DialBB 0.5 ドキュメント

リリース *v0.6.0*

2023 年 11 月 24 日

Contents:

第1章　はじめに

DialBB (*Dial*ogue System Development Framework with *B*uilding *B*locks) 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，人工知能学会 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.

DialBB のインストールの仕方とサンプルアプリケーションの動かし方は README を見てください.

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「対話システムを知ろう」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.

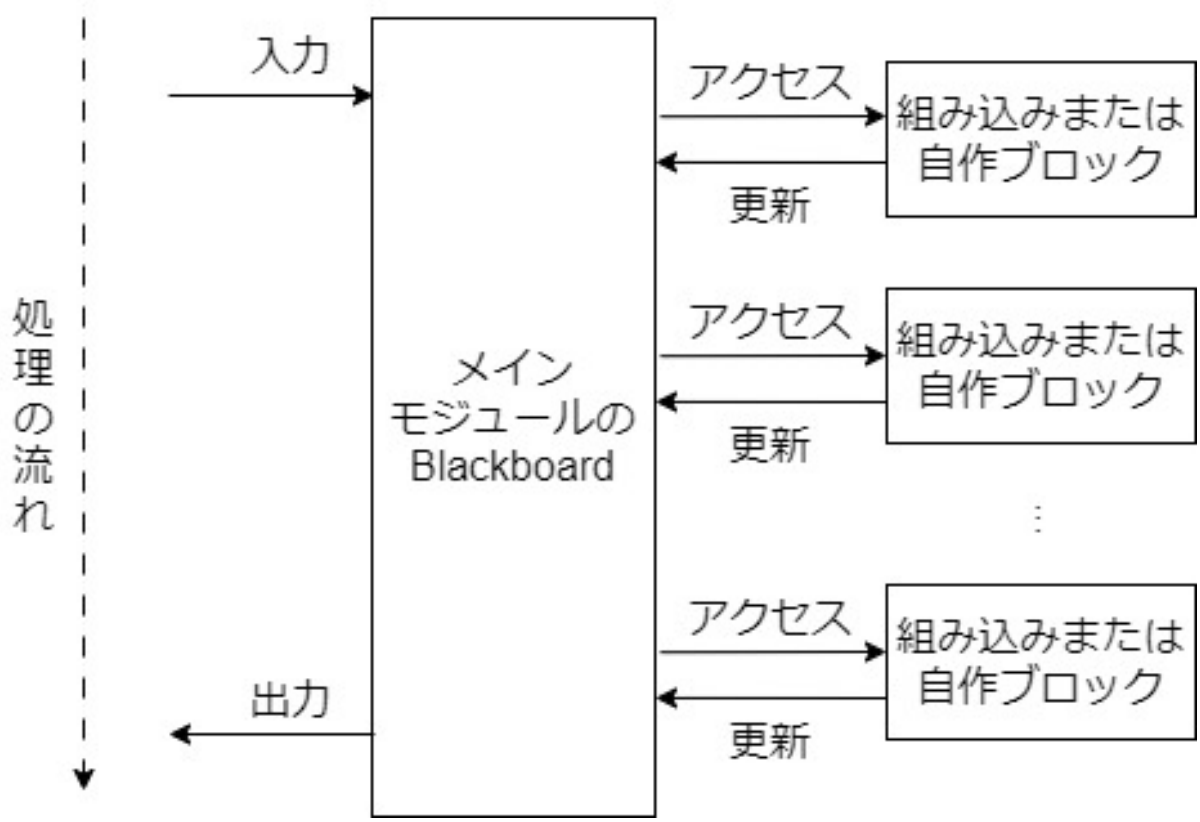
DialBB は株式会社 C4A 研究所が開発し，著作権を保有し，非商用向けに公開しています．詳しくはライセンスを参照ください.

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．

以下に DialBB のアプリケーションの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 blackboard[1] と呼びます．各ブロックは，blackboard 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 blackboard に追加されます．すでに同じキーを持つ要素が blackboard 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．

[1] ver.0.2 以前は payload と呼んでいました．

詳細は「**フレームワーク仕様**」で説明します.

