DialBB ドキュメント

リリース *v0.1*

Mikio Nakano

2022 年 08 月 01 日

Contents:

第1章　はじめに

DialBB (Dialogue Building Blocks) は対話システムを構築するためのフレームワークです.

対話システムは情報分野の様々な技術を統合して構築されます．本フレームワークを用いることで，対話システム技術の知識や情報システム開発経験の少ない人でも対話システムが構築でき，様々な情報技術を学ぶことができることを目指しています．また，アーキテクチャのわかりやすさ，拡張性の高さ，コードの読みやすさなどを重視し，プログラミング・システム開発教育の教材にしてもらえることも目指しています.

DialBB で対話システムを構築するには，Python を動かす環境が必要です．もし，Python を動かす環境がないなら，Python 環境構築ガイドなどを参考に，環境構築を行ってください．また，Python の解説書としては，みんなの Python 第 4 版がお薦めです.

対話システムの一般向けの解説として，東中竜一郎著：AI の雑談力や情報処理学会誌の解説記事があります.

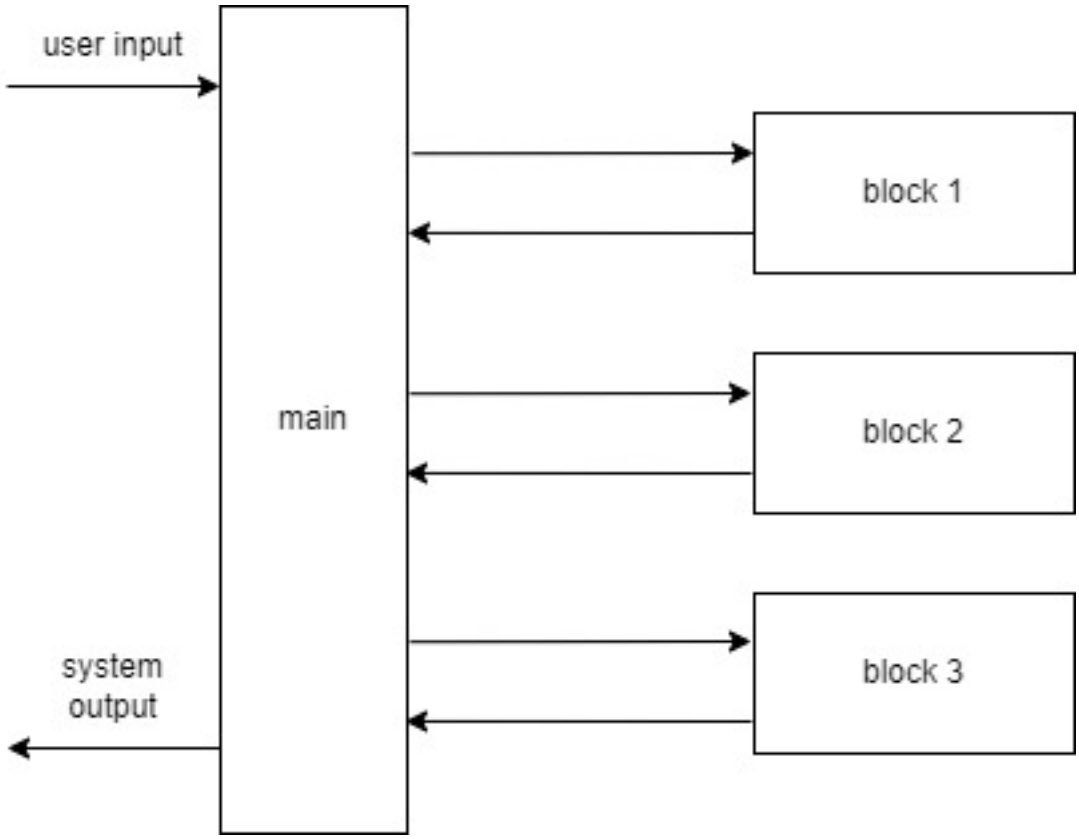
また，東中，稲葉，水上著：Python でつくる対話システムは，対話システムの実装の仕方について Python のコードを用いて説明しています.

第2章　DialBBの概要

はじめにに書いたように，DialBB は対話システムを作るためのフレームワークです．

フレームワークとは，それ単体でアプリケーションとして成立はしないが，データや追加のプログラムを与えることでアプリケーションを作成するものです．

DialBB のアプリケーションは，ブロックと呼ぶモジュールが順に処理を行うことで，ユーザからの入力発話に対するシステム発話を作成し返します．以下に基本的なアーキテクチャを示します．



メインモジュールは，対話の各ターンで入力されたデータ（ユーザ発話を含みます）を各ブロックが順次処理することによって応答を返します．このデータのことを payload と呼びます．各ブロックは，payload の要素のいくつかを受け取り，辞書形式のデータを返します．返されたデータは payload に追加されます．すでに同じキーを持つ要素が payload にある場合は上書きされます．

どのようなブロックを使うかは，configuration ファイルで設定します．ブロックは，あらかじめ DialBB が用意しているブロック（組み込みブロック）でもアプリケーション開発者が作成するブロックでも構いません．

メインモジュールが各ブロックにどのようなデータを渡し，どのようなデータを受け取るかも configuration ファイルで指定します．

詳細は「**フレームワーク仕様**」で説明します.

