DialBB ドキュメント

リリース ***v0.2.0***

2022 年 12 月 02 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 開発の目的については、SLUD 研究会の論文に書きましたので、そちらも合わせてご参照ください。

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.

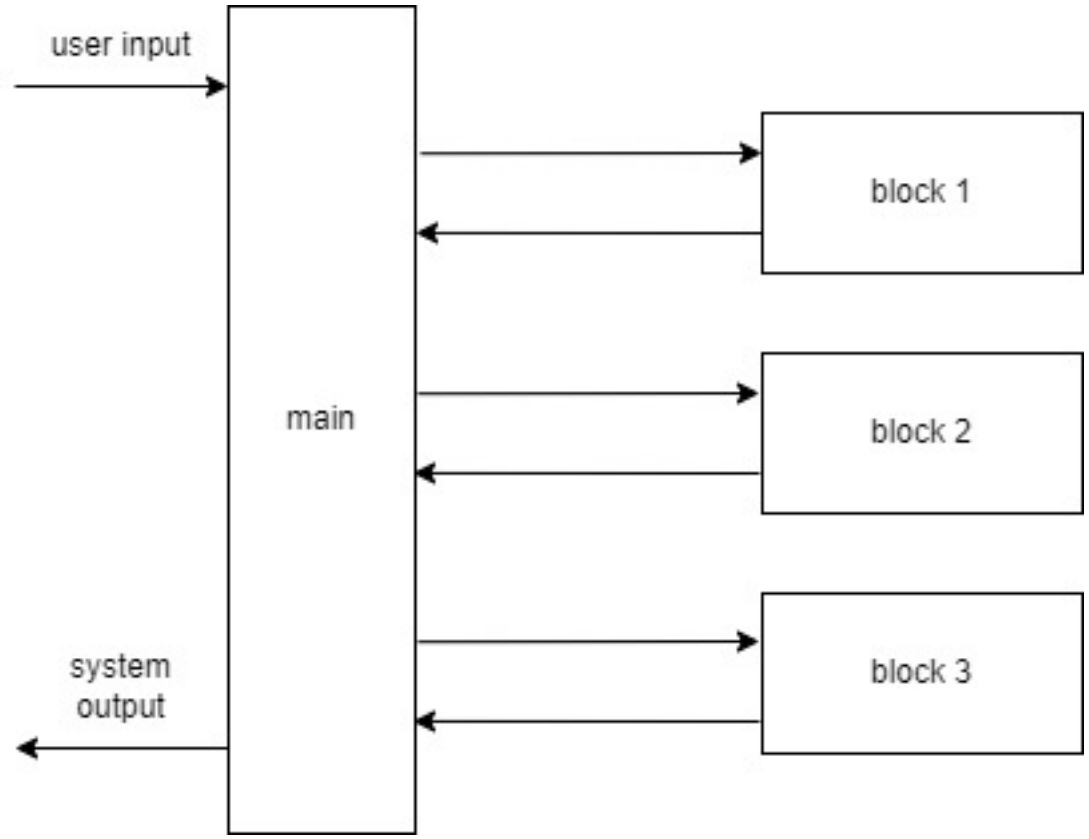
また，東中，稲葉，水上著：Python でつくる対話システムは，対話システムの実装の仕方について Python のコードを用いて説明しています.

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）を各ブロックに順次処理させることにより，システム発話を作成して返します．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，コンフィギュレーション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コンフィギュレーション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ケーションの実行の仕方

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 で，以下の手順で動作することを確認しています．

他のバージョンの Python でも動作する可能性が高いです．また，MacOS でも動作しますが，インストールに追加の手順が必要な場合があります．詳細は割愛します．

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 https://github.com/c4a-ri/dialbb.git
```

3.3 python 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 -m snips_nlu download en（英語アプリケーションを作成・利用する場合）
$ python -m snips_nlu download ja（日本語アプリケーションを作成・利用する場合）
```

