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”)

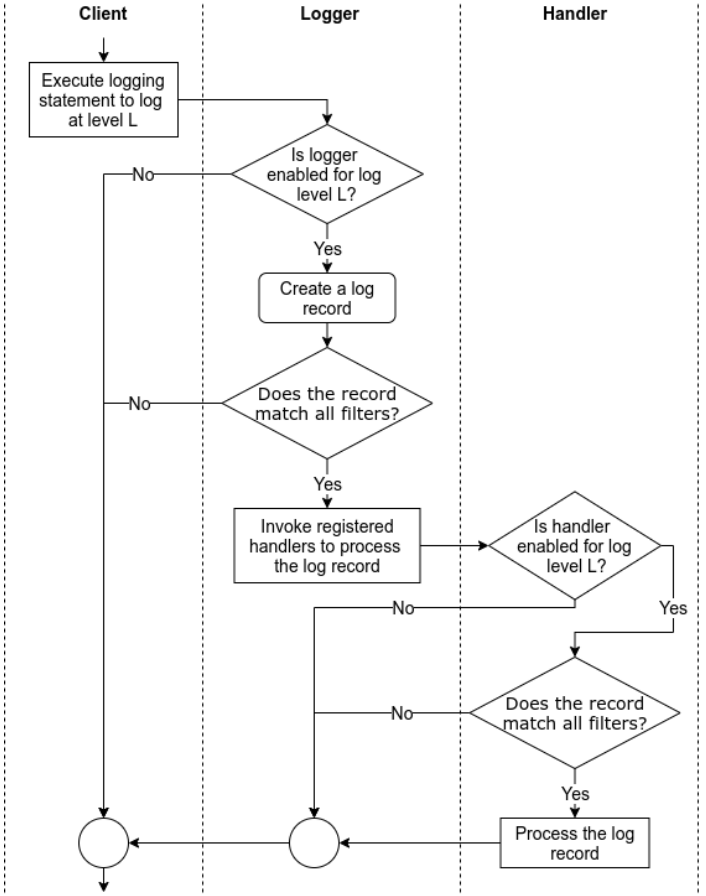
logging.getLogger(“coralogix.client”)

- 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.

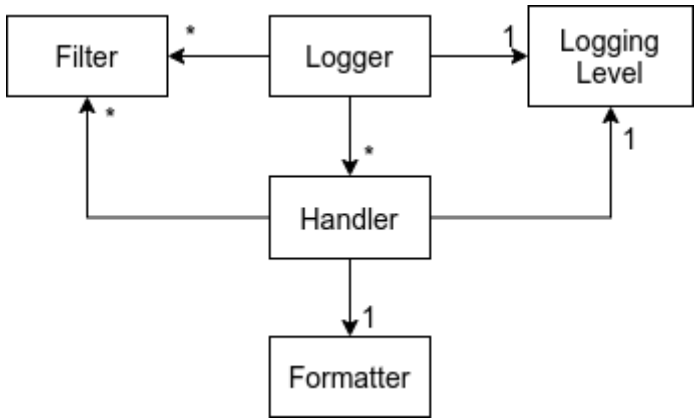
Logging in Python – bazowy concept

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}'

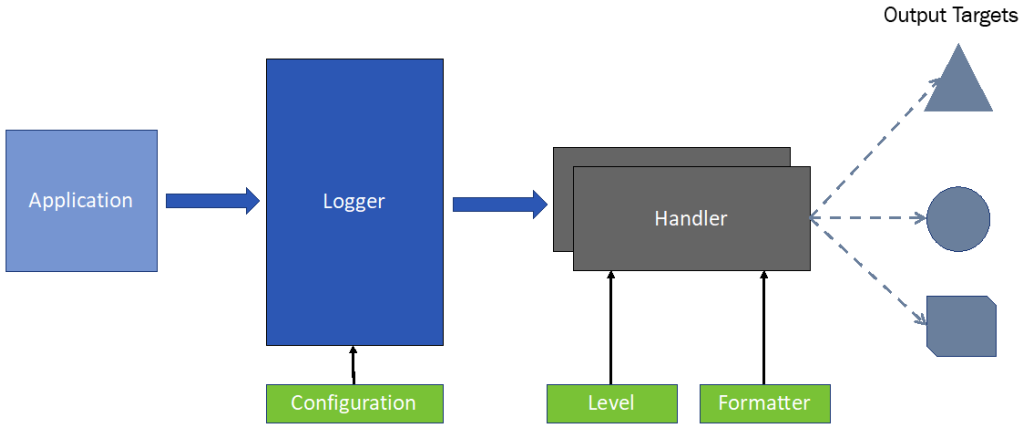
  str.template, e.g., '$levelname:$name:$message'

- 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

Formatowanie plików konfoguracyjnych

Configpraser

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

Funkcja lambda

Lambda function

Lambda function - wstęp

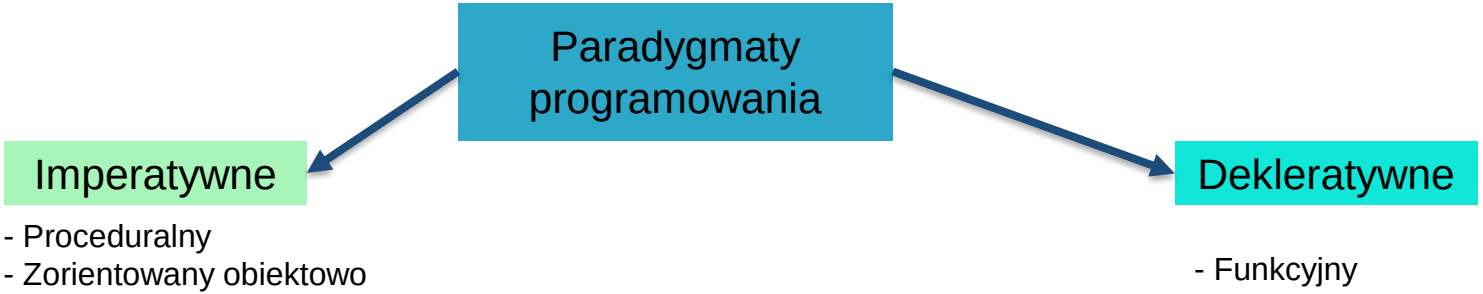
- Funkcje lambda są bardzo popularną koncepcją programistyczną, która jest szczególnie zakorzeniona w programowaniu funkcyjnym.
- W innych językach programowania funkcje lambda są czasami nazywane funkcjami anonimowymi, wyrażeniami lambda lub literałami funkcyjnymi.
- Python i inne języki, takie jak Java, C#, a nawet C++, mają dodane funkcje lambda do swojej składni, podczas gdy języki takie jak LISP lub rodzina języków ML, Haskell, OCaml i F#, używają lambda jako podstawowej koncepcji.
- Ze względu na składnię jest nieznacznie bardziej ograniczone niż zwykłe funkcje, ale wiele można dzięki niej zrobić.
- Wyrażenia lambda w Pythonie i innych językach programowania mają swoje korzenie w rachunku lambda, modelu obliczeniowym wymyślonym przez Alonzo Churcha
- Funkcje lambda w Pythonie można definiować tylko za pomocą wyrażeń. Składnia dlafunkcji lambda wygląda następująco:

```
lambda <arguments>: <expression>
```