第3章　サンプルアプリケーション

3.1 DialBB 付属のサンプルアプリケーション

DialBB には以下のサンプルアプリケーションが付属しています。

3.1.1 オウム返しサンプルアプリケーション

ただオウム返しを行うアプリケーションです．組み込みブロッククラスは使っていません．

`sample_apps/parrot` にあります。

3.1.2 Snips+STN アプリケーション

以下の組み込みブロックを用いたサンプルアプリケーションです．

- 日本語アプリケーション
  - *Japanese canonicalizer* （日本語文字列正規化ブロック）
  - *Sudachi tokenizer* （*Sudachi* ベースの日本語単語分割ブロック）
  - *SNIPS understander* （*SNIPS* を用いた言語理解ブロック）
  - *STN manager* （状態遷移ネットワークベースの対話管理ブロック）
- 英語アプリケーション
  - *Simple canonicalizer* （単純文字列正規化ブロック）
  - *Whitespace tokenizer* （空白ベースの単語分割ブロック）
  - *SNIPS understander* （*SNIPS* を用いた言語理解ブロック）
  - *STN manager* （状態遷移ネットワークベースの対話管理ブロック）

`sample_apps/network_ja/`に日本語版が，`sample_apps/network_en/`に英語があります．

3.1.3 実験アプリケーション

Snips+ネットワークベース対話管理アプリケーションをもとに、組み込みブロックの様々な機能を試すためのアプリケーションです（日本語のみ）以下の組み込みブロックを用いています。

- *Japanese canonicalizer* **（日本語文字列正規化ブロック）**
- *Sudachi tokenizer* **（***Sudachi* **ベースの日本語単語分割ブロック）**
- *SNIPS understander* **（***SNIPS* **を用いた言語理解ブロック）**
- *spaCy-Based Named Entity Recognizer* **（***spaCy* **を用いた固有表現抽出ブロック）**
- *STN manager* **（状態遷移ネットワークベースの対話管理ブロック）**

`sample_apps/lab_app_ja/`にあります。

3.1.4 ChatGPT 対話アプリケーション

(ver. 0.6 で追加）以下の組み込みブロックを用い，OpenAI の ChatGPT を用いて対話を行います.

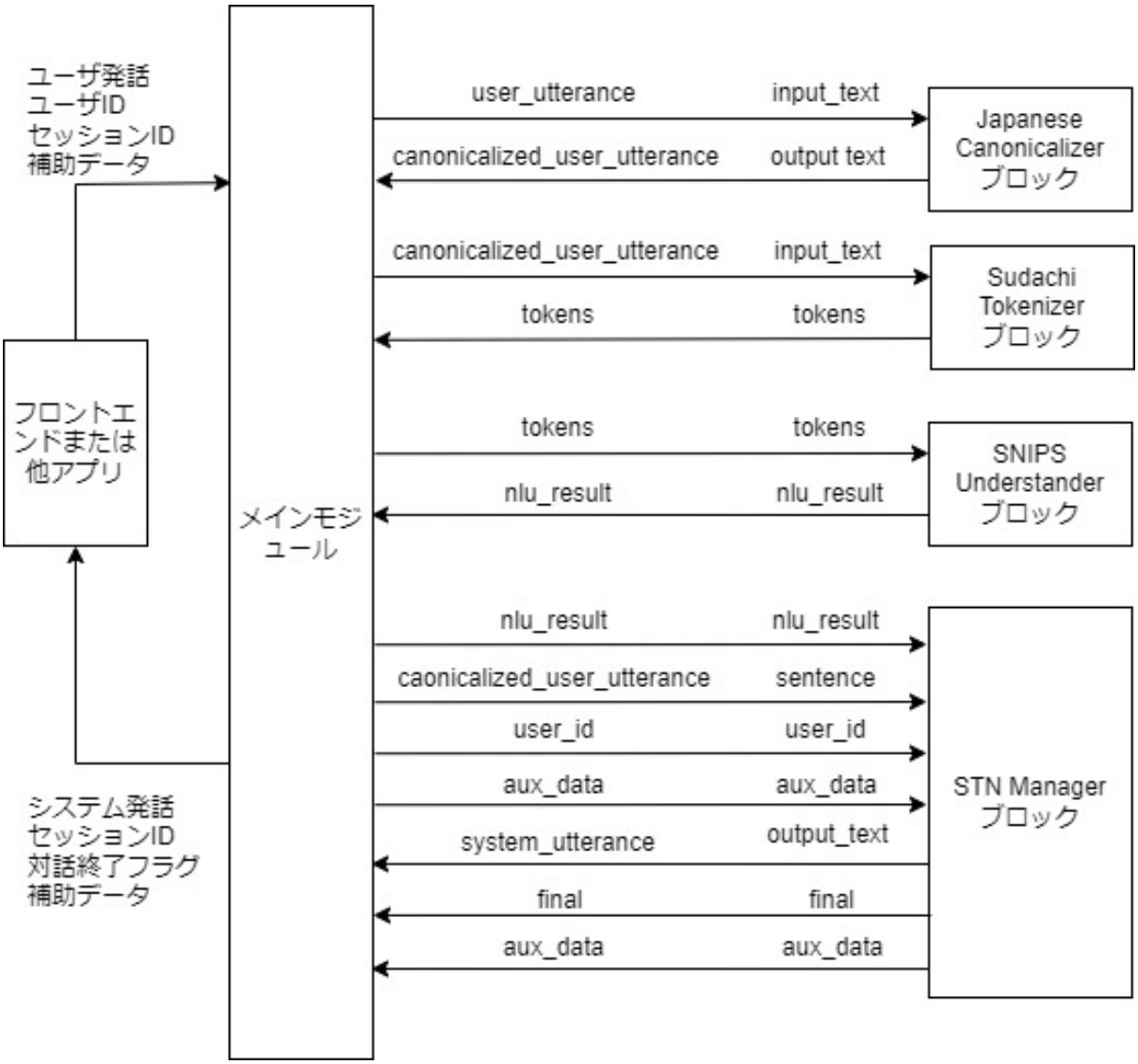
- *ChatGPT Dialogue* **（***ChatGPT* **ベースの対話ブロック）**