- 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ケーションのサーバの起動

ただオウム返しを行うアプリケーションです．

```
$ 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．curl をインストールしていない場合は，「動作確認」に書いてある方法でテストしてください．

- 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":null,
 "session_id":"dialbb_session1",
 "system_utterance":"I'm a parrot. You can say anything.",
 "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,
 "final":false,
 "session_id":"dialbb_session1",
 "system_utterance":"You said \"こんにちは\"",
 "user_id":"user1"}
```

3.6 組み込みブロックを用いたサンプルアプリケーションの起動

DialBB には，あらかじめ作成してあるブロック（組み込みブロック）を用いたサンプルアプリケーションがあります．

3.6.1 起動

以下のコマンドで起動します

- 英語アプリケーション

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

- 日本語アプリケーション

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

3.6.2 動作確認

上記でアプリケーションを起動したサーバのホスト名か IP アドレスを<hostname>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.txt --output \
  sample_apps/network_en/_test_outputs.txt
```

☒ sample_apps/network_en/_test_outputs.txt に対話のやりとりが書き込まれます．

- 日本語

```
$ python dialbb/util/test.py sample_apps/network_ja/config.yml \
sample_apps/network_ja/test_inputs.txt --output \
sample_apps/network_ja/_test_outputs.txt
```

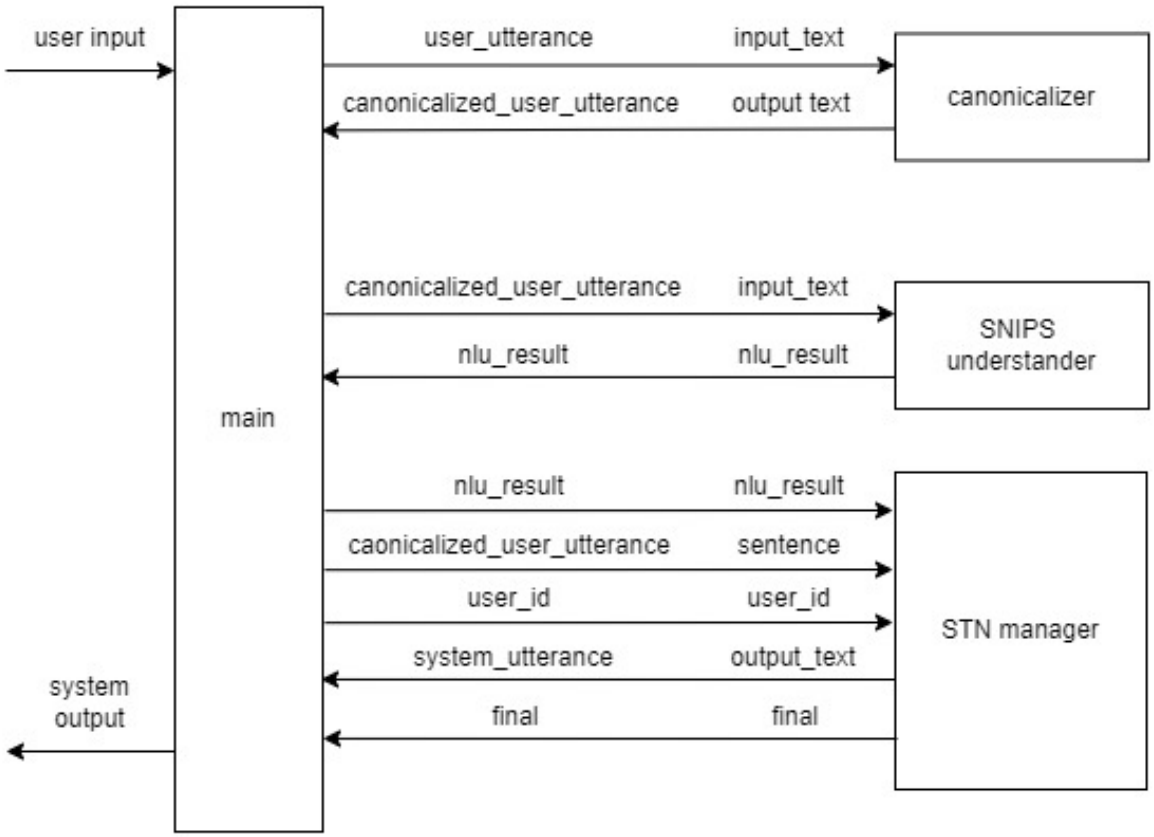
☒ sample_apps/network_ja/_test_outputs.txt に対話のやりとりが書き込まれます.

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，日本語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．

4.1 システム構成

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．



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: 正規化ブロック．ユーザ入力文の正規化（大文字 → 小文字，全角⇒半角の変換，Unicode 正規化など）を行います．
- SNIPS understander: 言語理解ブロック．形態素解析を行った後，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: 対話管理ブロック．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 で説明します．

4.2 アプリケーションを構成するファイル

本アプリケーションを構成するファイルは sample_apps/network_ja ディレクトリ（フォルダ）にあります．ここにあるファイルを変更することで，どのようにすればアプリケーションを変更することができるかを知ることができます．ファイルを大幅に変更すれば全く異なる対話システムも作ることができます．

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

- config.yml

  アプリケーションを規定するコンフィギュレーションファイルです．どのようなブロックを使うかや，各ブロックが読み込むファイルなどが指定されています．このファイルのフォーマットはコンフィギュレーションで詳細に説明します

- config_gs_template.yml

  config.yml は言語理解と対話管理の知識を Excel で書いたものを使うようになっていますが，Excel の代わりに Google Spreadsheet を用いる場合のコンフィギュレーションファイルのテンプレートです．config_gs.yml にコピーし，Google Spreadsheet にアクセスするための情報を加えることで使用できます．

- 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です．

- scenario_functions.py

  対話管理ブロックで用いるプログラムです．

- dictionary_functions.py

  言語理解用の辞書を Excel 記述ではなく，関数によって定義する場合の例が含まれています．

- test_inputs.txt

  システムテストで使うテストシナリオです．

4.3 言語理解ブロック

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 対話管理ブロック

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 の辞書型）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 [--port <port>] <config file>
```