Lambda function - historia

- Alonzo Church sformalizował rachunek lambda, język oparty na czystej abstrakcji, w latach trzydziestych XX wieku. Funkcje lambda są również nazywane abstrakcjami lambda, co stanowi bezpośrednie odniesienie do modelu abstrakcji oryginalnego dzieła Alonzo Churcha.
- Rachunek lambda może zakodować dowolne obliczenia. Jest kompletny Turinga, ale w przeciwieństwie do koncepcji maszyny Turinga jest czysty i nie zachowuje żadnego stanu.
- Języki funkcyjne wywodzą się z logiki matematycznej i rachunku lambda, podczas gdy imperatywne języki programowania obejmują oparty na stanie model obliczeń wymyślony przez Alana Turinga. Te dwa modele obliczeń, rachunek lambda i maszyny Turinga, można przełożyć na siebie. Ta równoważność jest znana jako hipoteza Churcha-Turinga.
- Języki funkcjonalne bezpośrednio dziedziczą filozofię rachunku lambda, przyjmując deklaratywne podejście do programowania, które kładzie nacisk na abstrakcję, transformację danych, kompozycję i czystość (brak stanu i brak efektów ubocznych). Przykładami języków funkcjonalnych są Haskell, Lisp lub Erlang.
- Z kolei maszyna Turinga doprowadziła do programowania imperatywnego w językach takich jak Fortran, C czy Python.
- Styl imperatywny polega na programowaniu za pomocą instrukcji, sterowaniu przebiegiem programu krok po kroku za pomocą szczegółowych instrukcji. Takie podejście promuje mutację i wymaga zarządzania stanem.
- Separacja w obu rodzinach zawiera pewne niuanse, ponieważ niektóre języki funkcjonalne zawierają funkcje imperatywne, takie jak OCaml, podczas gdy funkcje funkcjonalne przenikają rodzinę języków imperatywnych, w szczególności wraz z wprowadzeniem funkcji lambda w Javie lub Pythonie.
- Python nie jest z natury językiem funkcjonalnym, ale wcześnie przyjął pewne koncepcje funkcjonalne. W styczniu 1994 roku do języka dodano map(), filter(), reduce() i operator lambda.

Lambda function – paradygmaty programowania w pythonie



Omówimy to na przykładzie:
Dla każdego przykładu obliczone zostanie wartość kwadratowa listy wartości liczb pierwszych

Omówimy to na przykładzie:
Dla każdego przykładu obliczone zostanie wartość kwadratowa listy wartości liczb pierwszych

```
x = [1, 3, 5, 7, 11]
```

Lambda function – Paradygmaty - Podejście funkcyjne

```
x = [1, 3, 5, 7, 11]

def square(value):

    return value * value

print(list(map(square, x)))
```

Analiza:

- Square funkcja bierze pojedynczy argument i zwraca wartość mnożenia argumentu przez siebie
- Map funkcja zastosowuje iteracyjnie wejścia listy do funkcji
- Map funkcja jest tzw lazy funkcja i daje wynik jako obiekt iteratora dlatego potrzeba jest rzutowania iteratora na listę

Założenia:

- Definiuje każdy zadanie w zestaw funkcji
- W teorii wejściowe dane przechodzą przez zestaw funkcji
- Każda funkcja niezależnie od drugiej produkuje dane wyjściowe
- w przykładzie map funkcja iteracyjnie przetwarza każde wejście na wartość wyjściową
- Podobieństwo jest do matematycznych funkcji. Wywołując 2 razy z takimi samymi danymi wejściowymi otrzymujemy te sam rezultat dwa razy. Przykładem takiej funkcji jest funkcja print()

Lambda function – Paradygmaty - Podejście proceduralne

```
x = [1, 3, 5, 7, 11]
def square(prime_list):
    square_list = []
    for x in prime_list:
        square_list.append(x*x)
    return square_list
print(square(x))
```

Analiza:

- Square funkcja przyjmuje listę jako wejście
- Tworzona jest pusta lista
- Następnie lista ta wypełniana jest w pętli i wyliczane jest kwadrat wartości argumentu.

Założenia:

- Polega na wywołaniach proceduralnych
- Wspiera modularność kodu i separuje je w tzw małe bloki
- Wykonuje kod krok po kroku
- Wszystkie pythonowe skrypty wywoływane są proceduralnie. Każdy blok kodu wywoływany jest po następnym

Lambda function – Paradygmaty - Podejście obiektowe

```
x = [1, 3, 5, 7, 11]
class ListOperation():

    def __init__(self, any_list):
        self.any_list = any_list
        self.square()

    def square(self):
        self.sq = [i*i for i in self.any_list]

create_square = ListOperation(x)
print(create_square.sq)
```

Założenia:

- Podejście obiektowe łatwo ponownie używać
- Łatwiej zrozumieć
- Łatwo rozszerzać kod

Analiza:

- ListOperation to prototyp clasy
- __init__ konstruktor klasy gdzie przyjmowany jest parametr any_list i wywoływana jest funkcja square
- Square jest to metoda classy gdzie obliczana jest wartość kwadratu poszczególnych lementów
- Create_square jest to isntancja klasy ListOperation

Lambda function – podstawowy przykład

- Funkcja tożsamości, funkcja, która zwraca swój argument, jest wyrażona za pomocą standardowej definicji funkcji Pythona przy użyciu słowa kluczowego def
- Identity() przyjmuje argument x i zwraca go po wywołaniu
- W przeciwieństwie do tego, jeśli użyjesz konstrukcji lambda Pythona, wyrażenie składa się z

  Słowa kluczowego: lambda

  Zmiennej powiązanej: x

  Ciała: x
- W kontraście do zmienne powiązanej jest zmienna wolna i można się do niej odwołać w treści wyrażenia. Wolna zmienna może być stałą lub zmienną zdefiniowaną w otaczającym zakresie funkcji.
- Można zastosować powyższą funkcję do argumentu, umieszczając funkcję i jej argument w nawiasach
- Redukcja to strategia rachunku lambda służąca do obliczania wartości wyrażenia. W bieżącym przykładzie polega to na zastąpieniu zmiennej powiązanej x argumentem 2
- Wyrażenie może być również nazwane przez add_one

```
# Normal function
def identity(x):
    return x

print(identity(2))

# Lambda function
x=2
I = lambda x: x
print(I(2))
```

```
# Lambda funkcja z dodaniem +1 do argumentu

print((lambda x: x + 1)(2))
add_one = lambda x: x + 1
print(add_one(2))
```

Lambda function – podstawowy przykład

- Wszystkie te funkcje przyjmują jeden argument.
- W definicji wyrażeń lambda argumenty nie są otoczone nawiasami.
- Funkcje są wyrażane w wyrażeniach lambda Pythona, wymieniając argumenty i oddzielając je przecinkiem, ale bez otaczania ich nawiasami.
- Funkcja lambda przypisana do full_name przyjmuje dwa argumenty i zwraca ciąg interpolujący dwa parametry pierwszy i ostatni.
- Zgodnie z oczekiwaniami definicja wyrażenia lambda wymienia argumenty bez nawiasów, podczas gdy wywołanie funkcji odbywa się dokładnie tak, jak normalna funkcja Pythona, z nawiasami otaczającymi argumenty.

```
full_name = lambda first, last: f'Full name: {first.title()} {last.title()}'

print(full_name('Jan', 'Kowal'))
```

Lambda function – podstawowy przykład

- Innym wzorcem używanym w innych językach, takich jak JavaScript, jest natychmiastowe wykonanie funkcji lambda. Jest to znane jako natychmiast wywołane wyrażenie funkcyjne (IIFE)
- Powyższa funkcja lambda jest zdefiniowana, a następnie natychmiast wywoływana z dwoma argumentami (2 i 3). Zwraca wartość 5, która jest sumą argumentów.
- Python nie zachęca do używania bezpośrednio wywoływanych wyrażeń lambda. Wynika to po prostu z tego, że wyrażenie lambda jest wywoływalne, w przeciwieństwie do ciała normalnej funkcji.
- Funkcje lambda są często używane z funkcjami wyższego rzędu, które przyjmują jedną lub więcej funkcji jako argumenty lub zwracają jedną lub więcej funkcji.
- Funkcja lambda może być funkcją wyższego rzędu, przyjmując funkcję (normalną lub lambda) jako argument

```
full_name = lambda first, last: f'Full name: {first.title()} {last.title()}'

print(full_name('Jan', 'Kowal'))
```

```
ho_func = lambda x, func: x + func(x)

print(ho_func(2, lambda x: x * x))
print(ho_func(2, lambda x: x + 3))
```

