# DialBB 0.10 ドキュメント

リリース ***v0.10.0***

**2025 年 03 月 05 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，SIGDIAL の論文や人工知能学会全国大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，東中著：AI の雑談力や情報処理学会誌の中野の解説記事「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.

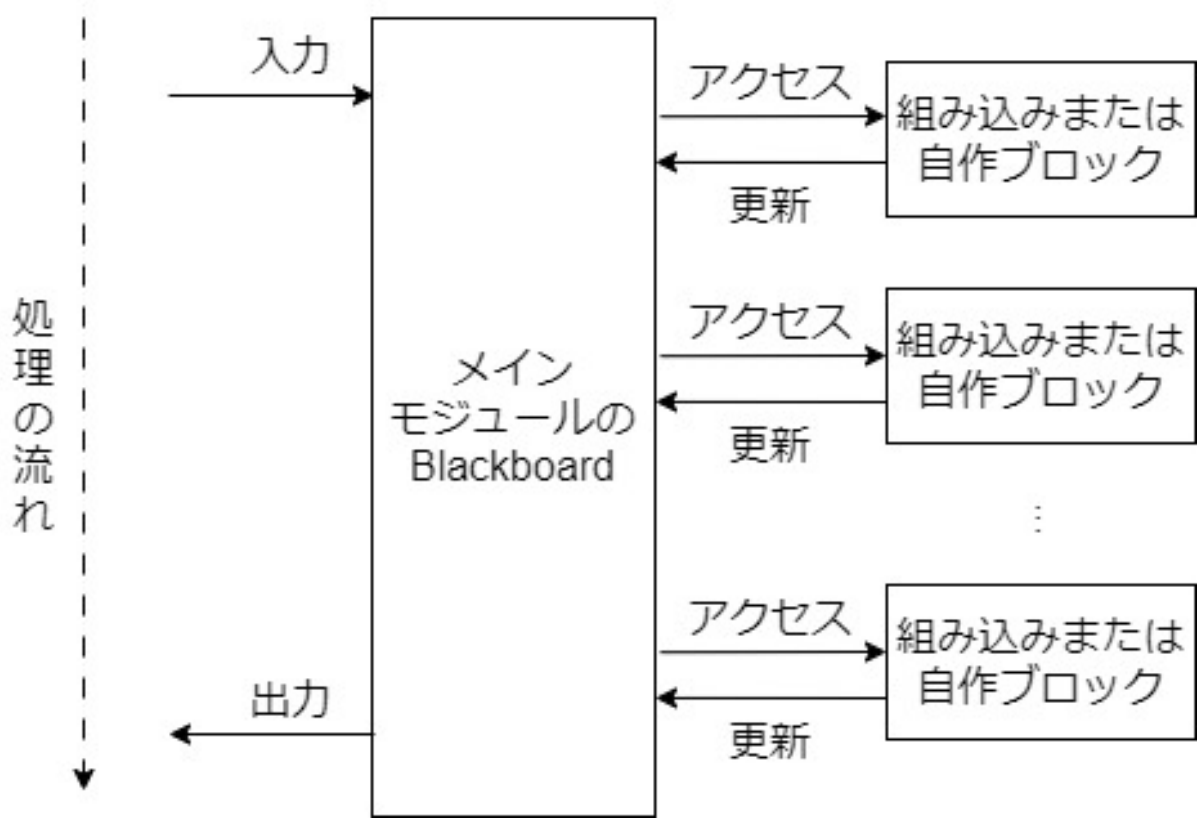
DialBB は株式会社 C4A 研究所が開発し，著作権を保有し，Apache License 2.0 のもとにに公開しています.

# 第2章　DialBBの概要

1 章に書いたように，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．この入力の内容は同じ JSON 形式のデータ blackboard[1] に書き込まれます．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

[1] ver.0.2 以前は payload と呼んでいました．

レーションファイルで指定します.

詳細は 4 章で説明します.