第3章　インストールとサンプルアプリの実行の仕方

本章では，DialBB をインストールしてサンプルアプリケーションを実行する方法について説明します．もし以下の作業を行うことが難しければ，詳しい人に聞いてください．

3.1 実行環境

Ubuntu 20.04 および Windows 10 上の python 3.8.10, python3.7.9 で動作確認を行っていますが，MacOS や，バージョン 3.9 以上の Python でも動作すると考えられます．

3.2 DialBB のインストール

github のソースコードを clone します．

```
$ git clone git@github.com:c4a-ri/dialbb.git
```

3.3 pytyhon library のインストール

clone したディレクトリに移動し，以下を実行してください．

```
$ cd dialbb
$ pip install -r requirements.txt （python 3.8 の場合）
$ pip install -r requirements3.7.txt （python 3.7 の場合）
$ python -m snips_nlu download en
$ python -m snips_nlu download ja
```

- 注意
  - Windows 上の Anaconda を用いて実行する場合，Anaconda Prompt を管理者モードで起動しないといけない可能性があります．
  - pyenv を使っている場合，以下のエラーが出る可能性があります．

    ```
    ModuleNotFoundError: No module named '_bz2'
    ```

    それに対する対処法はこの記事などを参照ください．

3.4 graphviz のインストール

Graphviz のサイトなどを参考に graphviz をインストールします．ただ，Graphviz がなくてもアプリケーションを動作させることは可能です．

3.5 オウム返しサンプルアプリのサーバの起動

ただオウム返しを行うアプリです．日本語アプリのみです．

```
$ python run_server.py sample_apps/parrot/config.yml
```

3.5.1 動作確認

別のターミナルから以下を実行してください．

- 最初のアクセス

```
$ curl -X POST -H "Content-Type: application/json" \
  -d '{"user_id":"user1"}' http://localhost:8080/init
```

以下のレスポンスが帰ります．

```
{"aux_data":{},
 "session_id":"dialbb_session1",
 "system_utterance":"こちらはオウム返し bot です．何でも言って見てください．",
 "user_id":"user1"}
```

- 2 回目以降のアクセス

```
$ curl -X POST -H "Content-Type: application/json" \
  -d '{"user_utterance": "こんにちは", "user_id":"user1", "session_id":"dialbb_
↪session1"}' \
  http://localhost:8080/dialogue
```

以下のレスポンスが帰ります．

```
{"aux_data":null,
 "session_id":"dialbb_session1",
 "system_utterance":"「こんにちは」と仰いましたね．",
 "user_id":"user1"}
```

3.6 ビルトインブロックを用いたサンプルアプリ

DialBB には，あらかじめ作成してあるブロック（ビルトインブロック）を用いたサンプルアプリがあります．

3.6.1 起動

以下のコマンドで起動します

- 英語アプリ

```
$ python run_server.py sample_apps/network_en/config.yml
```

- 日本語アプリ

```
$ python run_server.py sample_apps/network_ja/config.yml
```

3.6.2 動作確認

上記でアプリを起動したサーバのホスト名か IP アドレスをとしたとき，ブラウザから以下の URL に接続すると対話画面が現れますので，そこで対話してみてください．

```
http://<hostname>:8080
```

サーバを Windows 上で動作させた場合，ブラウザ上に対話画面が出ないことがあります．その場合は，以下の URL に接続すると，簡易な対話画面が出ます．

```
http://localhost:8080/test
```

3.6.3 テストセットを用いた動作確認

以下のコマンドで，ユーザ発話を順に処理して対話するテストを行うことができます．

- 英語

```
$ python dialbb/util/test.py  sample_apps/network_en/config.yml sample_apps/network_
↪en/test_inputs.json
```

- 日本語

```
$ python dialbb/util/test.py  sample_apps/network_ja/config.yml sample_apps/network_
↪ja/test_inputs.json
```

第4章　日本語サンプルアプリケーションの説明

本節では，sample_apps/network_ja にあるサンプルアプリケーションを通して，DialBB アプリケーションの構成を説明します．

sample_apps/network_ja ディレクトリ（フォルダ）をコピーして編集することで，違うアプリケーションを作ることができます．どこにコピーしても構いません．

4.1 ファイル構成

sample_apps/network_ja には以下のファイルが含まれています．

ファイル名	説明
config.yml	アプリケーションを規定する configuration ファイル
sample-knowledge-ja.xlsx	言語理解ブロック，対話管理ブロックで用いる知識を記述したもの
scenario_functions.py	対話管理ブロックで用いるプログラム
test_inputs.json	システムテストで使うデータ