Lambda function – różnice miedzy zwykłą funkcją

- Sprawdźmy, jak Python widzi funkcję zbudowaną za pomocą pojedynczej instrukcji return w porównaniu z funkcją zbudowaną jako wyrażenie (lambda).
- Można zobaczyć, że dis() udostępnia czytelną wersję kodu bajtowego Pythona, umożliwiając inspekcję instrukcji niskiego poziomu, których interpreter Pythona użyje podczas wykonywania programu.
- Kod bajtowy interpretowany przez Pythona jest taki sam dla obu funkcji. Ale można zauważyć, że nazewnictwo jest inne: nazwa funkcji to add dla funkcji zdefiniowanej za pomocą def, podczas gdy funkcja lambda w Pythonie jest postrzegana jako lambda.

```
import dis

add = lambda x, y: x + y

print(type(add))
dis.dis(add)
print(add)


def addf(x, y): return x + y

print(type(addf))
dis.dis(addf)
print(addf)
```

```
<class 'function'>
  3           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
<function <lambda> at 0x0000013CA135F040>
<class 'function'>
 10           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
<function addf at 0x0000013CA1C3CDC0>
```

Lambda function – cechy

Jak widzieliśmy w poprzednich sekcjach, forma lambda przedstawia różnice składniowe od normalnej funkcji. W szczególności funkcja lambda ma następujące cechy:

- Może zawierać tylko wyrażenia i nie może zawierać instrukcji w swojej treści. Funkcja lambda nie może zawierać żadnych instrukcji. W funkcji lambda instrukcje takie jak return, pass, assert lub raise spowodują zgłoszenie wyjątku SyntaxError.
- Jest napisany jako pojedyncza linia wykonania. W przeciwieństwie do normalnej funkcji, funkcja lambda w Pythonie jest pojedynczym wyrażeniem. Chociaż w treści wyrażenia lambda można rozłożyć wyrażenie na kilka wierszy za pomocą nawiasów lub łańcucha wielowierszowego, pozostaje ono pojedynczym wyrażeniem
- Nie obsługuje adnotacji typu.
- Można go natychmiast wywołać (IIFE - immediately invoked function execution). - natychmiastowe wykonanie funkcji

Lambda function – agrumenty

Podobnie jak normalny obiekt funkcji zdefiniowany za pomocą def, wyrażenia lambda w Pythonie obsługują wszystkie różne sposoby przekazywania argumentów. To zawiera:

- Argumenty pozycyjne
- Nazwane argumenty (czasami nazywane argumentami słów kluczowych)
- Zmienna lista argumentów (często określana jako varargs)
- Zmienna lista argumentów słów kluczowych
- Argumenty oparte tylko na słowach kluczowych

```
print((lambda x, y, z: x + y + z)(1, 2, 3))

print((lambda x, y, z=3: x + y + z)(1, 2))

print((lambda x, y, z=3: x + y + z)(1, y=2))

print((lambda *args: sum(args))(1,2,3))

print((lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3))

print((lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3))
```

Lambda function – dekorator

- Dekorator można zastosować do lambdy. Chociaż nie jest możliwe udekorowanie lambdy za pomocą składni @decorator, dekorator jest tylko funkcją, więc może wywoływać funkcję lambda,
- Nazwa funkcji lambda pojawia się jako <lambda>, podczas gdy add_two jest wyraźnie identyfikowane dla normalnej funkcji.
- Dekorowanie funkcji lambda w ten sposób może być przydatne do celów debugowania, być może do debugowania zachowania funkcji lambda używanej w kontekście funkcji wyższego rzędu lub funkcji kluczowej. Zobaczmy przykład z map()

```
list(map(trace(lambda x: x*2), range(3)))
```

[TRACE] func: <lambda>, args: (0,), kwargs: {}
[TRACE] func: <lambda>, args: (1,), kwargs: {}
[TRACE] func: <lambda>, args: (2,), kwargs: {}

```
def trace(f):
    def wrap(*args, **kwargs):
        print(f"[TRACE] func: {f.__name__}, args: {args}, kwargs:
{kwargs}")
        return f(*args, **kwargs)

    return wrap


@trace
def add_two(x):
    return x + 2

add_two(3)

print((trace(lambda x: x ** 2))(3))
```

[TRACE] func: add_two, args: (3,), kwargs: {}
[TRACE] func: <lambda>, args: (3,), kwargs: {}

Lambda function – Zamknięcia

Zamknięcie (Closure) to funkcja, w której każda wolna zmienna, wszystko poza parametrami, użyte w tej funkcji, jest powiązana z określoną wartością zdefiniowaną w otaczającym zakresie tej funkcji. W efekcie domknięcia definiują środowisko, w którym działają, więc można je wywoływać z dowolnego miejsca.

Pojęcia lambda i domknięć niekoniecznie są ze sobą powiązane, chociaż funkcje lambda mogą być domknięciami w taki sam sposób, jak normalne funkcje mogą być domknięciami.

External_func() zwraca inner_func(), zagnieżdżoną funkcję, która oblicza sumę trzech argumentów.

- x jest przekazywany jako argument do funkcji external_func().
- y jest zmienną lokalną dla funkcji external_func().
- z jest argumentem przekazanym do inner_func().
- Aby przetestować zachowanie funkcji external_func() i inner_func(), wywoływana jest funkcja zewnętrzna_func() w pętli for
- W linii 9 kodu, inner_func() zwracana przez wywołanie external_func() jest powiązana z zamknięciem nazwy.
- W linii 5. inner_func() przechwytuje x i y, ponieważ ma dostęp do swojego środowiska osadzania, tak że po wywołaniu domknięcia może operować na dwóch wolnych zmiennych x i y.

Podobnie lambda może być również domknięciem. Oto ten sam przykład z funkcją lambda

- W linii 6, zewnętrzna_funkcja() zwraca lambda i przypisuje ją do zamknięcia zmiennej.
- W linii 3 ciało funkcji lambda odwołuje się do x i y.
- Zmienna y jest dostępna w czasie definiowania, podczas gdy x jest definiowana w czasie wykonywania, gdy wywoływana jest funkcja zewnętrzna_func().

```
def outer_func(x):
    y = 4
    def inner_func(z):
        print(f"x = {x}, y = {y}, z = {z}")
        return x + y + z
    return inner_func

for i in range(3):
    closure = outer_func(i)
    print(f"closure({i+5}) = {closure(i+5)}")
```

```
x = 0, y = 4, z = 5
closure(5) = 9
x = 1, y = 4, z = 6
closure(6) = 11
x = 2, y = 4, z = 7
closure(7) = 13
```

```
def outer_func(x):
    y = 4
    return lambda z: x + y + z

for i in range(3):
    closure = outer_func(i)
    print(f"closure({i+5}) = {closure(i+5)}")
```

```
closure(5) = 9
closure(6) = 11
closure(7) = 13
```

Lambda function – jako funkcja

```
# funkcja
f = lambda x, y, z: x + y + z
print(f(2, 3, 4))
# Lista lambda
L = [(lambda x: x**2),
    (lambda x: x**3),
    (lambda x: x**4)]
for f in L:
    print(f(2))

# funkcja - porównywanie
lower = (lambda x, y: x if x < y else y)
print(lower('cc', 'bb'))
print(lower(8, 5))

# zagnieżdzenie
action = (lambda x: (lambda y: x + y))
act = action(99)
print(act(2))
```

```
# mnożnik
def multip(n):
  return lambda a : a * n

mydoubler = multip(2)
mytripler = multip(3)

print(mydoubler(11))
print(mytripler(11))

# sześcian
cube = lambda x: x*x*x
print(cube(7))
```