# 第3章　チュートリアル

## 3.1 はじめに

DialBB にはいくつかのサンプルアプリケーションが付属しています．本章ではこれらのうち，日本語のアプリケーションを用いて，DialBB のアプリケーションの構成と，DialBB を用いてアプリケーションを構築する方法を説明します．

これらのアプリケーションの動作のさせ方は README を見てください．

## 3.2 オウム返しサンプルアプリケーション

### 3.2.1 説明

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．

sample_apps/parrot/config.yml が，この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，その内容は以下のようになっています．

```yaml
blocks:
  - name: parrot
    block_class: parrot.Parrot
    input:
      input_text: user_utterance
      input_aux_data: aux_data
    output:
      output_text: system_utterance
      output_aux_data: aux_data
      final: final
```

blocks は，本アプリケーションで用いるブロックの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**と呼びます）のリストです．本アプリケーションでは，一つのブロックのみを用います．

name はブロックの名前を指定します．ログで用いられます．

block_class は，このブロックのクラス名を指定します．このクラスのインスタンスが作られて，メインモジュールと情報をやり取りします．クラス名は，コンフィギュレーションファイルからの相対パスまたは dialbb ディレクトリからの相対パスで記述します．

ブロッククラスは，diabb.abstract_block.AbstractBlock の子孫クラスでないといけません．

input はメインモジュールからの情報の受信を規定します．例えば，

```
input_text: user_utteranee
```

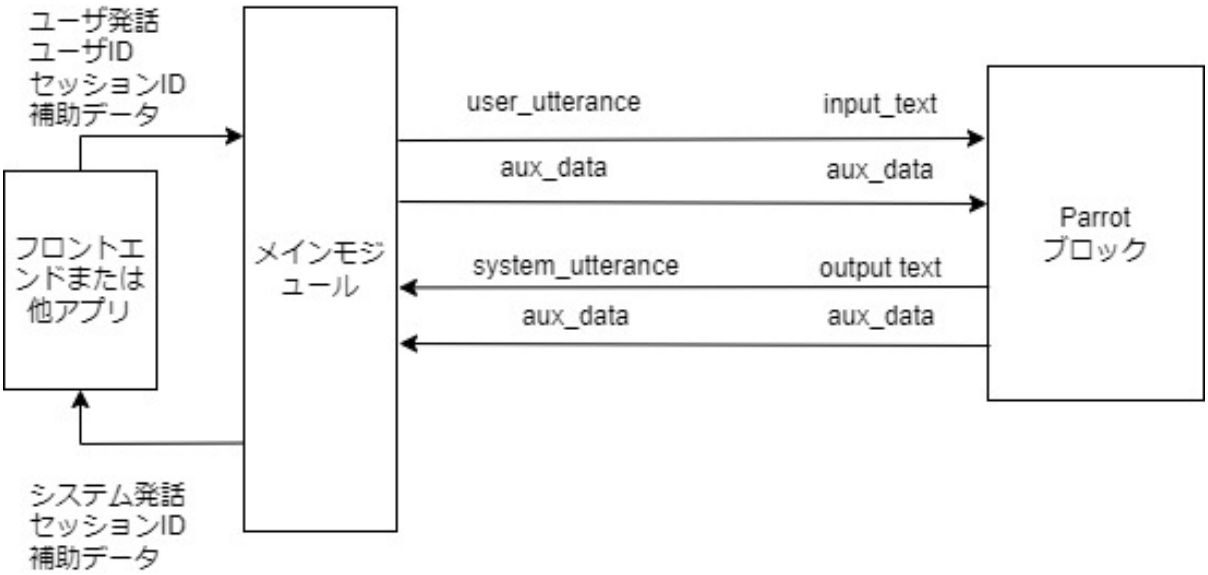
は，メインモジュールの blackboard['user_utterane'] を、ブロッククラスの process メソッドの引数（辞書型）の input_text 要素として参照できることを意味します．

output はメインモジュールへの情報の送信を規定します．例えば，

```
output_text: system_utterance
```

は，メインモジュールの blackboard['output_text'] を、ブロッククラスの process メソッドの出力（辞書型）の output_text 要素で上書きまたは追加することを意味しています．

これを図示すると以下のようになります．




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．

さらに，sample_apps/parrot/parrot.py を見ることで DialBB におけるブロッククラスの概念が理解できると思います．

### 3.2.2 デバッグモード

以下のように，環境変数 DIALBB_DEBUG に yes を設定することにより，ログレベルがデバッグモードになります．

```
export DIALBB_DEBUG=yes;python run_server.py sample_apps/parrot/config.yml
```