3.2 Snips + STN アプリケーションの説明

以下では，SNIPS+STN 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.

3.2.1 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，以下のつの組み込みブロックを利用しています．なお，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，「**組み込みブロッククラスの仕様**」で説明します．

- Japanese Canonicalizer: 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．
- Sudachi Tokenizer: 正規化されたユーザ入力文を単語に分割します．形態素解析器 Sudachi を用います．
- 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: 対話管理と言語生成を行います．状態遷移ネットワーク (State Transition Network) を用いて対話管理を行い，システム発話を出力します．

メインモジュールとブロックを結ぶ矢印の上の記号は，左側がメインモジュールの blackboard におけるキーで，右側がブロックの入出力におけるキーです．

3.2.2 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

  SNIPS Understander ブロックと STN Manage ブロックで用いる知識を Excel ではなく，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．これをコピーし，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

  SNIPS Undderstander ブロックと STN Manager ブロックで用いる知識を記述したものです．

- scenario_functions.py

  STN Manager ブロックで用いるプログラムです．

- dictionary_functions.py

  SNIPS Undderstander 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．

3.2.3 SNIPS Understander ブロック

言語理解結果

SNIPS Understander ブロックは，入力発話を解析し，言語理解結果を出力します．言語理解結果はタイプとスロットの集合からなります．

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．

言語理解知識

SNIPS Understander ブロックが用いる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．

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．

3.2.4 STN Manager 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4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．

4.1 入出力

DialBB のメインモジュールは，クラス API（メソッド呼び出し）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．

4.1.1 DialogueProcessor クラス

アプリケーションの作成は，`dialbb.main.DialogueProcessor` クラスのオブジェクトを作成することで行います．

これは以下の手順で行います．

- 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
```

- 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[1]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>)
>>> response = dialogue_processor.process(<リクエスト>, initial=True)  # 対話セッション開始時
>>> response = dialogue_processor.process(<リクエスト>) # セッション継続時
```

<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．

[1] process メソッドの仕様は v0.2.0 で変更になりました．

```
{
  "<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，以下で説明します．

DialogueProcessor.process は**スレッドセーフではありません．**

4.1.2 リクエスト

セッション開始時

以下の形の JSON です．

```
{
  "user_id": <ユーザ ID: 文字列>,
  "aux_data": <補助データ: オブジェクト（値の型は任意）>
}
```

- 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．JSON オブジェクトで，内容はアプリ毎に決めます．

セッション開始後

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: オブジェクト (値の型は任意>}
```

- 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．

4.1.3 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: オブジェクト（値の型は任意>
}
```

- <セッション ID>は，対話のセッションの ID です．対話開始のリクエストを送信した際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．

4.2 WebAPI

アプリケーションに WebAPI 経由でアクセスすることもできます．

4.2.1 サーバの起動

環境変数 PYTHONPATH を設定します．

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

コンフィギュレーションファイルを指定してサーバを起動します．