Lambda function – map, filter, reduce

```
# Map
print(list(map(lambda *a: a, range(3))))
print(list(map(lambda *a: a, range(3), 'abc')))
print(list(map(lambda *a: a, range(3), 'abc', range(4, 7))))
print(list(map(lambda *a: a, (1, 2, 3, 4), 'abc')))
print(list(map(lambda x: x**2, range(10))))

# Filter
print(list(filter(lambda x: (x%2 != 0) , [5, 7, 22, 97, 54, 62, 77, 23, 73,
61])))

evens = filter(lambda number: number % 2 == 0, range(10))
odds = filter(lambda number: number % 2 == 1, range(10))
print(f"Even numbers in range from 0 to 9 are: {list(evens)}")
print(f"Odd numbers in range from 0 to 9 are: {list(odds)}")

animals = ["giraffe", "snake", "lion", "squirrel"]
animals_s = filter(lambda animal: animal.startswith('s'), animals)
print(f"Animals that start with letter 's' are: {list(animals_s)}")
```

```
# Reduce
from functools import reduce

print(reduce(lambda a, b: a + b, [2, 2]))
print(reduce(lambda a, b: a + b, [2, 2, 2]))
print(reduce(lambda a, b: a + b, range(100)))
print(reduce((lambda x, y: x + y), [5, 8, 10, 20, 50, 100]))

# Dropwhile, takewhile
from itertools import dropwhile,takewhile
print(list(dropwhile(lambda x: x <= 3, [1, 3, 5, 4, 2])))
print(list(takewhile(lambda x: x <= 3, [1, 3, 5, 4, 2])))
```

Lambda function – przykłady

```
def dispatch_dict(operator, x, y):
    return {
        'add': lambda: x + y,
        'sub': lambda: x - y,
        'mul': lambda: x * y,
        'div': lambda: x / y,
    }.get(operator, lambda: None)()

print(dispatch_dict('mul', 2, 8))
print(dispatch_dict('unknown', 2, 8))


t = [(1, 'd'), (2, 'b'), (4, 'a'), (3, 'c')]
print(sorted(t, key=lambda x: x[0]))
print(sorted(t, key=lambda x: x[1]))

print(sorted(range(-5, 6), key=lambda x: x * x))
```

```
import heapq
nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums))
print(heapq.nsmallest(3, nums))


portfolio = [
{'nazwa': 'IBM', 'udzialy': 100, 'cena': 91.1},
{'nazwa': 'AAPL', 'udzialy': 50, 'cena': 543.22},
{'nazwa': 'FB', 'udzialy': 200, 'cena': 21.09},
{'nazwa': 'HPQ', 'udzialy': 35, 'cena': 31.75},
{'nazwa': 'YHOO', 'udzialy': 45, 'cena': 16.35},
{'nazwa': 'ACME', 'udzialy': 75, 'cena': 115.65}
]

tanie = heapq.nsmallest(3, portfolio, key=lambda s: s['cena'])
drogie = heapq.nlargest(3, portfolio, key=lambda s: s['cena'])

print(tanie)
print(drogie)
```

Zadanie 4

- Posortuj malejąco listę słowników studentów przy użyciu funkcji lambda względem końcowej oceny za studia według klucza:

ocena za studia = 0,2 * ocena za egzamina + 0,2 * ocena za obrona + 0,6 * średnia ocen z pozostałych przedmiotów

```
students = [
{'imie': 'Artur', 'nazwisko': 'Kowalczyk', 'matematyk': 5.0, 'fizyka': 4.5, 'informatyka': 4.0, 'elektryka': 3.0, 'egzamin koncowy': 4.0, 'obrona': 3.0},
{'imie': 'Joanna', 'nazwisko': 'Sikora', 'matematyk': 3.0, 'fizyka': 3.5, 'informatyka': 4.0, 'elektryka': 4.0, 'egzamin koncowy': 3.0, 'obrona': 5.0},
{'imie': 'Janusz', 'nazwisko': 'Adamczyk', 'matematyk': 3.5, 'fizyka': 4.0, 'informatyka': 3.5, 'elektryka': 3.0, 'egzamin koncowy': 4.0, 'obrona': 4.0},
{'imie': 'Ewa', 'nazwisko': 'Siwinska', 'matematyk': 4.5, 'fizyka': 3.5, 'informatyka': 4.5, 'elektryka': 4.5, 'egzamin koncowy': 5.0, 'obrona': 5.0},
{'imie': 'Radek', 'nazwisko': 'Smalec', 'matematyk': 4.0, 'fizyka': 4.5, 'informatyka': 3.0, 'elektryka': 5.0, 'egzamin koncowy': 4.0, 'obrona': 4.0},
{'imie': 'Wiktoria', 'nazwisko': 'Kaszka', 'matematyk': 5.0, 'fizyka': 5.0, 'informatyka': 4.5, 'elektryka': 4.0, 'egzamin koncowy': 3.0, 'obrona': 3.0},
]
```

4

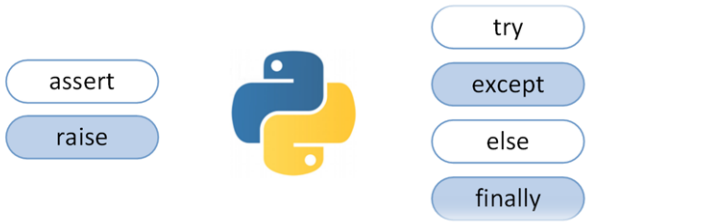
Zarządzanie wyjątkami

Exceptions Handling

Exceptions – Wstęp

Kiedy Python wykonuje skrypt i napotyka sytuację, z którą nie może sobie poradzić, to:

- zatrzymuje program;
- tworzy specjalny rodzaj danych, zwany wyjątkiem.
- Tym wyjątkiem jest obiekt.



W Pythonie błąd może być błędem składniowym lub wyjątkiem

Czynności te są nazywane zgłaszaniem wyjątku. Można powiedzieć, że Python zawsze zgłasza wyjątek (lub że wyjątek został zgłoszony), gdy program nie może wykonać kodu.

- zgłoszony wyjątek oczekuje, że zostanie obsłużony;
- jeśli nic się nie stanie, aby zająć się zgłoszonym wyjątkiem, program zostanie przymusowo zakończony i zobaczysz komunikat o błędzie wysłany do konsoli przez Pythona;
- w przeciwnym razie, jeśli wyjątek zostanie rozwiązany i odpowiednio obsłużony, zawieszony program może zostać wznowiony, a jego wykonywanie może być kontynuowane.
- Python ma wbudowane 63* wyjątki, które można przedstawić w postaci hierarchii w kształcie drzewa. Powodem tego jest to, że wyjątki są dziedziczone z BaseException, najbardziej ogólnej klasy wyjątków.
- A to podejście mówi ci, że możesz także tworzyć własne specyficzne klasy wyjątków – jedynym ograniczeniem jest to, że musisz utworzyć podklasę BaseException lub inną pochodną klasę wyjątków.

Exceptions – Wstęp

Wyjątki i ich hierarchię można znaleźć pod https://docs.python.org/3/library/exceptions.html

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

Exceptions – Exceptions a Syntax Errors

- Błędy składniowe (Syntax errors) występują, gdy parser wykryje niepoprawną instrukcję

```
>>> print( 0 / 0 ))
  Input In [2]
    print( 0 / 0 ))
                  ^
SyntaxError: unmatched ')'
```

- Strzałka wskazuje miejsce, w którym analizator składni napotkał błąd składniowy. W tym przykładzie było o jeden nawias za dużo. Po usunięciu nawiasu

```
>>> print( 0 / 0 )
Traceback (most recent call last):
  File "C:\Users\caryk\AppData\Local\Programs\Python\Python39\lib\site-packages\IPython\core\interactiveshell.py", line 3369, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-3-f5008b880c58>", line 1, in <cell line: 1>
    print( 0 / 0 )
ZeroDivisionError: division by zero
```

- Teraz występuje błąd wyjątku (Exception error). Ten typ błędu występuje, gdy poprawny składniowo kod Pythona powoduje błąd. Ostatni wiersz komunikatu wskazuje typ napotkanego błędu wyjątku.