port（ポート番号）のデフォルトは 8080 です．

クライアントからの接続（セッションの開始時）

- URI

  ```
  http://<server>:<port>/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．

クライアントからの接続（セッション開始後）

- URI

```
http://<server>:<port>/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(<コンフィギュレーションファイル> <追加のコンフィギュレーション>)
>>> session_id = <新しく作成した session id>            # 対話開始時
>>> session_id = <クライアントから送られ来た session id> # 2 回目以降のターン
>>> response = dialogue_processor.process(<リクエスト>, session_id)
```

  注意: process メソッドの仕様は v0.2.0 で変更になりました．

  <追加のコンフィギュレーション>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，コンフィギュレーション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，コンフィギュレーションファイルと追加のコンフィギュレーションで同じ key が用いられていた場合，追加のコンフィギュレーション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 コンフィギュレーション

コンフィギュレーション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．

コンフィギュレーションに必ず必要なのは 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（これをブロックコンフィギュレーションと呼びます）のリストで，以下のような形をしています．

```
blocks
  - <ブロックコンフィギュレーション>
  - <ブロックコンフィギュレーション>
  ...
  - <ブロックコンフィギュレーション>
```

各ブロックコンフィギュレーション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．モジュールを検索するパス（`sys.path` の要素の一つ．環境変数 `PYTHONPATH` で設定するパスはこれに含まれます）からの相対で記述します（組み込みクラスの場合 `dialbb.builtin_blocks` からの相対パスでも書けますが推奨されません）．コンフィギュレーション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 の子孫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．以下のように定義します．

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

    super().__init__(*args)

    <このブロック独自の処理>
```

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

  入力 input を処理し，出力を返します．入力，出力とメインモジュールの payload の関係はコンフィギュレーションで規定されます．(「コンフィギュレーション」を参照) session_id はメインモジュールから渡される文字列で，対話のセッション毎にユニークなものです．

5.4.2 利用できる変数

- self.config

  コンフィギュレーション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

  ブロックコンフィギュレーション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．(文字列)