```
$ python <DialBB のディレクトリ>/run_server.py [--port <port>] <config file>
```

port（ポート番号）のデフォルトは 8080 です．

4.2.2 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

  クラス API の場合のリクエストと同じ JSON 形式のデータです.

- レスポンス

  クラス API の場合のレスポンスと同じ JSON 形式のデータです.

4.2.3 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

  クラス API の場合のリクエストと同じ JSON 形式のデータです.

- レスポンス

  クラス API の場合のレスポンスと同じ JSON 形式のデータです.

4.3 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.

  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.

  組み込みクラスは, dialbb.builtin_blocks.<モジュール名>.<クラス名>の形で指定してください. dialbb.builtin_blocks からの相対パスでも書けますが, 非推奨です.

- 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 blackboard(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 blackboard['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 blackboard 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,

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

  の処理が行われます. blackboard 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.

4.4 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.

4.4.1 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 blackboard 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.

4.4.2 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

  コンフィギュレーションに書いてある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.

4.4.3 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.

4.5 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.

4.6 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>
```

(次のページに続く)

(前のページからの続き)

```
<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5章　組み込みブロッククラスの仕様

組み込みブロッククラスとは，DialBB にあらかじめ含まれているブロッククラスです．

ver0.3 で正規化ブロッククラスが変更になりました．また，新たに単語分割のブロッククラスが導入されました．それに伴い，SNIPS 言語理解の入力も変更になっています．

5.1 Japanese canonicalizer （日本語文字列正規化ブロック）

(`dialbb.builtin_blocks.preprocess.japanese_canonicalizer.JapaneseCanonicalizer`)

入力文字列の正規化を行います．

5.1.1 入出力

- 入力
  - `input_text`: 入力文字列（文字列）
    - 例："ＣＵＰ Noodle 好き"
- 出力
  - `output_text`: 正規化後の文字列（文字列）
    - 例："cupnoodle 好き"

5.1.2 処理内容

入力文字列に対して以下の処理を行います．

- 前後のスペースの削除
- 英大文字 → 英小文字
- 改行の削除
- 全角 → 半角の変換（カタカナを除く）
- スペースの削除
- Unicode 正規化（NFKC）

5.2 Simple canonicalizer （単純文字列正規化ブロック）

(dialbb.builtin_blocks.preprocess.simple_canonicalizer.SimpleCanonicalizer)

ユーザ入力文の正規化を行います．主に英語が対象です．

5.2.1 入出力

- 入力
  - input_text: 入力文字列（文字列）
    - 例：" I like ramen"
- 出力
  - output_text: 正規化後の文字列（文字列）
    - 例："i like ramen"

5.2.2 処理内容

入力文字列に対して以下の処理を行います．

- 前後のスペースの削除
- 英大文字 → 英小文字
- 改行の削除
- スペースの連続を一つのスペースに変換

5.3 Sudachi tokenizer （Sudachi ベースの日本語単語分割ブロック）

(dialbb.builtin_blocks.tokenization.sudachi_tokenizer.SudachiTokenizer)

Sudachi を用いて入力文字列を単語に分割します．

5.3.1 入出力

- 入力
  - input_text: 入力文字列（文字列）
    - 例："私はラーメンが食べたい"
- 出力
  - tokens: トークンのリスト（文字列のリスト）

    - 例：[' 私',' は',' ラーメン',' が',' 食べ',' たい']

  - tokens_with_indices: トークン情報のリスト（dialbb.tokenization.abstract_tokenizer.TokenWIthIndices クラスのオブジェクトのリスト）．トークン情報には，トークンに加えてそのトークンが元の文字列の何文字目から何文字目までなのかの情報も含まれています．

5.3.2 処理内容

Sudachi の SplitMode.C を用いて単語分割します．

ブロックコンフィギュレーションの sudachi_normalization の値が True の時，Sudachi 正規化を行います．デフォルト値は False です．

5.4 Whitespace tokenizer （空白ベースの単語分割ブロック）

(dialbb.builtin_blocks.tokenization.whitespace_tokenizer.WhitespaceTokenizer)

入力を空白で区切って単語に分割します．主に英語が対象です．

5.4.1 入出力

- 入力
  - input_text: 入力文字列（文字列）
    - 例："i like ramen"
- 出力
  - tokens: トークンのリスト（文字列のリスト）
    - 例：['i','like','ramen']
  - tokens_with_indices: トークン情報のリスト（dialbb.tokenization.abstract_tokenizer.TokenWIthIndices クラスのオブジェクトのリスト）．トークン情報には，トークンに加えてそのトークンの開始位置が元の文字列の何文字目から何文字目までなのかの情報も含まれています．

5.4.2 処理内容

単純正規化ブロックで正規化された入力を空白で区切って単語に分割します.

5.5 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.

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.

5.5.1 入出力

- 入力
  - tokens: トークンのリスト（文字列のリスト）
    - 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": {"favorite_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>}},
 ....]
```