4.2 システム構成とコンフィギュレーション

4.2.1 入出力

- ここでは DialBB のアプリケーションの入出力のデータ構造を説明します．なお，ブラウザから接続して使う場合，これらのデータ構造を意識する必要はありません．
- 各ターン（一回の発話のやりとりのこと）での入力は以下のような辞書形式のデータです．
  - 対話開始時

```
{
  "user_id": <ユーザ ID: 文字列>,
  "aux_data": <補助データ：データ型は任意>
}
```

  - 対話開始後

```
{
   "user_id": <ユーザ ID：文字列>,
   "session_id": <セッション ID：文字列>,
   "user_utterance": <ユーザ発話：文字列>,
   "aux_data":<補助データ：データ型は任意>
}
```

- 各ターンでの出力は以下のような辞書形式のデータです．

```
{
  "session_id":<セッション ID: 文字列>,
  "system_utterance",<システム発話文字列: 文字列>,
  "user_id":<ユーザ ID: 文字列>,
  "final": <対話終了フラグ: ブール値>
  "aux_data": <補助データ: オブジェクト>
}
```

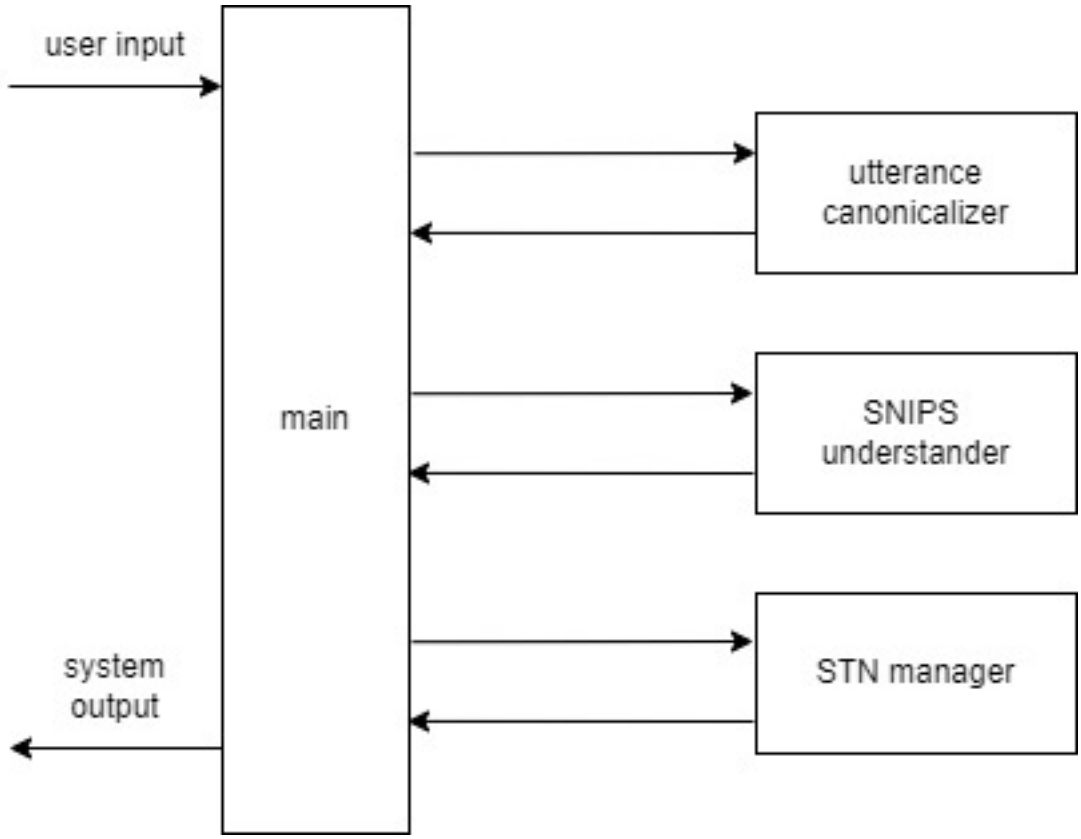
4.2.2 ブロック

本アプリケーションでは，以下の 3 つの組み込みブロックを利用しています．

- Utterance canonicalizer: ユーザ入力文の正規化（大文字 → 小文字，全角⇒半角の変換など）を行います．
- SNIPS understander: 言語理解を行います．SNIPS_NLU を利用して，ユーザ発話タイプ（インテントとも呼びます）の決定とスロットの抽出を行います．
- STN manager: 状態遷移ネットワーク (State-Transition Network) を用いて対話管理を行います．

組み込みブロックとは，DialBB にあらかじめ含まれているブロックです．これらの組み込みブロックの詳細は，builtin_blocks で説明しています．

本アプリケーションは以下のようなシステム構成をしています．



4.2.3 コンフィギュレーション

本アプリケーションのコンフィギュレーションファイルは以下のような yaml ファイルです.

```
language: ja   # 言語を指定

blocks:  # ブロックのリスト
  - name: canonicalizer  # ブロック名
    # ブロックのクラス
    block_class: preprocess.utterance_canonicalizer.UtteranceCanonicalizer
    input: # ブロックへの入力
      input_text: user_utterance
    output: # ブロックからの出力
      output_text: canonicalized_user_utterance
  - name: understander
    block_class: understanding_with_snips.snips_understander.Understander
    input:
      input_text: canonicalized_user_utterance
    output:
      nlu_result: nlu_result
    knowledge_file: sample-knowledge-ja.xlsx  # 知識記述ファイル
  - name: manager
    block_class: stn_management.stn_manager.Manager
```