- `self.config_dir`

  コンフィギュレーションファイルのあるディレクトリです．アプリケーションディレクトリと呼ぶこともあります．

5.4.3 利用できるメソッド

以下のロギン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この値を参照することができます．

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

5.6 テストシナリオを用いたテスト

以下のコマンドでテストシナリオを用いたテストができます．

```
$ python dialbb/util/test.py <アプリケーションコンフィギュレーション> \
  <テストシナリオ> [--output <出力ファイル>]
```

テストシナリオは以下の形式のテキストファイルです．

```
<対話の区切り>
<System: <システム発話>
User: <ユーザ発話>
System: <システム発話>
User: <ユーザ発話>
...
System: <システム発話>
User: <ユーザ発話>
System: <システム発話>
<対話の区切り>
<System: <システム発話>
User: <ユーザ発話>
System: <システム発話>
User: <ユーザ発話>
...
System: <システム発話>
User: <ユーザ発話>
System: <システム発話>
<対話の区切り>
...
```

<対話の区切り>は，"----init"で始まる文字列です．

テストスクリプトは，<ユーザ発話>を順番にアプリケーションに入力して，システム発話を受け取ります．システム発話がスクリプトのシステム発話と異なる場合は warning を出します．テストが終了すると，出力されたシステム発話を含め，テストシナリオと同じ形式で対話を出力することができます．テストシナリオと出力ファイルを比較することで，応答の変化を調べることができます．

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 （文字列正規化ブロック）

(dialbb.builtin_blocks.preprocess.utterance_canonicalizer.UtteranceCanonicalizer)

ユーザ入力文の正規化を行います．

コンフィギュレーションの 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．

6.1.1 入出力

- 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"

6.1.2 処理内容

正規化ブロックはユーザ発話文字列に対して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）
- Unicode 正規化（NFKC）（日本語のみ）

6.2 SNIPS understander （SNIPS を用いた言語理解ブロック）

(dialbb.builtin_blocks.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．

コンフィギュレーションの 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．日本語の場合は、SNIPS を適用する前に Sudachi を用いて形態素解析を行います。

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．

6.2.1 入出力

- 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（辞書型または辞書型のリスト）
    - 後述のブロックコンフィギュレーションのパラメータ num_candidates が 1 の場合、言語理解結果は辞書型で以下のような形式です。

```
{"type": <ユーザ発話タイプ（インテント）>,
 "slots": {<スロット名>: <スロット値>, ..., <スロット名>: <スロット値>}}
```

以下が例です。

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

    - num_candidates が 2 以上の場合、複数の理解結果候補のリストになります。

```
[{"type": <ユーザ発話タイプ（インテント）>,
  "slots": {<スロット名>: <スロット値>, ..., <スロット名>: <スロット値>}},
 {"type": <ユーザ発話タイプ（インテント）>,
  "slots": {<スロット名>: <スロット値>, ..., <スロット名>: <スロット値>}},
 ....]
```

6.2.2 ブロックコンフィギュレーションのパラメータ

- knowledge_file（文字列）

  知識を記述した Excel ファイルを指定します。コンフィギュレーションファイルのあるディレクトリからの相対パスで記述します。

- function_definitions（文字列）

  辞書関数（開発者による辞書関数の定義を参照）を定義したモジュールの名前です。複数ある場合は':' でつなぎます。モジュール検索パスにある必要があります。（コンフィギュレーションファイルのあるディレクトリはモジュール検索パスに入っています。）

- flags_to_use（文字列のリスト）

  各シートの flag カラムにこの値のうちのどれかが書かれていた場合に読み込みます。

- sudachi_normalization（ブール値）

  日本語の場合，この値が True の場合，言語理解モデルの構築時と言語理解実行時に Sudachi の正規化（表記揺れの吸収を含む）を行います.