Exceptions – podnoszenie wyjątku i wyjątek AssertionError

- Możemy użyć funkcji raise, aby zgłosić wyjątek, jeśli wystąpi warunek. Instrukcję można uzupełnić o niestandardowy wyjątek.
- Jeśli chcesz zgłosić błąd, gdy wystąpi określony warunek przy użyciu risee, możesz to zrobić w następujący sposób:

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was:
{}'.format(x))
```

- Program zatrzymuje się i wyświetla nasz wyjątek na ekranie, oferując wskazówki dotyczące tego, co poszło nie tak.
- W Python jest możliwe wykonania asercji. Stwierdzamy, że spełniony jest pewien warunek. Jeśli ten warunek okaże się prawdziwy, to wszystko ok! Program może być kontynuowany. Jeśli warunek okaże się fałszywy, program może zgłosić wyjątek AssertionError.
- Spójrz na poniższy przykład, w którym stwierdzono, że kod zostanie wykonany w systemie Linux

```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

- Jeśli uruchomisz ten kod na komputerze z systemem Linux, asercja przejdzie. Gdybyś uruchomił ten kod na komputerze z systemem Windows, wynikiem potwierdzenia byłoby Fałsz, a wynikiem będzie błąd asercji

Exceptions – Podstawowa struktura

- Jeśli podejrzewamy, że kod może zgłosić wyjątek, powinienniśmy użyć try: problematic_code without code block, aby otoczyć „problematyczny” fragment kodu.
- W efekcie, gdy zostanie zgłoszony wyjątek, wykonanie kodu nie zostanie zakończone, ale kod następujący po klauzuli wyjątku spróbuje poradzić sobie z problemem w elegancki sposób.

```
try:
    print(int('a'))
except ValueError:
    print('You tried to do a nasty thing...')
```

- Za każdym razem, gdy spróbujesz przekonwertować literę „a” na wartość całkowitą, zauważysz wyjątek.
- W przypadku, gdy otrzymujesz dane z zewnętrznego źródła (konsoli, pliku itp.) nie powinieneś ufać typom danych, więc mądrze jest otoczyć delikatny kod (int() w tym przykładzie) try... wyjątkiem blok.

Exceptions – Podstawowa struktura

- Blok try i except w Pythonie służy do przechwytywania i obsługi wyjątków.
- Python wykonuje kod następujący po instrukcji try jako „normalną” część programu. Kod następujący po instrukcji wyjątku jest odpowiedzią programu na wszelkie wyjątki w poprzedzającej klauzuli try.
- Jak widzieliśmy wcześniej, gdy poprawny składniowo kod napotka błąd, Python zgłosi błąd wyjątku. Ten błąd wyjątku spowoduje awarię programu, jeśli nie zostanie obsłużony.
- Except określa sposób reagowania programu na wyjątki.

```
try:
    # Wykonywany kod
except:
    # wykonywany blok jeżeli wystąpi wyjątek.
```

Exceptions – Podstawowa struktura

- Linux_interaction() może działać tylko w systemie Linux.
- Asert w tej funkcji zgłosi wyjątek AssertionError, jeśli wywołasz go w systemie operacyjnym innym niż Linux.
- W przypadku wystąpienia wyjątku w programie uruchamiającym tę funkcję, program będzie kontynuował działanie oraz poinformuje o tym, że wywołanie funkcji nie powiodło się.
- To, czego nie udało ci się zobaczyć, to rodzaj błędu, który został zgłoszony w wyniku wywołania funkcji. Aby dokładnie zobaczyć, co poszło nie tak, musisz przechwycić błąd, który zgłosiła funkcja.
- Pierwszy komunikat to AssertionError, informujący, że funkcja może być wykonana tylko na komputerze z systemem Linux. Drugi komunikat informuje, która funkcja nie została wykonana.

```
import sys
def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux
systems."
    print('Doing something.')

try:
    linux_interaction()
except:
    print('Linux function was not executed')
```

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
    print('The linux_interaction() function was not executed')
```

Exceptions – atrybuty

- Zagłębimy się w interesujące cechy koncepcji wyjątków i zapoznamy się z możliwymi sposobami wykorzystania tych koncepcji w swoich aplikacjach.
- Except może określać zmienną po nazwie wyjątku. W tym przykładzie jest to e_variable. Ta zmienna jest powiązana z instancją wyjątku z argumentami przechowywanymi w atrybucie args obiektu e_variable.
- Niektóre obiekty wyjątków zawierają dodatkowe informacje o samym wyjątku.
- Wyjątek ImportError – wywoływany, gdy instrukcja import ma problem z próbą załadowania modułu.
- Atrybuty to:

  nazwa – reprezentuje nazwę modułu, który próbowano zaimportować;

  path – reprezentuje odpowiednio ścieżkę do dowolnego pliku, który wyzwolił wyjątek. Może być None.

```
try:
    print(int('a'))
except ValueError as e_variable:
    print(e_variable.args)
```

```
try:
    import abcdefghijk

except ImportError as e:
    print(e.args)
    print(e.name)
    print(e.path)
```

Exceptions – atrybuty

- Wyjątek UnicodeError – zgłaszany, gdy wystąpi błąd kodowania lub dekodowania związany z Unicode. Jest to podklasa ValueError.
- UnicodeError ma atrybuty opisujące błąd kodowania lub dekodowania.
- encoding – nazwa kodowania, które wywołało błąd.
- reason – ciąg opisujący konkretny błąd kodeka.
- object – obiekt, który kodek próbował zakodować lub zdekodować.
- start – pierwszy indeks nieprawidłowych danych w obiekcie.
- end – indeks po ostatnich nieprawidłowych danych w obiekcie.

```
try:
    b'\x80'.decode("utf-8")
except UnicodeError as e:
    print(e)
    print(e.encoding)
    print(e.reason)
    print(e.object)
    print(e.start)
    print(e.end)
```

Exceptions – try except

- Możesz mieć więcej niż jedno wywołanie funkcji w swojej klauzuli try i przewidywać przechwytywanie różnych wyjątków.
- Należy tutaj zauważyć, że kod w klauzuli try zostanie zatrzymany, gdy tylko napotkany zostanie wyjątek.
- Łapanie wyjątków ukrywa wszystkie błędy… nawet te, które są zupełnie nieoczekiwane.
- Dlatego powinno się unikać pustych klauzul w programach w Pythonie.
- Zamiast tego lepiej odnieść się do konkretnych klas wyjątków, które chcemy przechwycić i obsłużyć.
- W przedstawionym przykładzie najpierw wywoływana jest funkcję linux_interaction(), a następnie próbowany jest otwarcie pliku

```
import sys
def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux
systems."
    print('Doing something.')

try:
    linux_interaction()
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
except AssertionError as error:
    print(error)
    print('Linux linux_interaction() function was not executed')
```

Wnioski

- Klauzula try jest wykonywana aż do momentu napotkania pierwszego wyjątku.
- Wewnątrz klauzuli except lub procedury obsługi wyjątku określasz, w jaki sposób program zareaguje na wyjątek.
- Możesz przewidzieć wiele wyjątków i rozróżnić sposób, w jaki program powinien na nie reagować.
- Unikaj używania pustych klauzul except.

Exceptions – try except

- Możesz mieć więcej niż jedno wywołanie funkcji w swojej klauzuli try i przewidywać przechwytywanie różnych wyjątków.
- Należy tutaj zauważyć, że kod w klauzuli try zostanie zatrzymany, gdy tylko napotkany zostanie wyjątek.
- Łapanie wyjątków ukrywa wszystkie błędy… nawet te, które są zupełnie nieoczekiwane.
- Dlatego powinno się unikać pustych klauzul w programach w Pythonie.
- Zamiast tego lepiej odnieść się do konkretnych klas wyjątków, które chcemy przechwycić i obsłużyć.
- W przedstawionym przykładzie najpierw wywoływana jest funkcję linux_interaction(), a następnie próbowany jest otwarcie pliku