(次のページに続く)

(前のページからの続き)

```
    knowledge_file: sample-knowledge-ja.xlsx # 知識記述ファイル
    function_definitions: scenario_functions  # 知識記述の中で用いる関数の定義ファイル
    input:
      sentence: canonicalized_user_utterance
      nlu_result: nlu_result
      user_id: user_id
      session_id: session_id
    output:
      output_text: system_utterance
      final: final
```

blocks 要素は各ブロックのコンフィギュレーションのリストです．この順に処理が行われます．

name はブロックの名前で，ロギングなどに使われます．

block_class はブロックのクラス名です．このアプリケーションではあらかじめ用意してあるクラスのみを使います．

input と output は，ブロックへの入出力を規定します．‘ :‘ の左側がブロック内で参照するためのキーで，右側が payload でのキーです．例えば，canonicalizer ブロックの出力の output_text 要素が payload の canonicalized_user_utterance 要素の値になり，これが，understander ブロックへの入力の input_text 要素になります．

これらの要素に加え，各ブロックのコンフィギュレーションでは，ブロック独自の要素を持つことができます．例えば，understander や manager には knowledge_file 要素が，manager には function_definition 要素があります．これらをどう使うかは，ブロックのクラス定義の中で決められています．

4.3 言語理解

4.3.1 言語理解結果

言語理解ブロックは，入力発話を解析し，タイプとスロットを抽出します．

例えば，「好きなのは醤油」の言語理解結果は次のようになります．

```
{"type": "特定のラーメンが好き", "slots": {"favarite_ramen": "醤油ラーメン"}}}
```

"特定のラーメンが好き"がタイプで，"favarite_ramen"スロットの値が"醤油ラーメン"です．複数のスロットを持つような発話もあり得ます．

4.3.2 言語理解知識

言語理解用の知識は，sample-knowledge-ja.xlsx に書かれています．

言語理解知識は，以下の４つのシートからなります．

シート名	内容
utterances	タイプ毎の発話例
slots	スロットとエンティティの関係
entities	エンティティに関する情報
dictionary	エンティティ毎の辞書エントリーと同義語

これらの詳細は「**言語理解知識**」を参照してください．

4.3.3 SNIPS 用の訓練データ

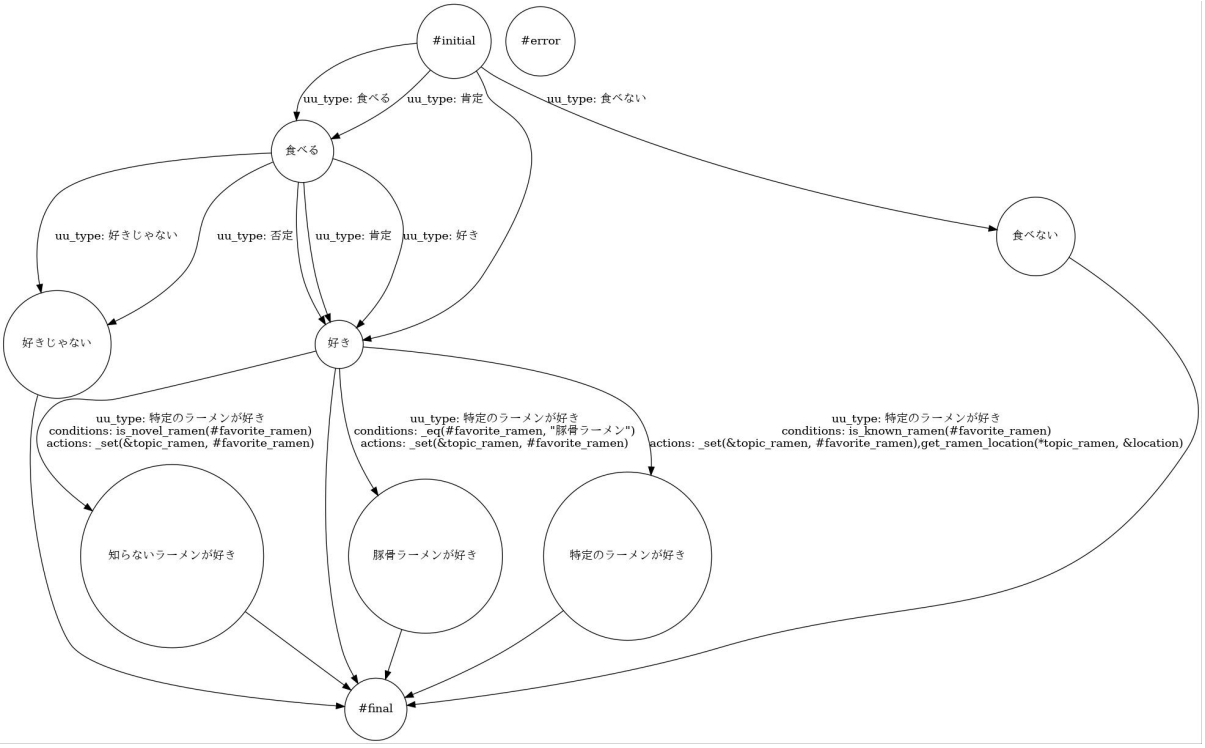
アプリを立ち上げると上記の知識は SNIPS 用の訓練データに変換され，モデルが作られます．