- num_candidates（Integer）

  言語理解結果の数（n-best の n）を指定します。

- knowledge_google_sheet (ハッシュ)
  - Excel の代わりに Google Sheet を用いる場合の情報を記述します。
    - sheet_id （文字列）

      Google Sheet の ID です。

    - key_file（文字列）

      Goole Sheet API にアクセスするためのキーファイルをコンフィギュレーションファイルのディレクトリからの相対パスで指定します。

6.2.3 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

シート名はブロックコンフィギュレーションで変更可能ですが，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.

（注）各シートのカラム名が ver0.2.0 で変更されました.

utterances シート

各行は次のカラムからなります.

- 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です．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します．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ます.

- 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>] で表現します．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 シートの synonyms カラムにあるものの場合，スロット値は，dictionary シートの entity カラムに書かれたものになります.

utterances シートのみならずこのブロックで使うシートにこれ以外のカラムがあっても構いません.

slots シート

各行は次のカラムからなります.

- flag

  utterances シートと同じ

- slot name

  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ます.

- entity class

  エンティティクラス名．スロットの値がどのようなタイプの名詞句なのかを表します．異なるスロットが同じエンティティクラスを持つ場合があります．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 クラスのエンティティを取ります．entity class カラムの値として辞書関数（dialbb/<関数名>の形）を使うことができます．これにより，dictionary シートに辞書情報を記述する代わりに，関数呼び出しで辞書記述を得ることができます．(例: dialbb/location）関数は以下の「**開発者による辞書関数の定義**」で説明します．また entity class カラムの値は，SNIPS の builtin entity でも構いません．(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.

- `flag`

  utterances シートと同じ

- `entity class`

  エンティティクラス名．slots シートで辞書関数を指定した場合は，こでも同じように辞書関数名を書く必要があります.

- `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.

- `flag`

  utterances シートと同じ

- `entity class`

  エンティティクラス名

- `entity`

  辞書エントリー名．言語理解結果にも含まれます.

- `synonyms`

  同義語を，または , または，で連結したもの

開発者による辞書関数の定義

辞書関数は，主に外部のデータベースなどから辞書情報を取ってくるときに利用します.

辞書関数はブロックコンフィギュレーションの `dictionary_function` で指定するモジュールの中で定義します.

辞書関数は引数にコンフィギュレーションとブロックコンフィギュレーションを取ります．これらに外部データベースへの接続情報などが書いてあることを想定しています.

辞書関数の返り値は辞書型のリストで，{"value": <文字列>, "synonyms": <文字列のリスト>}の形の辞書型のリストです．"synonyms"キーはなくても構いません．

以下に辞書関数の例を示します．

```
def location(config: Dict[str, Any], block_config: Dict[str, Any]) \
    -> List[Dict[str, Union[str, List[str]]]]:
    return [{"value": "札幌", "synonyms": ["さっぽろ", "サッポロ"]},
            {"value": "荻窪", "synonyms": ["おぎくぼ"]},
            {"value": "徳島"}]
```

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ベースの言語理解ブロック）

(dialbb.builtin_blocks.stn_manager.stn_management)

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（辞書型または辞書型のリスト）
  - user_id:ユーザ ID（文字列）
  - aux_data 補助データ（辞書型）
- 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"}

6.3.1 ブロックコンフィギュレーションのパラメータ

- knowledge_file（文字列）

  シナリオを記述した Excel ファイルを指定します。コンフィギュレーションファイルのあるディレクトリからの相対パスで記述します。

- function_definitions（文字列）

  シナリオ関数（開発者による辞書関数の定義を参照）を定義したモジュールの名前です。複数ある場合は':' でつなぎます。モジュール検索パスにある必要があります。（コンフィギュレーションファイルのあるディレクトリはモジュール検索パスに入っています。）

- flags_to_use（文字列のリスト）

  各シートの flag カラムにこの値のうちのどれかが書かれていた場合に読み込みます。

- knowledge_google_sheet (ハッシュ)

  SNIPS Understander と同じです。

6.3.2 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.

- flag

  utterance シートと同じ

- state

  状態名

- 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ます．state が同じ行は複数あり得ますが，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ます.

- 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ません.

- 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．遷移の条件となります.

