dir']))
```

New Interpolation

```
[section1]
home_dir: /WORK
my_dir: %(home_dir)s/TEMAT
key = value
my_path = $PATH
```

```
import configparser
import os


class EnvInterpolation(configparser.BasicInterpolation):
    """Interpolation which expands environment variables in values."""

    def before_get(self, parser, section, option, value, defaults):
        value = super().before_get(parser, section, option, value,
defaults)
        return os.path.expandvars(value)

config = configparser.ConfigParser(interpolation=EnvInterpolation())
print(config.read('new_interpolation.ini'))
print(config['section1']['my_dir'])
print(config['section1']['my_path'])
```

Zadanie 4

Wyobraź sobie sytuację, w której otrzymujesz plik konfiguracyjny zawierający dane dostępowe do różnych usług. Niestety plik to straszny bałagan, ponieważ zawiera dane wykorzystywane zarówno w środowisku produkcyjnym, jak i deweloperskim.

Twoim zadaniem będzie utworzenie dwóch plików o nazwach prod_config.ini i dev_config.ini przy użycia używając config parsera. Plik prod_config.ini powinien zawierać tylko sekcje dotyczące środowiska produkcyjnego, a plik dev_config.ini powinien zawierać tylko sekcje dotyczące środowiska programistycznego.

Aby rozróżnić środowiska, użyj opcji env dodanej do wszystkich sekcji w pliku mess.ini. Opcję env należy usunąć z sekcji przed przeniesieniem ich do plików.

```
[mariadb]
host = localhost
name = hello
user = user
password = password
env = dev

[sentry]
key = key
secret = secret
env = prod

[redis]
host = localhost
port = 6379
db = 0
env = dev

[github]
user = user
password = password
env = prod
```

3