これにより，コンソールに詳しいログが出力されますので、それを見ることで理解が深まると思います。

## 3.3 ChatGPT 対話アプリケーション

### 3.3.1 説明

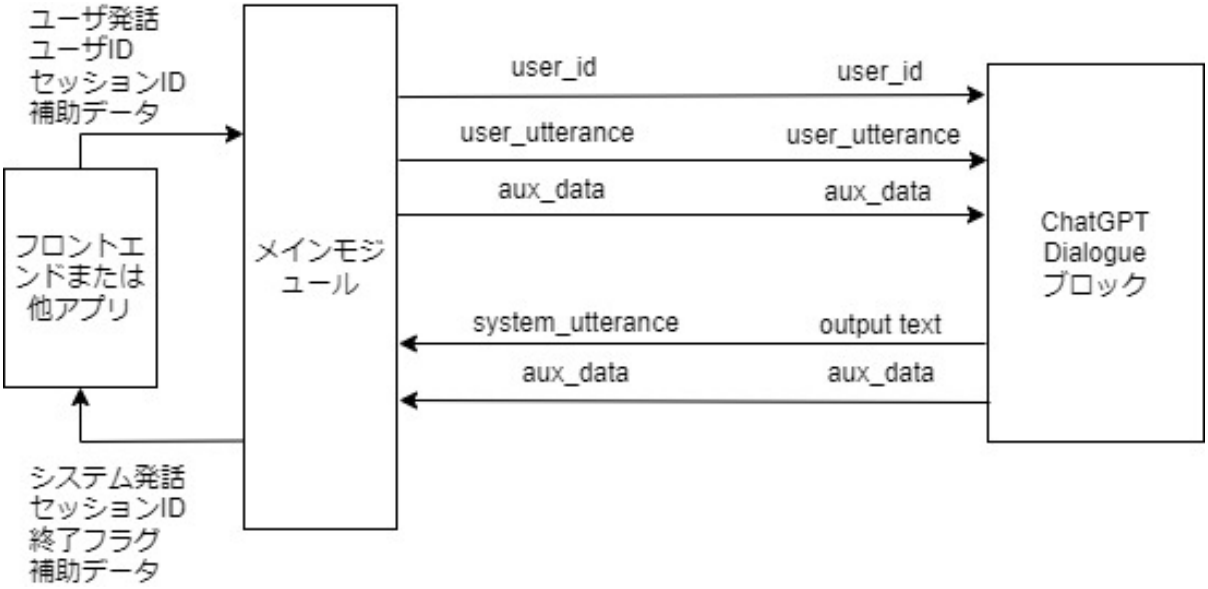
*ChatGPT Dialogue* （*ChatGPT* ベースの対話ブロック）を用い，OpenAI の ChatGPT を用いて対話を行います．

sample_apps/chatgpt/にあります．

sample_apps/chatgpt/config_ja.yml の内容は以下のようになっています．

```
blocks:
  - name: chatgpt
    block_class: dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT
    input:
      user_id: user_id
      user_utterance: user_utterance
      aux_data: aux_data
    output:
      system_utterance: system_utterance
      aux_data: aux_data
      final: final
    user_name: ユーザ
    system_name: システム
    first_system_utterance: "こんにちは．私の名前は由衣です．少しお話させてください．スイーツって好きですか？"
    prompt_template: prompt_template_ja.txt
    gpt_model: gpt-4o-mini
```

メインモジュールとの情報の授受を図示すると以下のようになります．




ブロックコンフィギュレーションのパラメータとして，input，output 以外にいくつか設定されています．

prompt_template は，システム発話のプロンプトのテンプレートを指定します．

プロンプトテンプレート sample_apps/chatgpt/prompt_template_ja.txt の中身は以下のようになっています．

```
# タスク説明

- あなたは対話システムで，ユーザと食べ物に関して雑談をしています．あなたの次の発話を 50 文字以内で生成してください．

# あなたのペルソナ

- 名前は由衣
- 28 歳
- 女性
- スイーツ全般が好き
- お酒は飲まない
- IT 会社の web デザイナー
- 独身
- 非常にフレンドリーに話す
- 外交的で陽気

#  状況

- ユーザとは初対面
- ユーザは同年代
- ユーザとは親しい感じで話す

# 対話の流れ

- 自己紹介する
- 自分がスイーツが好きと伝える
- スイーツが好きかどうか聞く
- ユーザがスイーツが好きな場合，どんなスイーツが好きか聞く
- ユーザがスイーツが好きでない場合，なんで好きじゃないのか聞く

# 現在までの対話

@dialogue_history
```

最後の@dialogue_history のところに，それまでの対話の履歴が以下のような形式で挿入されます．

```
システム：こんにちは．私の名前は由衣です．少しお話させてください．スイーツって好きですか？
ユーザ：まあまあ好きです
システム：そうなんですね！私も大好きです．どんなスイーツが好きですか？
ユーザ：どっちかというと和菓子が好きなんですよね
```

ここで，「システム」「ユーザ」はコンフィギュレーションの user_name, system_name で指定したものが

使われます.