5.5.2 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．このパラメータがセットされていない場合はすべての行が読み込まれます．

- canonicalizer

  言語理解知識を SNIPS の訓練データに変換する際に行う正規化の情報を指定します．

  - class

    正規化のブロックのクラスを指定します．基本的にアプリケーションで用いる正規化のブロックと同じものを指定します．

- tokenizer

  言語理解知識を SNIPS の訓練データに変換する際に行う単語分割の情報を指定します．

  - class

    単語分割のブロックのクラスを指定します．基本的にアプリケーションで用いる単語分割のブロックと同じものを指定します．

  - sudachi_normalization（ブール値．デフォルト値 False）

    単語分割に Sudachi Tokenizer を用いる場合，この値が True の時には，Sudachi 正規化を行います．

- num_candidates（Integer. デフォルト値 1）

  言語理解結果の最大数（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．（Google Sheet を利用する際の設定はこはたさんの記事が参考になりますが，Google Cloud Platform の設定画面の UI がこの記事とは多少変わっています．）

    * 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.

5.5.3 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.

5.6 STN manager（状態遷移ネットワークベースの対話管理ブロック）

(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）（必須ではありませんが指定することが推奨されます）
- 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）(ver. 0.4.0 で変更）入力の補助データを，後述のアクション関数の中でアップデートしたものに，遷移した状態の ID を含めたもの．アクション関数の中でのアップデートは必ずしも行われるわけではない．遷移した状態は，以下の形式で付加される．

    ```
    {"state": "特定のラーメンが好き"}
    ```

5.6.1 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.

- scenario_graph: (ブール値．デフォルト値 False）

この値が True の場合，シナリオシートの system utterance カラムと user utterance example カラムの値を使って，グラフを作成します．これにより，シナリオ作成者が直感的に状態遷移ネットワークを確認できます．

- repeat_when_no_available_transitions （ブール値．デフォルト値 False．ver. 0.4.0 で追加）

  この値が True のとき，デフォルト遷移（後述）以外の遷移で条件に合う遷移がないとき，遷移せず同じ発話を繰り返します．

5.6.2 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．

このシートの各行が，一つの遷移を表します．各行は次のカラムからなります．

- flag

  utterance シートと同じ

- state

  遷移元の状態名

- 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．各条件で使える引数については，**関数の引数**を参照してください．

- 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．各条件で使える引数については，**関数の引数**を参照してください．

- next state

  遷移先の状態名

（メモとして利用するために）シートにこれ以外のカラムがあっても構いません．

各行が表す遷移の 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 の状態に遷移します．その際，actions に書いてあるアクションが実行されます．

state カラムが同じ行（遷移元の状態が同じ遷移）は，**上に書いてあるものから順に**遷移の条件を満たしているかどうかをチェックします．

デフォルト遷移（user utterance type カラムも conditions カラムも空の行）は，state カラムが同じ行の中で一番下に書かれていなくてはなりません．

5.6.3 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．この状態がある場合，対話が始まった時（クライアントから最初にアクセスがあった時）に，この状態からの遷移が試みられます．state カラムの値が#prep の行の conditions にある条件がすべて満たされるかどうかを調べ，満たされた場合に，その行の actions のアクションを実行してから，next state の状態に遷移し，その状態のシステム発話が出力されます．

  最初のシステム発話や状態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．

  #prep からの遷移先は#initial でなくてもよくなりました．(ver. 0.4.0)

- #initial

  初期状態．#prep 状態がない場合，対話が始まった時（クライアントから最初にアクセスがあった時）この状態から始まり，この状態のシステム発話が output_text に入れられてメインプロセスに返されます．

#prep 状態または#initial 状態のどちらかがなくてはなりません．

- #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．

5.6.4 条件とアクション

文脈情報

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．

文脈情報にはあらかじめ以下のキーと値のペアがセットされています．

キー	値
_current_state_name	遷移前状態の名前（文字列）
_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）
_aux_data	メインプロセスから受け取った aux_data（辞書型のデータ）
_previous_system_utterance	直前のシステム発話（文字列）
_dialogue_history	対話履歴（リスト）