SNIPS 用の訓練データはアプリのディレクトリの_training_data.json です．このファイルを見ることで，うまく変換されているかどうかを確認できます．

4.4 対話管理

対話管理知識（シナリオ）は，sample-knowledge-ja.xlsx ファイルの scenario シートです．このシートの書き方の詳細は「**対話管理の知識記述**」を参照してください．

Graphviz がインストールされていれば，アプリケーションを起動したとき，シナリオファイルから生成した状態遷移ネットワークの画像ファイルを出力します．以下が本アプリケーションの状態遷移ネットワークです．



シナリオファイルで用いている遷移の条件や遷移後に実行する関数のうち，組み込み関数でないものが scenario_functions.py で定義されています.

第5章　フレームワーク仕様

ここではフレームワークとしての DialBB の仕様を説明します．Python プログラミングの知識がある読者を想定しています．

5.1 概要

DialBB のメインモジュールは，メソッド呼び出しまたは Web API 経由で，ユーザ発話を JSON 形式で受けとり，システム発話を JSON 形式で返します．

メインモジュールは，ブロックと呼ぶいくつかのサブモジュールを順に呼び出すことによって動作します．各ブロックは JSON 形式 (python の dict のデータ) を受け取り，JSON 形式のデータを返します．

各ブロックのクラスや入出力仕様はアプリケーション毎のコンフィギュレーションファイルで規定します．

5.2 入出力

5.2.1 WebAPI

サーバの起動

```
$ python run_server.py <config file>
```

セッションの開始時

- URI

  ```
  http://<server>:8080/init
  ```

- リクエストヘッダ

  ```
  Content-Type: application/json
  ```

- リクエストボディ

  以下の形の JSON です．

```
{
  "user_id": <ユーザ ID: 文字列>,
  "aux_data": <補助データ: データ型は任意>
}
```

  - user_id は必須で，aux_data は任意です．
  - <ユーザ ID>はユーザに関するユニークな ID です. 同じユーザが何度も対話する際に，以前の対話の内容をアプリが覚えておくために用います．
  - <補助データ>は，クライアントの状態をアプリに送信するために用います．フォーマットは任意の JSON オブジェクトで，アプリ毎に決めます．

- レスポンス

```
{
  "session_id":<セッション ID: 文字列>,
  "system_utterance": <システム発話文字列: 文字列>,
  "user_id":<ユーザ ID: 文字列>,
  "final": <対話終了フラグ: ブール値>
  "aux_data":<補助データ: データ型は任意>
}
```

  - <セッション ID>は，対話のセッションの ID です．この URI に POST する度に新しいセッション ID が生成されます．
  - <システム発話文字列>は，システムの最初の発話（プロンプト）です．
  - <ユーザ ID>は，リクエストで送られたユーザの ID です．
  - <対話終了フラグ>は，対話が終了したかどうかを表すブール値です．
  - <補助データ>は，対話アプリがクライアントに送信するデータです．サーバの状態などを送信するのに使います．

セッション開始後の対話

- URI

```
http://<server>:8080/dialogue
```

- リクエストヘッダ

```
Content-Type: application/json
```

- リクエストボディ

  以下の形の JSON です．

```
{"user_id": <ユーザ ID: 文字列>,
 "session_id": <セッション ID: 文字列>,
 "user_utterance": <ユーザ発話文字列: 文字列>,
 "aux_data":<補助データ: データ型は任意>}
```

  - user_id, session_id, user_utterance は必須．aux_data は任意です．
  - <セッション ID>は，サーバから送られたセッション ID です．
  - <ユーザ発話文字列>は，ユーザが入力した発話文字列です．

- レスポンス

  セッションの開始時のレスポンスと同じです．

5.2.2 クラス API

クラス API を用いる場合，dialbb.main.DialogueProcessor クラスのオブジェクトを作成することで，DialBB のアプリケーションを作成します．

これは以下の手順で行います．

- 以下のように，環境変数 PYTHONPATH に DialBB のディレクトリを追加します．

```
export PYTHONPATH=<DialBB のディレクトリ>:$PYTHONPATH
```

- python を立ち上げるか, DialBB を呼び出すアプリケーションの中で, 以下のように DialogueProcessor のインスタンスを作成し，process メソッドを呼び出します．

```
>>> from dialbb.main import DialogueProcessor
>>> dialogue_processor = DialogueProcessor(<configuration ファイル> <追加の
configuration>)
>>> response = dialogue_processor.process(<リクエスト>, initial=True) # 対話の開始
時
>>> response = dialogue_processor.process(<リクエスト>) # それ以降
```