### 3.3.2 ChatGPT アプリケーションを利用したアプリケーション作成

このアプリケーションを流用して新しいアプリケーションを作るには以下のようにします.

- sample_apps/chatgpt をディレクトリ毎コピーします．DialBB のディレクトリとは全く関係ないディレクトリで構いません.
- config.yml や prompt_template_ja.txt をを編集します．これらのファイルの名前を変更しても構いません.
- 以下のコマンドで起動します.

```
export PYTHONPATH=<DialBB ディレクトリ>;python run_server.py <コンフィギュレーションファイル>
```

## 3.4 シンプルアプリケーション

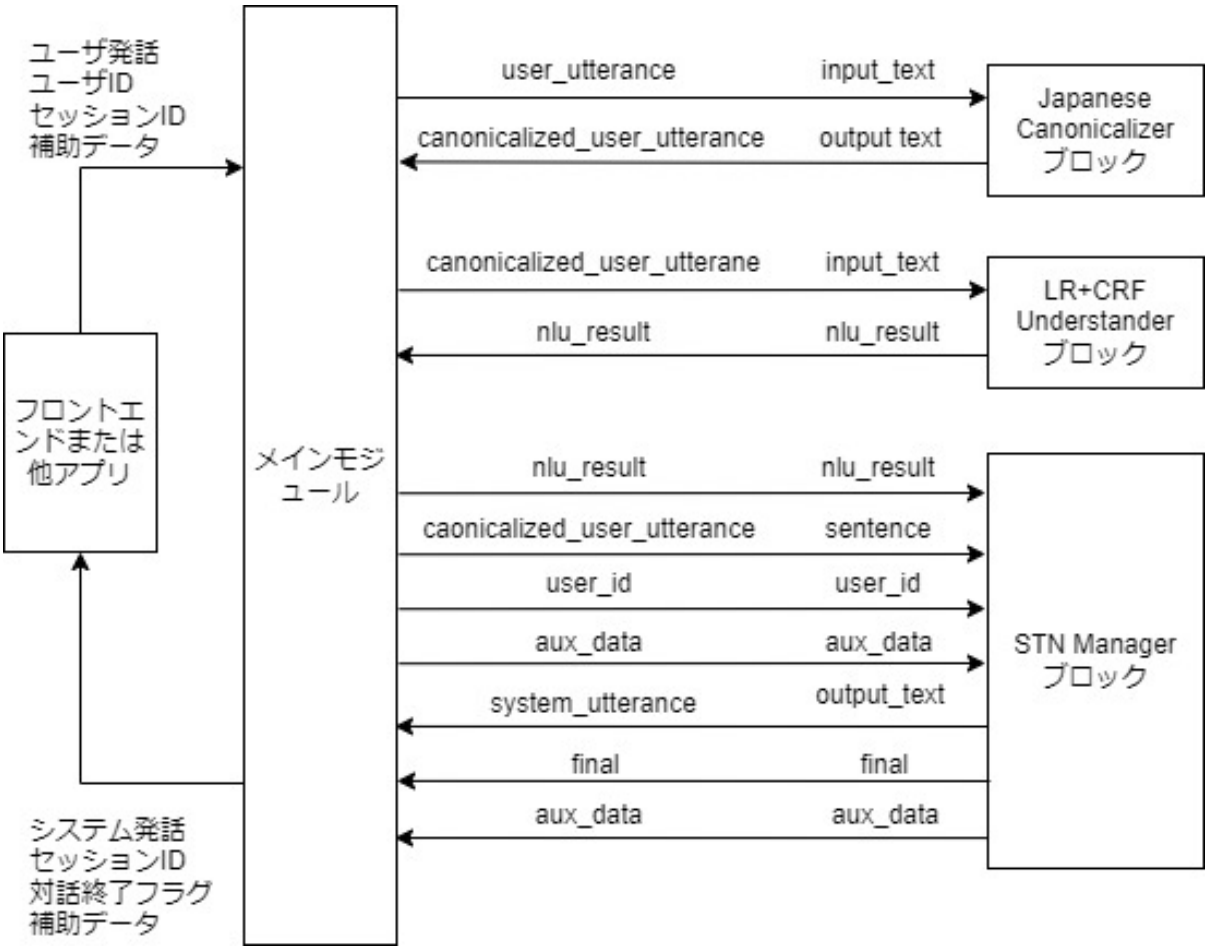
以下の組み込みブロックを用いたサンプルアプリケーションです．(v0.9 から Snips 言語理解を使わないアプリケーションに置き換わりました)