```
import sys
def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux
systems."
    print('Doing something.')

try:
    linux_interaction()
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
except AssertionError as error:
    print(error)
    print('Linux linux_interaction() function was not executed')
```

Wnioski

- Klauzula try jest wykonywana aż do momentu napotkania pierwszego wyjątku.
- Wewnątrz klauzuli except lub procedury obsługi wyjątku określasz, w jaki sposób program zareaguje na wyjątek.
- Możesz przewidzieć wiele wyjątków i rozróżnić sposób, w jaki program powinien na nie reagować.
- Unikaj używania pustych klauzul except.

Exceptions – wyjątki łańcuchowe

- Python 3 wprowadził bardzo interesującą cechę o nazwie „Łańcuch wyjątków”, aby skutecznie radzić sobie z wyjątkami.
- Wyobraź sobie sytuację, w której już obsługujesz wyjątek, a Twój kod przypadkowo uruchamia dodatkowy wyjątek.
- Informacje powinny być dostępne dla kodu następującego po błędnym kodzie. To jest przykład niejawnego łączenia łańcuchów.
- Inny przypadek pojawia się, gdy świadomie chcemy obsłużyć wyjątek i przetłumaczyć go na inny typ wyjątku.
- Taka sytuacja jest typowa, gdy istnieje dobry powód, aby ujednolicone zachowanie jednego fragmentu kodu zachowywało się podobnie do innego fragmentu kodu, na przykład starszego kodu.
- W tej sytuacji dobrze byłoby również zachować szczegóły poprzedniego wyjątku. To jest przykład jawnego tworzenia łańcuchów.
- Ta koncepcja tworzenia łańcuchów wprowadza dwa atrybuty instancji wyjątków:

  atrybut __context__, który jest nieodłączny dla niejawnie połączonych wyjątków;

  atrybut __cause__, który jest nieodłączny dla jawnie połączonych wyjątków.
- Te atrybuty pomagają programiście zachować odniesienie do oryginalnego obiektu wyjątku w wygodny i spójny sposób do późniejszego przetwarzania, takiego jak rejestrowanie itp.

Exceptions – wyjątki łańcuchowe niejawne

- Przyjrzyj się poniższemu kodowi i wynikowi śledzenia wstecznego.
- Zwróć uwagę na fakt, że nie zgłaszamy żadnego wyjątku jawnie za pomocą instrukcji raise, ale powodujemy to niejawnie
- Wynik wykonania kodu zawiera komunikat, który łączy kolejne tracebacki:

```
During handling of the above exception, another exception occurred:
```

- Zawiera interesującą informację wskazującą, że właśnie byliśmy świadkami łańcucha wyjątków.

```
a_list = ['First error', 'Second error']

try:
    print(a_list[3])
except Exception as e:
    print(0 / 0)
```

- Do pierwotnego obiektu wyjątku e odwołuje się teraz atrybut __context__ następującego wyjątku f
- Klauzula z wyjątkiem Exception jest szeroka i zwykle powinna być używana w ostateczności, aby przechwycić wszystkie nieobsłużone wyjątki.
- Jest tak szeroki, ponieważ nie wiemy, jaki rodzaj wyjątku może wystąpić.
- Tak więc, gdy wystąpi kolejny wyjątek (znacznie lepiej prognozowany), nadal możemy wiele powiedzieć o naturze pierwszego wyjątku.

```
a_list = ['First error', 'Second error']

try:
    print(a_list[3])
except Exception as e:
    try:
        print(1 / 0)
    except ZeroDivisionError as f:
        print('Inner exception (f):', f)
        print('Outer exception (e):', e)
        print('Outer exception referenced:', f.__context__)
        print('Is it the same object:', f.__context__ is e)
```

Exceptions – wyjątki łańcuchowe jawne

- Tym razem chcemy przekonwertować jawny typ obiektu wyjątku na inny typ obiektu wyjątku w momencie wystąpienia drugiego wyjątku.
- Wyobraź sobie, że Twój kod odpowiada za końcowy proces sprawdzania przed wystrzeleniem rakiety. Lista kontroli jest długa, a różne kontrole mogą skutkować różnymi wyjątkami.
- Teraz widać, że wygodniej byłoby przekonwertować każdy typ wyjątku na jego własny wyjątek (jak RocketNotReadyError) i zarejestrować pochodzenie wyjątku.
- Po raz kolejny wynik wykonania kodu zawiera interesującą informację wskazującą, że właśnie byliśmy świadkami łańcucha wyjątków:

```
The above exception was the direct cause of the
following exception:
```

- Aby przechwycić przyczynę wyjątku RocketNotReadyError, należy uzyskać dostęp do atrybutu __Cause__ obiektu RocketNotReadyError

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e

crew = ['John', 'Mary', 'Mike']
print('Final check procedure')

personnel_check()
```

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e

crew = ['John', 'Mary', 'Mike']
print('Final check procedure')

try:
    personnel_check()
except RocketNotReadyError as f:
    print('General exception: "{}", caused by "{}"'.format(f, f.__cause__))
```

Exceptions – wyjątki łańcuchowe jawne

- Należy zwrócić uwagę na fakt, że dzięki polimorfizmowi i jawnemu łączeniu łańcuchów nasze podejście stało się bardziej ogólne.
- jesteśmy w stanie przeprowadzić dwie różne kontrole, z których każda zwraca inny typ wyjątku.
- I nadal jesteśmy w stanie obsłużyć je poprawnie, ponieważ ukrywamy niektóre szczegóły za obiektem wyjątku RocketNotReadyError.

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e


def fuel_check():
    try:
        print('Fuel tank is full in {}%'.format(100 / 0))
    except ZeroDivisionError as e:
        raise RocketNotReadyError('Problem with fuel gauge') from e


crew = ['John', 'Mary', 'Mike']
fuel = 100
check_list = [personnel_check, fuel_check]

print('Final check procedure')

for check in check_list:
    try:
        check()
    except RocketNotReadyError as f:
        print('RocketNotReady exception: "{}", caused by "{}"'.format(f, f.__cause__))
```

Exceptions – atrybut traceback

- Każdy obiekt wyjątku posiada atrybut __traceback__.
- Python pozwala nam operować na szczegółach śledzenia, ponieważ każdy obiekt wyjątku (nie tylko łańcuchowy) posiada atrybut __traceback__.
- Użyjmy do tego poprzedniego przykładu

```
class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e


crew = ['John', 'Mary', 'Mike']

print('Final check procedure')

try:
    personnel_check()
except RocketNotReadyError as f:
    print(f.__traceback__)
    print(type(f.__traceback__))
```