  <追加の configuration>は，以下のような辞書形式のデータで，key は文字列でなければなりません．

```
{
  "<key1>": <value1>,
    "<key2>": <value2>
...
}
```

  これは，configuration ファイルから読み込んだデータに追加して用いられます．もし，configuration ファイルと追加の configuration で同じ key が用いられていた場合，追加の configuration の値が用いられます．

<リクエスト>と response（レスポンス）は辞書型のデータで，Web API のリクエスト，レスポンスと同じです．

5.3 configuration

configuration は辞書形式のデータで，yaml ファイルで与えることを前提としています．

configuration に必ず必要なのは blocks 要素のみです．blocks 要素は，各ブロックがどのようなものかを規定するもの（これを block configuration と呼びます）のリストで，以下のような形をしています．

```
blocks
  - <block configuration>
  - <block configuration>
  ...
  - <block configuration>
```

各 block configuration の必須要素は以下です．

- name

  ブロックの名前．ログで用いられます．

- block_class

  ブロックのクラス名です．組み込みクラスの場合は dialbb.builtin_blocks からの相対パスで記述します．開発者の自作ブロックの場合は，モジュールが検索されるパスからの相対パスで記述します．configuration ファイルのあるディレクトリは，モジュールが検索されるパス（sys.path の要素）に自動的に登録されます．

- input

  メインモジュールからブロックへの入力を規定します．辞書型のデータで，key がブロック内での参照に用いられ，value が payload（メインモジュールで保持されるデータ）での参照に用いられます．例えば，

  ```
  input:
    sentence: canonicalized_user_utterance
  ```

  のように指定されていたとすると，ブロック内で input['sentence'] で参照できるものは，メインモジュールの payload['canonicalized_user_utterance'] です．

- output

  ブロックからメインモジュールへの出力を規定します．input 同様，辞書型のデータで，key がブロック内での参照に用いられ，value が payload での参照に用いられます．

  ```
  output:
    output_text: system_utterance
  ```

の場合，ブロックからの出力を output とすると，

```
        payload['system_utterance'] = output['output_text']
```

の処理が行われます．payload がすでに system_utterance をキーとして持っていた場合は，その値は上書きされます．

5.4 ブロックの自作方法

開発者は自分でブロックを作成することができます．

ブロックのクラスは diabb.abstract_block.AbstractBlock の子クラスでないといけません．

5.4.1 実装すべきメソッド

- __init__(self, *args)

  コンストラクタです．

```
def __init__(self, *args):

    super().__init__(*args)

    <このブロック独自の処理>
```

- process(self, input: Dict[str, Any], initial: bool = False) -> Dict[str, Any]

  入力 input を処理し，出力を返します．input はメインモジュールから渡される辞書型データです．入力，出力とメインモジュールの payload の関係は configuration で規定されます．initial が True の時は対話開始時の処理を行います．（ユーザ発話は空文字列です．）

5.4.2 利用できる変数

- self.config

  configuration の内容を辞書型データにしたものです．これを参照することで，独自に付け加えた要素を読みこむことが可能です．

- self.block_config

  block configuration の内容を辞書型データにしたものです．これを参照することで，独自に付け加えた要素を読みこむことが可能です．

- self.name

  ブロックの名前です．(string)

- self.config_dir

  configuration ファイルのあるディレクトリです．

5.4.3 利用できるメソッド

以下のログメソッドが利用できます.

- log_debug(self, message: str, session_id: str = "unknown")

  標準エラー出力に debug レベルのログを出力します．`session_id` にセッション ID を指定するとログに含めることができます.

- log_info(self, message: str, session_id: str = "unknown")

  標準エラー出力に info レベルのログを出力します.

- log_warning(self, message: str, session_id: str = "unknown")

  標準エラー出力に warning レベルのログを出力します.

- log_error(self, message: str, session_id: str = "unknown")

  標準エラー出力に error レベルのログを出力します.

5.5 デバッグモード

Python 起動時の環境変数 `DIALBB_DEBUG` の値が `yes`（大文字小文字は問わない）の時，デバッグモードで動作します．この時，`dialbb.main.DEBUG` の値が `True` になります．アプリ開発者が作成するブロックの中でも以下のこの値を参照することができます.

`dialbb.main.DEBUG` が `True` の場合，ログレベルは debug に設定され，その他の場合は info に設定されます.