対話履歴は，以下の形です．

```
[
  {"speaker": "user",
   utterance": <正規化後のユーザ発話 (文字列)>},
  {"speaker": "system",
   utterance": <システム発話>},
  {"speaker": "user",
   utterance": <正規化後のユーザ発話 (文字列)>},
  {"speaker": "system",
   utterance": <システム発話>},
  ...
]
```

これらに加えて新しいキーと値のペアをアクション関数内で追加することができます．

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．

5.6.5 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.

  - _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.

5.6.6 連続遷移

システム発話（の 1 番目）が$skip である状態に遷移した場合，システム応答を返さず，即座に次の遷移を行います. これは，最初の遷移のアクションの結果に応じて二つ目の遷移を選択するような場合に用います.

5.6.7 言語理解結果候補が複数ある場合の処理

入力の nlu_result がリスト型のデータで，複数の言語理解結果候補を含んでいる場合，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．

5.6.8 リアクション

アクション関数の中で，文脈情報の_reaction に文字列をセットすると，状態遷移後のシステム発話の先頭に，その文字列を付加します．

例えば，_set(&_reaction, "そうですね") というアクション関数を実行した後に遷移した状態のシステム発話が"ところで今日はいい天気ですね"であれば，"そうですね ところで今日はいい天気ですね"という発話をシステム発話として返します．

5.6.9 Subdialogue

遷移先の状態名が#gosub:<状態名 1>:<状態名 2>の形の場合，<状態名 1>の状態に遷移して，そこから始まる subdialogue を実行します．そして，その後の対話で，遷移先が:exit になったら，<状態名 2>の状態に移ります．

例えば, 遷移先の状態名が#gosub:request_confirmation:confirmed の形の場合, request_confirmatin から始まる subdialogue を実行し，遷移先が:exit になったら，confirmed に戻ります．

subdialogue の中で subdialogue に遷移することも可能です．

5.6.10 音声入力を扱うための仕組み

ver. 0.4.0 で，音声認識結果を入力として扱うときに生じる問題に対処するため，以下の変更が行われました．

ブロックコンフィギュレーションパラメータの追加

- input_confidence_threshold （float．デフォルト値 0.0）

  入力が音声認識結果の時，その確信度がこの値未満の場合に，確信度が低いとみなします．入力の確信度は，aux_data の confidence の値です．aux_data に confidence キーがないときは，確信度が高いとみなします．確信度が低い場合は，以下に述べるパラメータの値に応じて処理が変わります．

- confirmation_request（オブジェクト）

  これは以下の形で指定します.

```
confirmation_request:
  function_to_generate_utterance: <関数名（文字列）>
  acknowledgement_utterance_type: <肯定のユーザ発話タイプ名（文字列）>
  denial_utterance_type: <否定のユーザ発話タイプ名（文字列）>
```

  これが指定されている場合，入力の確信度が低いときは，状態遷移をおこなわず，function_to_generate_utterance で指定された関数を実行し，その返り値を発話します（確認要求発話と呼びます）.

  そして，それに対するユーザ発話に応じて次の処理を行います.

  - ユーザ発話の確信度が低い時は，遷移を行わず，前の状態の発話を繰り返します.
  - ユーザ発話のタイプが acknowledgement_utterance_type で指定されているものの場合，確認要求発話の前のユーザ発話に応じた遷移を行います.
  - ユーザ発話のタイプが denial_utterance_type で指定されているものの場合，遷移を行わず，元の状態の発話を繰り返します.
  - ユーザ発話のタイプがそれ以外の場合は，通常の遷移を行います.

  ただし，入力がバージイン発話の場合（aux_data に barge_in 要素があり，その値が True の場合）はこの処理を行いません.

  function_to_generate_utterance で指定する関数は，ブロックコンフィギュレーションの function_definitions で指定したモジュールで定義します．この関数の引数は，このブロックの入力の nlu_result と文脈情報です．返り値はシステム発話の文字列です.

- utterance_to_ask_repetition（文字列）

  これが指定されている場合，入力の確信度が低いときは，状態遷移をおこなわず，この要素の値をシステム発話とします．ただし，バージインの場合（aux_data に barge_in 要素があり，その値が True の場合）はこの処理を行いません.

  confirmation_request と utterance_to_ask_repetition は同時に指定できません.