- *Japanese Canonicalizer* （日本語文字列正規化ブロック）
- *LR-CRF Understander* （ロジスティック回帰と条件付き確率場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/simple_ja/にあります.

### 3.4.1 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．これらの組み込みブロックの詳細は，5 章で説明します．

- 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．
- LR-CRF Understander: 言語理解を行います．ロジスティック回帰 (Logistic Regression) と条件付き確率場 (Conditional Random Fields) 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．

### 3.4.2 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/simple_ja` ディレクトリ（フォルダ）にあります．

`sample_apps/simple_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 4.3 章で詳細に説明します．

- `config_gs_template.yml`

  LR-CRF 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．

- `simple-nlu-knowledge-ja.xlsx`

  LR-CRF Understander ブロックで用いる知識（言語理解知識）を記述したものです．

- `simple-scenario-ja.xlsx`

  STN Manager ブロックで用いる知識（シナリオ）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

- `test_inputs.txt`

  システムテストで使うテストシナリオです．

### 3.4.3 LR-CRF Understander ブロック

#### 言語理解結果

LR-CRF 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": {
     "好きなラーメン": "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

LR-CRF Understander ブロックが用いる言語理解用の知識は，`simple-nlu-knowledge-ja.xlsx` に書かれています．言語理解知識の記述法の詳細は `nlu_knowledge` を参照してください．以下に簡単に説明します．

言語理解知識は，以下の 2 つのシートからなります．

| シート名 | 内容 |
|---|---|
| utterances | タイプ毎の発話例と，その発話例から抽出されるべきスロット |
| slots | スロットとエンティティの関係 |

utterances シートの一部を以下に示します.

| flag | type | utterance | slots |
|---|---|---|---|
| Y | 肯定 | はい | |
| Y | 否定 | そうでもない | |
| Y | 特定のラーメンが好き | 豚骨ラーメンが好きです | 好きなラーメン=豚骨ラーメン |
| Y | 地方を言う | 荻窪 | 地方=荻窪 |
| Y | ある地方の特定のラーメンが好き | 札幌の味噌ラーメンが好きです | 地方=札幌，好きなラーメン=味噌ラーメン |

一行目は「はい」のタイプが「肯定」で，スロットはないことを示しています.「はい」の言語理解結果は以下のようになります.

```
{
   "type": "肯定"
}
```

「札幌の味噌ラーメンが好きです」の言語理解結果は以下のようになります.

```
{
  "type": "ある地方の特定のラーメンが好き",
  "slots": {
     "好きなラーメン": "味噌ラーメン",
     "地方": "札幌"
  }
}
```

`flag` 列は，その行を使用するかどうかをコンフィギュレーションで規定するためのものです.

次に，`slots` シートの内容の一部を以下に示します.

| flag | slot name | entity | synonyms |
|---|---|---|---|
| Y | 好きなラーメン | 豚骨ラーメン | とんこつラーメン，豚骨スープのラーメン，豚骨 |
| Y | 好きなラーメン | 味噌ラーメン | みそらーめん，みそ味のラーメン，味噌 |

`slot name` 列はスロット名，`entity` はスロット値，`synonyms` は同義語のリストです．例えば，一行目は，好きなラーメンのスロット値として，とんこつラーメンや豚骨などが得られた場合，言語理解結果においては，豚骨ラーメンに置き換えられる，ということを表しています.