第6章　組み込みブロックの仕様

組み込みブロックとは，DialBB にあらかじめ含まれているブロックです．

6.1 Utterance canonicalizer

(preprocess.utterance_canonicalizer.UtteranceCanonicalizer)

ユーザ入力文の正規化を行います．

configuration の language 要素が ja の場合は日本語，en の場合は英語用の正規化を行います．

- 入力
  - input_text: ユーザ発話文字列（文字列）
    - 例："ＣＵＰ Noodle 好き"
- 出力
  - output_text: 正規化後のユーザ発話（文字列）
    - 例："cupnoodle 好き"

正規化は以下の処理を行います．

- 大文字 → 小文字
- 全角 → 半角の変換（カタカナを除く）
- スペースの連続を一つのスペースに変換（英語のみ）
- スペースの削除（日本語のみ）

6.2 SNIPS understander

(understanding_with_snips.snips_understander.Understander)

SNIPS_NLU を利用して，ユーザ発話タイプ（インテントとも呼びます）の決定とスロットの抽出を行います．

configuration の language 要素が ja の場合は日本語，en の場合は英語の言語理解を行います．

本ブロックは，起動時に Excel で記述した言語理解用知識を読み込み，SNIPS の訓練データに変更し，SNIPS のモデルを構築します．

実行時は SNIPS のモデルを用いて言語理解を行います．

- 入力
  - `input_text`: 正規化後のユーザ発話（文字列）
    - 例："好きなのは醤油"
- 出力
  - `nlu_result`: 言語理解結果（辞書型）
    - 例：`{"type": "特定のラーメンが好き", "slots": {"favarite_ramen": "醤油ラーメン"}}`
- 知識記述は Excel ファイルで行います． block configuration の knowledge_file にファイル名を指定します．ファイル名は configuration ファイルからの相対パスで記述します．

6.2.1 言語理解知識

言語理解知識は，以下の４つのシートからなります．

シート名	内容
utterances	タイプ毎の発話例
slots	スロットとエンティティの関係
entities	エンティティに関する情報
dictionary	エンティティ毎の辞書エントリーと同義語

シート名は block configuration で変更可能ですが，変更することはほとんどないと思いますので，詳細な説明は割愛します．

utterances シート

各行は次のカラムからなります．

カラム名	内容
flag	利用するかどうかを決めるフラグ．Y: yes, T: test などを書くことが多い．どのフラグの行を利用するかはコンフィギュレーションに記述する．サンプルアプリのコンフィギュレーションでは，すべての行を使う設定になっている．
type	発話のタイプ（インテント）
utterance	発話例．スロットを (豚骨ラーメン)[favorite_ramen] が好きですのように (<スロットに対応する言語表現>)[<スロット名>] で表現する．スロットに対応する言語表現＝言語理解結果に表れる（すなわち manager に送られる）スロット値ではないことに注意．言語表現が dictionary の synonyms カラムにあるものの場合，スロット値は，dictionary シートの value カラムに書かれたものになる．

slots シート

各行は次のカラムからなります.

カラム名	内容
flag	utterances シートと同じ
slot	スロット名．utterances シートの発話例で使うもの．言語理解結果でも用いる．
entity	エンティティ名．スロットの値がどのようなタイプの名詞句なのかを表す．異なるスロットが同じエンティティを持つ場合がある．例えば，(東京)[source_station] から (京都)[destination_station] までの特急券を買いたいのように，source_station，destination_station とも station エンティティを取る．entity カラムの値は，SNIPS の builtin entity でも良い．(例: snips/city)

SNIPS の builtin entity を用いる場合，以下のようにしてインストールする必要があります.

```
$ snips-nlu download-entity snips/city ja
```

SNIPS の builtin entity を用いた場合の精度などの検証は不十分です.

entities シート

各行は次のカラムからなります.

カラム名	内容
flag	utterances シートと同じ
entity	エンティティ名
use synonyms	同義語を使うかどうか (Yes または No)
automatically extensible	辞書にない値でも認識するかどうか (Yes または No)
matching strictness	エンティティのマッチングの厳格さ 0.0 - 1.0

dictionary シート

各行は次のカラムからなります.

カラム名	内容
flag	utterances シートと同じ
entity	エンティティ名
value	辞書エントリー名．言語理解結果にも含まれる
synonyms	同義語を,,，,, で連結したもの

SNIPS の訓練データ

アプリを立ち上げると上記の知識は SNIPS の訓練データに変換され，モデルが作られます．

SNIPS の訓練データはアプリのディレクトリの_training_data.json です．このファイルを見ることで，うまく変換されているかどうかを確認できます．

6.3 STN manager

状態遷移ネットワーク (State-Transition Network) を用いて対話管理を行います．

- 入力
  - sentence: 正規化後のユーザ発話（文字列）
  - nlu_result:言語理解結果（辞書型）
  - user_id:ユーザ ID（文字列）
  - session_id セッション ID（文字列）
- 出力
  - output_text: システム発話（文字列）
    - 例："醤油ラーメン好きなんですね"
  - final: 対話終了かどうかのフラグ（ブール値）
  - aux_data 補助データ（辞書型）遷移した状態の ID を含めて返す
    - 例：{"state": "特定のラーメンが好き"}
- 知識記述は Excel ファイルで行います． block configuration の knowledge_file にファイル名を指定します．ファイル名は configuration ファイルからの相対パスで記述します．