- ignore_out_of_context_barge_in （ブール値．デフォルト値 False）

  この値が True の場合，入力がバージイン発話 (リクエストの aux_data の barge_in の値が True) の場合，デフォルト遷移以外の遷移の条件を満たさなかった場合（すなわちシナリオで予想された入力ではない）か，または，入力の確信度が低い場合，遷移しません．この時に，レスポンスの aux_data の barge_in_ignored を True とします.

- reaction_to_silence （オブジェクト）

  action 要素を持ちます．action キーの値は文字列で repeat か transition です．action 要素の値が transition の場合，action キーが必須です．その値は文字列です.

  入力の aux_data が long_silence キーを持ちその値が True の場合で，かつ，デフォルト遷移以外の遷移の条件を満たさなかった場合，このパラメータに応じて以下のように動作します.

- このパラメータが指定されていない場合，通常の状態遷移を行います．
- action の値が"repeat"の場合，状態遷移を行わず直前のシステム発話を繰り返します．
- action の値が transition の場合，destination で指定されている状態に遷移します．

組み込み条件関数の追加

以下の組み込み条件関数が追加されています．

- _confidence_is_low()

  入力の aux_data の confidence の値がコンフィギュレーションの input_confidence_threshold の値以下の場合に True を返します．

- _is_long_silence()

  入力の aux_data の long_silence の値が True の場合に True を返します．

直前の誤った入力を無視する

入力の aux_data の rewind の値が True の場合，直前のレスポンスを行う前の状態から遷移を行います．直前のレスポンスを行った際に実行したアクションによる文脈情報の変更も元に戻されます．

音声認識の際に，ユーザ発話を間違って途中で分割してしまい，前半だけに対する応答を行ってしまった場合に用います．

文脈情報は元に戻りますが，アクション関数の中でグローバル変数の値を変更していたり，外部データベースの内容を変更していた場合にはもとに戻らないことに注意してください．

5.7 ChatGPT Dialogue （ChatGPT ベースの対話ブロック）

(ver0.6 で追加)

(dialbb.builtin_blocks.chatgpt.chatgpt_ja.ChatGPT_Ja, （日本語用）dialbb.builtin_blocks.chatgpt.chatgpt_ja.ChatGPT_En（英語用）)

OpenAI 社の ChatGPT を用いて対話を行います．

これらのクラスは dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT のサブクラスです．dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT のサブクラスを新たに作ることで，ChatGPT を使った新たなブロックを作ることができます．

5.7.1 入出力

- 入力
  - user_utterance: 入力文字列（文字列）
  - aux_data: 補助データ（辞書型）
  - user_id: 補助データ（辞書型）
- 出力
  - system_utterance: 入力文字列（文字列）
  - aux_data: 補助データ（辞書型）
  - final: 対話終了かどうかのフラグ（ブール値）

ChatGPT_Ja, ChatGPT_En では，入力の aux_data, user_id 利用せず，出力の aux_data は入力の aux_data と同じもので，final は常に False ですが，ChatGPT のサブクラスを新しく作る時に変更することができます．

これらのブロックを使う時には，環境変数 OPENAI_KEY に OpenAI のライセンスキーを設定する必要があります．

5.7.2 ブロックコンフィギュレーションのパラメータ

- first_system_utterance （文字列，デフォルト値は""）

  対話の最初のシステム発話です．
- prompt_prefix （文字列，デフォルト値は""）
- prompt_postfix （文字列，デフォルト値は""）

  ChatGPT のプロンプトに使う文字列です．以下で説明します．
- gpt_model （文字列，デフォルト値は gpt-3.5-turbo）

  Open AI GPT のモデルです．gpt-4 などが指定できます．

5.7.3 処理内容

- 対話の最初はブロックコンフィギュレーションの first_system_utterance の値をシステム発話として返します．
- 2 回目以降のターンでは，以下のプロンプトを ChatGPT に与え，返ってきたものをシステム発話として返します．

  日本語の場合

```
<prompt_prefix の値>
システム「<システムの 1 番目の発話>」
ユーザ「<ユーザの 1 番目の発話>」
システム「<システムの 2 番目の発話>」
ユーザ「<ユーザの 2 番目の発話>」
...
システム「<システムの最新の発話>」
ユーザ「<ユーザの最新の発話>」
<prompt_postfix の値>
```