Exceptions – atrybut traceback

- mamy do czynienia z obiektem typu traceback,
- Aby to osiągnąć, moglibyśmy użyć metody format_tb() dostarczanej przez wbudowany moduł traceback, aby uzyskać listę łańcuchów opisujących śledzenie.
- Moglibyśmy użyć metody print_tb(), również dostarczanej przez moduł traceback, aby wypisać napisy bezpośrednio na standardowe wyjście.
- Odpowiednie dane wyjściowe ujawniają sekwencję wyjątków i dowodzą, że wykonanie nie zostało przerwane przez wyjątki, ponieważ widzimy końcowe sformułowanie Kontrola końcowa dobiegła końca.
- Teraz twój dziennik może być wypełniony wieloma szczegółami dotyczącymi wystrzelenia rakiety do późniejszego zbadania.
- W rzeczywistych projektach rozwojowych możesz wykorzystać zarejestrowane ślady po kompleksowych sesjach testowych w celu zebrania statystyk, a nawet zautomatyzowania procesów zgłaszania błędów.
- Aby uzyskać więcej informacji na temat połączonych wyjątków i atrybutów śledzenia wstecznego, należy zapoznać się z dokumentem PEP 3134

```
import traceback

class RocketNotReadyError(Exception):
    pass


def personnel_check():
    try:
        print("\tThe captain's name is", crew[0])
        print("\tThe pilot's name is", crew[1])
        print("\tThe mechanic's name is", crew[2])
        print("\tThe navigator's name is", crew[3])
    except IndexError as e:
        raise RocketNotReadyError('Crew is incomplete') from e


crew = ['John', 'Mary', 'Mike']

print('Final check procedure')

try:
    personnel_check()
except RocketNotReadyError as f:
    print(f.__traceback__)
    print(type(f.__traceback__))
    print('\nTraceback details')
    details = traceback.format_tb(f.__traceback__)
    print("\n".join(details))

print('Final check is over')
```

Exceptions – try except else

- W Pythonie, używając instrukcji else, możesz poinstruować program, aby wykonał określony blok kodu tylko w przypadku braku wyjątków

try:

```
    # Wykonywany kod
```

except:

```
    # wykonywany blok jeżeli wystąpi wyjątek.
```

else:

```
    # jeżeli nie wystąpi wyjątek, wykonywany jest następujący blok.
```

```
try:
    windows_interaction()
    #linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
```

```
def division_a_b(a, b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print("a/b result in 0")
    else:
        print(c)

division_a_b(2.0, 3.0)
division_a_b(3.0, 3.0)
```

```
import sys

def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')

def windows_interaction():
    assert ('win' in sys.platform), "Function can only run on Windows systems."
    print('Windows setting up')

try:
    windows_interaction()
    #linux_interaction()
except AssertionError as error:
    print(error)
else:
    print('Executing the else clause.')
```

Exceptions – try except else finally

- Wyobraź sobie, że zawsze musiałeś zaimplementować jakąś akcję, aby posprzątać po wykonaniu kodu. Python umożliwia to za pomocą klauzuli finally

try:

```
    # Wykonywany kod
```

except:

```
    # wykonywany blok jeżeli wystąpi wyjątek.
```

else:

```
    # jeżeli nie wystąpi wyjątek, wykonywany jest następujący blok.
```

finally:

```
    # ten blok kodu zawsze jest odpalany
```

```
import sys

def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')

def windows_interaction():
    assert ('win' in sys.platform), "Function can only run on Windows systems."
    print('Windows setting up')

try:
    windows_interaction()
    #linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

Exceptions – podsumowanie

- raise pozwala rzucić wyjątek w dowolnym momencie.
- assert umożliwia sprawdzenie, czy określony warunek jest spełniony, i zgłoszenie wyjątku, jeśli nie jest.
- W klauzuli try wszystkie instrukcje są wykonywane do momentu napotkania wyjątku.
- except służy do przechwytywania i obsługi wyjątków napotkanych w klauzuli try.
- else umożliwia tworzenie sekcji kodu, które powinny być uruchamiane tylko wtedy, gdy w klauzuli try nie napotkano żadnych wyjątków.
- Finally umożliwia wykonywanie sekcji kodu, które powinny zawsze działać, z wcześniej napotkanymi wyjątkami lub bez nich.

Exceptions – przykład 1

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print("Error: can\'t find file or read data")
else:
    print("Written content in the file successfully")
    fh.close()
```

```
try:
    fh = open("testfile", "r")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print("Error: can\'t find file or read data")
else:
    print("Written content in the file successfully")
    fh.close()
```

Exceptions – przykład 2

```
import math

class NumTypeError(TypeError):
    pass

class NegativeNumError(Exception):
    def __init__(self):
        super().__init__("Negative number not supported")

def sqrt(num):
    if not isinstance(num, (int, float)) :
        raise NumTypeError("only numbers are allowed")

    if num < 0:
        raise NegativeNumError

    return math.sqrt(num)

if __name__ == "__main__":
    try:
        print(sqrt(9))
        print(sqrt('a'))
        print(sqrt(-9))
    except NumTypeError as e:
        print(e)
    except NegativeNumError as e:
        print(e)
```

Zadanie 5

Wykorzystując kod zawarty w exceptions_zadanie.py:

- Spróbuj rozszerzyć skrypt listy kontrolnej, aby obsługiwał więcej różnych kontroli, wszystkie zgłaszane jako wyjątki RocketNotReady.
- Dodaj własne kontrole dla: baterii i obwodów.

3

Domknięcią funkcji

Closures

Closures – wstęp

- W Pythonie możesz zdefiniować funkcję z wnętrza innej funkcji. Ta funkcja jest nazywana funkcją zagnieżdżoną.
- Zamknięcie Pythona to funkcja zagnieżdżona, która pozwala nam na dostęp do zmiennych funkcji zewnętrznej nawet po jej zamknięciu.
- Podobnie jak zagnieżdżone pętle, możemy również zagnieżdżać funkcje. To powiedziawszy, Python daje nam możliwość definiowania funkcji w ramach funkcji.
- Zamknięcia Pythona to te wewnętrzne funkcje, które są zawarte w funkcji zewnętrznej.
- Zamknięcia mogą uzyskiwać dostęp do zmiennych obecnych w zewnętrznym zakresie funkcji. Może uzyskiwać dostęp do tych zmiennych nawet po zakończeniu wykonywania funkcji zewnętrznej.
- Z definicji domknięcie jest funkcją zagnieżdżoną, która odwołuje się do jednej lub więcej zmiennych ze swojego otaczającego zakresu.

```
def outer_func():
    greeting = 'Hello in '

    def inner_func():
        print(greeting + inner_func.__name__ + " but calling in " + outer_func.__name__ )

    inner_func()
```

```
def outer_func():
    greeting = 'Hello in '

    def inner_func():
        print(greeting + inner_func.__name__ + " but calling in " + outer_func.__name__ )

    return inner_func
```

Closures – wstęp

```
def outer_func():
    greeting = 'Hello in '

    def inner_func():
        print(greeting + inner_func.__name__ + " but calling in " + outer_func.__name__ )

    inner_func()
```

- W tym przykładzie definiujemy inner_func wewnątrz funkcji outer_func.
- Funkcja inner_func jest nazywana funkcją zagnieżdżoną.
- Wewnątrz funkcji inner_func uzyskujesz dostęp do zmiennej greeting z jej zakresu nielokalnego.
- Python nazywa zmienną greeting zmienną wolną
- Kiedy patrzysz na funkcję inner_fuc, faktycznie patrzysz na samą funkcję inner_func oraz zmienną greeting
- Resumując połączenie funkcji inner_func i zmiennej greeting nazywamy zamknięciem.
- Z definicji domknięcie jest funkcją zagnieżdżoną, która odwołuje się do jednej lub więcej zmiennych ze swojego otaczającego zakresu.

```
def outer_func():
    greeting = 'Hello in '

    def inner_func():
        print(greeting + inner_func.__name__ + " but calling in " + outer_func.__name__ )

    return inner_func
```

- W tym przykładzie funkcja outer_func zwraca funkcję inner_func zamiast jej wykonywania.
- Ponadto, gdy funkcja outer_func zwraca funkcję inner_func, w rzeczywistości zwraca zamknięcie

Closures – adres pamięci

Wartość zmiennej greeting jest współdzielona między dwoma zakresami:

- Funkcja outer_func
- Zamknięcie

```
def outer_func():
    greeting = 'Hello in '
    print(hex(id(greeting)))

    def inner_func():
        print(greeting + inner_func.__name__ + " but calling in "
+ outer_func.__name__ )
        print(hex(id(greeting)))

    return inner_func

fn = outer_func()
fn()
```