6.3.1 対話管理の知識記述

対話管理知識（シナリオ）は，Excel ファイルの scenario シートです．

各行は次のカラムからなります．

カラム名	内容
flag	utterance シートと同じ
state	state の ID
system utterance	state の状態で生成されるシステム発話の候補．システム発話文字列に含まれる{<変数>}は，対話中にその変数に代入された値で置き換えられる．state が同じ行は複数あり得るが，同じ state の行の system utterance すべてが発話の候補となり，ランダムに生成される．
user utterance example	ユーザ発話の例．対話の流れを理解するために書くだけで，システムでは用いられない．
user utterance type	ユーザ発話を言語理解した結果得られるユーザ発話のタイプ．遷移の条件となる．
conditions	条件（の並び）．遷移の条件を表す関数呼び出し．複数あっても良い．複数ある場合は，；で連結する．各条件は<関数名>(<引数 1>，<引数 2>，...，<引数 n>) の形をしている．引数は 0 個でも良い．
actions	アクション（の並び）．遷移した際に実行する関数呼び出し．複数あっても良い．複数ある場合は，；で連結する．各条件は<関数名>(<引数 1>，<引数 2>，...，<引数 n>) の形をしている．引数は 0 個でも良い．
next state	遷移先の state

基本的に 1 行が一つの遷移を表します．各遷移の user utterance type が空かもしくは言語理解結果と一致し，conditions が空か全部満たされた場合，遷移の条件を満たし，next state に遷移します．その際，actions を実行します．

6.3.2 特別な state

以下の state ID はあらかじめ定義されています．

state ID	説明
#initial	初期状態．対話はこの状態から始まる．
#error	内部エラーが起きたときこの状態に移動する．システム発話を生成して終了する．

また，`#final_say_bye` のように，`#final` ではじまる state ID は最終状態を表します．最終状態ではシステム発話を生成して対話を終了します．

6.3.3 条件とアクション

STN Manager は，対話のセッションごとに文脈情報を保持しています．文脈情報は変数とその値の組の集合で，値はどのようなデータ構造でも構いません．

条件やアクションの関数は文脈情報にアクセスします．

関数の引数

condition や action で用いる関数の引数には次のタイプがあります．

引数のタイプ	形式	説明
特殊変数	#で始まる文字列	言語理解をもとにセットされる変数の値#<スロット名>: 直前のユーザ発話のスロット値．スロット値が空の場合は空文字列になる．#sentence: 直前のユーザ発話（正規化したもの）#user_id: ユーザ ID（文字列）
変数	*で始まる文字列	文脈情報における変数の値*<変数名>の形．変数の値は文字列でなくてはならない．文脈情報にその変数がない場合は空文字列になる．
変数参照	&で始まる文字列	&<変数名>: 文脈情報での変数の名前．関数定義内で文脈情報を参照するために用いる．
定数	""で囲んだ文字列	文字列

6.3.4 関数定義

condition や action で用いる関数は，DialBB 組み込みのものと，開発者が定義するものがあります．condition で使う関数は bool 値を返し，action で使う関数は何も返しません．

組み込み関数

組み込み関数には以下があります．

関数	condition or action	説明	使用例
_eq(x, y)	condition	x と y が同じなら True を返す	_eq(*a, "b"): 変数 a の値が"b"なら True を返す. _eq(#food, "ラーメン"): #food スロットが"ラーメン"なら True を返す
_ne(x, y)	condition	x と y が同じでなければ True を返す	_ne(*a, *b): 変数 a の値と変数 b の値が異なれば True を返す_ne(#food, "ラーメン"): #food スロットが"ラーメン"なら False を返す
_contains(x, y)	condition	x が文字列として y を含む場合 True を返す	_contains(#sentence, "はい") : ユーザ発話が「はい」を含めば True を返す
_not_contains(x, y)	condition	x が文字列として y を含まない場合 True を返す	_not_contains(#sentence, "はい") : ユーザ発話が「はい」を含めば True を返す
_member(x, y)	condition	文字列 y を':' で分割してできたリストに文字列 x が含まれていれば True を返す	_member(#food, "ラーメン:チャーハン:餃子")
_not_member(x, y)	condition	文字列 y を':' で分割してできたリストに文字列 x が含まれていなければ True を返す	_member(*favorite_food, "ラーメン:チャーハン:餃子")
_set(x, y)	action	変数 x に y をセットする	_set(&a, b): b の値を a にセットする．例:_set(&a, "hello"): a に"hello"をセットする．

開発者による関数定義

開発者が関数定義を行うときには，アプリケーションディレクトリの scenario_functions.py を編集します．

```
def get_ramen_location(ramen: str, variable: str, context: Dict[str, Any]) -> None:
    location:str = ramen_map.get(ramen, "日本")
    context[variable] = location
```

上記のように，シナリオで使われている引数にプラスして，文脈情報を受け取る dict 型の変数を必ず加える必要があります．

シナリオで使われている引数はすべて文字列でなくてはなりません．

引数には，特殊変数・変数の場合，その値が渡されます．

また，変数参照の場合は'&' を除いた変数名が，定数の場合は，""の中の文字列が渡されます．

context は対話の最初に以下のキーと値のペアがセットされています．

キー	値
_state	state id
_config	config ファイルを読み込んでできた dict 型のデータ
_block_config	config ファイルのうち対話管理ブロックの設定部分（dict 型のデータ）