英語の場合

```
<prompt_prefix の値>
System: "<システムの 1 番目の発話>"
User: "<ユーザの 1 番目の発話>"
System: "<システムの 2 番目の発話>"
User: "<ユーザの 2 番目の発話>"
...
System: "<システムの最新の発話>"
User: "<ユーザの最新の発話>"
<prompt_postfix の値>
```

例として以下のようなプロンプトが与えられます．

```
システムはユーザと親しく話せます．システム「こんにちは．楽しくお話しましょう．何についてお話しますか？」
ユーザー「映画について話したいです」
システム「素晴らしいですね．映画について大好きです．最近見た映画やお気に入りのジャンルがあれば教えてください．」
ユーザー「宮崎駿監督の最新作を見ました」
に続くシステムの発話を最大 100 文字で生成してください．
```

ChatGPT の応答は以下のようなものになります．

```
システム「それは素晴らしいですね．宮崎駿監督の作品は常に素晴らしいです．その映画についてどんな感想を持ちましたか？」
```

これから以下の文字列を取り出して，システム発話として返します．

```
それは素晴らしいですね．宮崎駿監督の作品は常に素晴らしいです．その映画についてどんな感想を持ちましたか？
```

5.7.4 拡張方法

先に述べたように，dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT のサブクラスを作り，自作ブロックとして利用することで，柔軟に制御を変更することができます.

サブクラスを作る際には，以下のメソッドを実装します.

```
_generate_system_utterance(self, dialogue_history: List[Dict[str, str]],
                                 session_id: str, user_id: str,
                                 aux_data: Dict[str, Any]) -> Tuple[str, Dict[str,␣
↪Any], bool]:
```

- 引数
  - dialogue_history（辞書型のリスト）

    以下のような配列で表された対話の履歴

    ```
    [
        {"speaker": "system", "utterance": <システム発話文字列>},
        {"speaker": "user", "utterance": <ユーザ発話文字列>}
        ...
    ]
    ```

  - session_id （文字列）

    セッション ID

  - user_id （文字列）

    ユーザ ID

  - aux_data (辞書型)

    メインプロセスから受け取った補助データ

- 返り値は以下の Tuple
  - system_utterance （文字列）

    システム発話

  - aux_data (辞書型)

    アップデートした補助データ

  - final (ブール値)

    対話の終わりかどうか

5.8 spaCy-Based Named Entity Recognizer（spaCy を用いた固有表現抽出ブロック）

(dialbb.builtin_blocks.ner_with_spacy.ne_recognizer.SpaCyNER)

(ver0.6 で追加)

spaCy および GiNZA を用いて固有表現抽出を行います.

5.8.1 入出力

- 入力
  - input_text: 入力文字列（文字列）
  - aux_data: 補助データ（辞書型）
- 出力
  - aux_data: 補助データ（辞書型）

    入力された aux_data に固有表現抽出結果を加えたものです.

    固有表現抽出結果は，以下の形です.

    ```
    {"NE_<ラベル>": "<固有表現>", "NE_<ラベル>": "<固有表現>", ...}
    ```

    <ラベル>は固有表現のクラスです．固有表現は見つかった固有表現で，input_text の部分文字列です．同じクラスの固有表現が複数見つかった場合，:で連結します.

    例

    ```
    {"NE_Person": "田中:鈴木", "NE_Dish": "味噌ラーメン"}
    ```

    固有表現のクラスについては，spaCy/GiNZA のモデルのサイトを参照してください.

    * ja-ginza-electra (5.1.2):，https://pypi.org/project/ja-ginza-electra/
    * en_core_web_trf (3.5.0):, https://huggingface.co/spacy/en_core_web_trfhttps://pypi.org/project/ja-ginza-electra/

5.8.2 ブロックコンフィギュレーションのパラメータ

- model (文字列．必須)

  spaCy/GiNZA のモデルの名前です．ja_ginza_electra (日本語)，en_core_web_trf (英語）などを指定できます.

- patterns (オブジェクト．任意)

  ルールベースの固有表現抽出パターンを記述します．パターンは，spaCy のパターンの説明に書いてあるものを YAML 形式にしたものです．

  以下が日本語の例です．

```
patterns:
  - label: Date
    pattern: 昨日
  - label: Date
    pattern: きのう
```

5.8.3 処理内容

spaCy/GiNZA を用いて input_text 中の固有表現を抽出し，aux_data にその結果を入力して返します．