- conditions

  条件（の並び）．遷移の条件を表す関数呼び出し．複数あっても構いません．複数ある場合は，；で連結します．各条件は<関数名>(<引数 1>，<引数 2>，...，<引数 n>) の形をしています．引数は 0 個でも構いません.

- 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．複数あっても構いません．複数ある場合は，;で連結します．各条件は<関数名>(<引数 1>，<引数 2>，...，<引数 n>) の形をしています．引数は 0 個でも構いません．

- next state

  遷移先の状態名

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．

シートにこれ以外のカラムがあっても構いません．

6.3.3 特別な状態

以下の状態名はあらかじめ定義されています．

- #prep

  準備状態．初期状態の前にある状態です．対話が始まった時に初期状態に遷移します．この時，この遷移のアクションを実行します．最初のシステム発話を状況に応じて変更するときに使います．日本語サンプルアプリは，対話が行われる時間に応じて挨拶の内容を変更します．この準備状態はなくても構いません．

- #initial

  初期状態．対話はこの状態から始まります．

- #error

  内部エラーが起きたときこの状態に移動します．システム発話を生成して終了します．

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．

6.3.4 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（python の辞書型データ）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

条件やアクションで用いる関数の引数には次のタイプがあります.

- 特殊変数 （#で始まる文字列）

  以下の種類があります.

  - #<スロット名> 直前のユーザ発話の言語理解結果（入力の nlu_result の値）のスロット値．スロット値が空の場合は空文字列になります.
  - #<補助データのキー> 入力の aux_data の中のこのキーの値．例えば#emotion の場合，aux_data['emotion'] の値．このキーがない場合は，空文字列になります.
  - #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いといけません．文脈情報にその変数がない場合は空文字列になります.

- 変数参照（&で始まる文字列）

  &<文脈情報での変数の名前> の形で，関数定義内で文脈情報の変数名を利用するときに用います.

- 定数（""で囲んだ文字列）

  文字列そのままを意味します.

6.3.5 関数定義

条件やアクションで用いる関数は，DialBB 組み込みのものと，開発者が定義するものがあります．条件で使う関数は bool 値を返し，アクション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.

- 条件で用いる関数
  - _eq(x, y)

    x と y が同じなら True を返します．例：_eq(*a, "b"): 変数 a の値が"b"なら True を返します．_eq(#food, "ラーメン"): #food スロットが"ラーメン"なら True を返します.

  - _ne(x, y)

    x と y が同じでなければ True を返します.

    例：_ne(*a, *b): 変数 a の値と変数 b の値が異なれば True を返します．_ne(#food, "ラーメン"): #food スロットが"ラーメン"なら False を返します.

- _contains(x, y)

  x が文字列として y を含む場合 True を返します.
  例：_contains(#sentence, "はい") : ユーザ発話が「はい」を含めば True を返します.

- _not_contains(x, y)

  x が文字列として y を含まない場合 True を返します.

  例： _not_contains(#sentence, "はい") : ユーザ発話が"はい"を含めば True を返します.

- _member_of(x, y)

  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 を返します.

  例：_member_of(#food, "ラーメン:チャーハン:餃子")

- _not_member_of(x, y)

  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 を返します.

  例：_not_member_of(*favorite_food, "ラーメン:チャーハン:餃子")

- アクションで用いる関数

  - _set(x, y)

    変数 x に y をセットします.

    例：_set(&a, b): b の値を a にセットします. _set(&a, "hello")： a に"hello"をセットします.

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辞書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辞書型のデータ
_block_config	config ファイルのうち対話管理ブロックの設定部分（辞書型のデータ）

6.3.6 言語理解結果候補が複数ある場合の処理

入力の nlu_result が辞書型のリストで、複数の言語理解結果候補を含んでいる場合の処理は次のようになります。

リストの先頭から順に、言語理解結果候補の type の値が、現在の状態から可能な遷移のうちのどれかの user utterance type の値に等しいかどうかを調べ、等しい遷移があれば、その言語理解結果候補を用います。

どの言語理解結果候補も上記の条件に合わない場合、リストの先頭の言語理解結果候補を用います。