- Aby znaleźć adres pamięci obiektu komórki, można użyć właściwości __closure__ który zwraca tuple

```
print(fn.__closure__)
```

- Adres pamięci parametru greeting użyty w outer_fun jak i inner_fun jest taki sam
- Kiedy uzyskasz dostęp do wartości zmiennej greeting, Python technicznie wykona „podwójny skok”, aby uzyskać wartość ciągu.
- Gdy funkcja outer_func() była poza zakresem, nadal można uzyskać dostęp do obiektu łańcuchowego, do którego odwołuje się zmienna powitania.
- W oparciu o ten mechanizm można myśleć o domknięciu jako o funkcji i rozszerzonym zakresie zawierającym wolne zmienne
- Aby znaleźć wolne zmienne, które zawiera zamknięcie, możesz użyć __code__.co_freevars

```
print(fn.__code__.co_freevars)
```

Closures – kiedy python tworzy zamknięcie

- Python tworzy nowy zasięg podczas wykonywania funkcji. Jeśli ta funkcja tworzy zamknięcie, Python również tworzy nowe zamknięcie.
- Rozważmy przykład multipier
- Funkcja multiplier zwraca multiplication (mnożenie) dwóch argumentów. Jednak zamiast tego używa zamknięcia.
- Później wywołujemy trzykrotnie funkcję multiplier
- Te wywołania funkcji tworzą trzy domknięcia. Każda funkcja mnoży liczbę przez 1, 2 i 3

```
def multiplier(x):
    def multiply(y):
        return x * y
    return multiply

m1 = multiplier(1)
m2 = multiplier(2)
m3 = multiplier(3)

a = 10

print(m1(a))
print(m2(a))
print(m3(a))
```

Closures – pętle

- Najpierw deklarujemy listę, która będzie przechowywać domknięcia.
- Następnie użyte zostało wyrażenia lambda, aby utworzyć zamknięcie i dołączyć zamknięcie do listy w każdej iteracji.
- Później rozpakowane zostają domknięcia z listy do zmiennych m1, m2 i m3.
- Na koniec przekazujemy wartości 10, 20 i 30 do każdego zamknięcia i wykonaj je.
- Wynik jednak jest inny niż spodziewany, x zaczyna się od 1 do 3 w pętli. Po pętli jego wartość wynosi 3.
- Każdy element listy jest następującym zamknięciem lambda y: x*y
- Python ewaluuje x gdy wywoływana jest m1(10), m2(10) i m3(10). W chwili wykonania domknięć x wynosi 3.
- W celu naprawienia tego należy poinstruować pythona aby przypisywał wartość x w pętli.

```
multipliers = []
for x in range(1, 4):
    multipliers.append(lambda y: x * y)

m1, m2, m3 = multipliers
a = 10

print(m1(a))
print(m2(a))
print(m3(a))
```

```
def multiplier(x):
    def multiply(y):
        return x * y
    return multiply


multipliers = []
for x in range(1, 4):
    multipliers.append(multiplier(x))

m1, m2, m3 = multipliers

print(m1(a))
print(m2(a))
print(m3(a))
```

Closures – kiedy można wykorzystywać zamknięcia

- Aby zastąpić niepotrzebne użycie klasy: Załóżmy, że masz klasę, która oprócz metody __init__ zawiera tylko jedną metodę. W takich przypadkach często bardziej elegancko jest użyć zamknięcia zamiast klasy.
- Aby uniknąć używania zasięgu globalnego: Jeśli masz zmienne globalne, z których będzie korzystać tylko jedna funkcja w twoim programie, pomyśl o zamknięciu. Zdefiniuj zmienne w funkcji zewnętrznej i użyj ich w funkcji wewnętrznej.
- Aby zaimplementować ukrywanie danych: Jedynym sposobem uzyskania dostępu do zamkniętej funkcji jest wywołanie funkcji otaczającej. Nie ma możliwości bezpośredniego dostępu do funkcji wewnętrznej.
- Aby zapamiętać środowisko funkcji nawet po zakończeniu wykonywania: Możesz później uzyskać dostęp do zmiennych tego środowiska w swoim programie.

Closures – pętle

- Ponieważ domknięcia są używane jako funkcje wywołania zwrotnego, zapewniają one pewnego rodzaju ukrywanie danych. Pomaga nam to ograniczyć użycie zmiennych globalnych.
- Kiedy mamy niewiele funkcji w naszym kodzie, domknięcia okazują się skutecznym sposobem. Ale jeśli potrzebujemy mieć wiele funkcji.

```
import logging

logging.basicConfig(filename='example.log',
                    level=logging.INFO)


def logger(func):
    def log_func(*args):
        logging.info(
            'Running "{}" with arguments
{}'.format(func.__name__,
                                                args))
        print(func(*args))

    return log_func


def add(x, y):
    return x + y


def sub(x, y):
    return x - y


add_logger = logger(add)
sub_logger = logger(sub)
add_logger(3, 3)
add_logger(4, 5)
sub_logger(10, 5)
sub_logger(20, 10)
```

Closures – klasa vs funkcja

```
class Summer():

    def __init__(self):
        self.data = []

    def __call__(self, val):

        self.data.append(val)
        _sum = sum(self.data)

        return _sum

summer = Summer()

s = summer(1)
print(s)
s = summer(2)
print(s)
s = summer(3)
print(s)
s = summer(4)
print(s)
```

```
def make_summer():

    data = []

    def summer(val):

        data.append(val)
        _sum = sum(data)

        return _sum

    return summer

summer = make_summer()
s = summer(1)
print(s)
s = summer(2)
print(s)
s = summer(3)
print(s)
s = summer(4)
print(s)
```

Closures – używanie nonlocal

- Ten kod wykonuje funkcję zewnętrzną calculate() i zwraca zamknięcie liczby nieparzystej (odd)
- Właśnie dlatego możemy uzyskać dostęp do zmiennej num metody calculate() nawet po wykonaniu funkcji zewnętrznej.
- Użycie słowa kluczowego nonlocal pozwala na modyfikacje zmiennej num wewnątrze funcji inner_func()

```
def calculate():
    num = 1
    def inner_func():
        nonlocal num
        num += 2
        return num
    return inner_func

odd = calculate()

print(odd())
print(odd())
print(odd())

odd2 = calculate()
print(odd2())
```

Closures – przykład

```
def pop(list):
    def get_last_item(my_list):
        # usuwa ostatni element z listy otrzymany przez
zewnętrzną funkcję
        return my_list[len(list) - 1]

    # usuwa ostatni element listy
    list.remove(get_last_item(list))

    # zwraca liste
    return list

test_list = [1,2,3,4,5]
print(pop(test_list))
print(pop(test_list))
print(pop(test_list))
print(pop(test_list))
```

```
def make_point(x, y):
    def point():
        print(f"Point({x}, {y})")
    def get_x():
        return x
    def get_y():
        return y
    def set_x(value):
        nonlocal x
        x = value
    def set_y(value):
        nonlocal y
        y = value

    point.get_x = get_x
    point.set_x = set_x
    point.get_y = get_y
    point.set_y = set_y

    return point

point = make_point(1, 2)
print(point.get_x())
print(point.get_y())

point.set_x(42)
point.set_y(7)
point()
```

Closures – Reguła LEGB

- Każda zmienna żyje w określonej przestrzeni nazw.
- Kiedy mówimy o szukaniu wartości nazwy w odniesieniu do kodu, do przestrzeni nazw odnosi się zakres (scope).
- Funkcje definiują zakres lokalny, a moduły zakres globalny.
- Rozwiązywanie konfliktów w zakresie nazw w Pythonie nazywane jest regułą LEGB (local, enclosing, global, builtin).

local - zakres lokalny
enclosing - zakres lokalny instrukcji def lub wyrażeń lambda zawierających daną funkcję
global - zakres globalny (moduł)
builtin - zakres wbudowany