#### 言語理解モデルの構築と利用

アプリを立ち上げると，上記の知識から，ロジスティック回帰と条件付き確率場のモデルが作られ，実行時に用いられます．

### 3.4.4 STN Manager ブロック

#### 概要

STN Manager ブロックは，状態遷移ネットワーク（State-Transition Network）を用いて対話管理と言語生成を行います．状態遷移ネットワークのことをシナリオとも呼びます．シナリオは，`simple-scenario-ja.xlsx` ファイルの `scenario` シートに書かれています．このシートの書き方の詳細は 5.5.2 章を参照してください．

#### シナリオ記述

シナリオ記述の一部を以下に示します．

| flag | state | system utterance | user utterance example | user utterance type | conditions | actions | next state |
|---|---|---|---|---|---|---|---|
| Y | 好き | 豚骨ラーメンとか塩ラーメンなどいろんな種類のラーメンがありますが，どんなラーメンが好きですか？ | 豚骨ラーメンが好きです． | 特定のラーメンが好き | _eq(#好きなラーメン, "豚骨ラーメン") | _set(&topic_ramen, #好きなラーメン) | 豚骨ラーメン好き |
| Y | 好き | | 豚骨ラーメンが好きです． | 特定のラーメンが好き | is_known_r 好きなラーメン) | _set(&topic_ramen, #好きなラーメン); get_ramen_location(*t &location) | 特定のラーメン好き en, |
| Y | 好き | | | 特定のラーメンが好き | is_novel_ra 好きなラーメン) | _set(&topic_ramen, #好きなラーメン) | 知らないラーメン好き |
| Y | 好き | | 近所の街中華のラーメンが好きなんだよね | | | | #final |

各行が一つの遷移を示します．

flag 列はは言語理解知識と同じく，その行を使用するかどうかをコンフィギュレーションで規定するためのものです．

state 列は遷移元の状態の名前，next state 列は遷移先の状態の名前です．

system utterance 列はその状態で出力されるシステム発話です．システム発話はその行の遷移とは関係なく，左側の state 列の値と結びついています．

user utterance example 列は，その遷移で想定する発話の例です．実際には使いません．

user utterance type 列と conditions 列はその遷移の条件を表します．以下の場合に遷移の条件が満たされます．

- user utterance type 列が空か，または，user utterance type 列の値が言語理解結果のユーザ発話タイプがその値と同じで，かつ，
- conditions 列が空か，または，conditions 列のすべての条件が満たされるとき

これらの条件は，上に書いてある遷移から順に，満たされるかどうかを調べて行きます．

user utterance type 列も conditions 列も空のものをデフォルト遷移と呼びます．基本的に，一つの state にデフォルト遷移が一つ必要で，その state が遷移元になっている行のうち一番下にないといけません．

#### 条件

conditions 列の条件は，関数呼び出しのリストです．関数呼び出しが複数ある場合は，; でつなぎます．

conditions 列で使われる関数は，条件関数と呼ばれ，True か False を返す関数です．すべての関数呼び出しが True を返した場合，条件が満たされます．

_で始まる関数は，組み込み関数です．それ以外の関数は，開発者が作成する関数で，このアプリケーションの場合，scenario_functions.py で定義されています．

_eq は二つの引数の値が同じ文字列なら，True を返す組み込み関数です．

#好きなラーメンのように，#で始まる引数は特殊な引数です．例えば言語理解結果のスロット名に#をつけたものはスロット値を表す引数です．#好きなラーメンは好きなラーメンスロットの値です．

"豚骨ラーメン"のように，""で囲まれた引数はその中の文字列がその値になります．

_eq(#好きなラーメン，"豚骨ラーメン") は，好きなラーメンスロットの値が豚骨ラーメンの時に True になります．

is_known_ramen(#好きなラーメン) はシステムが好きなラーメンスロットの値を知っていれば True を返し，そうでなければ False を返すように scenario_functions.py の中で定義されています．

条件関数の定義の中では，文脈情報と呼ばれるデータにアクセスすることができます．文脈情報は辞書型のデータで，条件関数や後述のアクション関数の中で，キーを追加することができます．また，あらかじめ値が入っている特殊なキーもあります．詳細は 5.5.4 章を参照してください．

### アクション

actionss 列には，その行の遷移が行われたときに，実行される処理を書きます．これは，関数呼び出しの列です．関数呼び出しが複数ある場合は，；でつなぎます．

actions 列で使われる関数は，アクション関数と呼ばれ，何も返しません．

条件関数と同様，_で始まる関数は，組み込み関数です．それ以外の関数は，開発者が作成する関数で，このアプリケーションの場合，scenario_functions.py で定義されています．

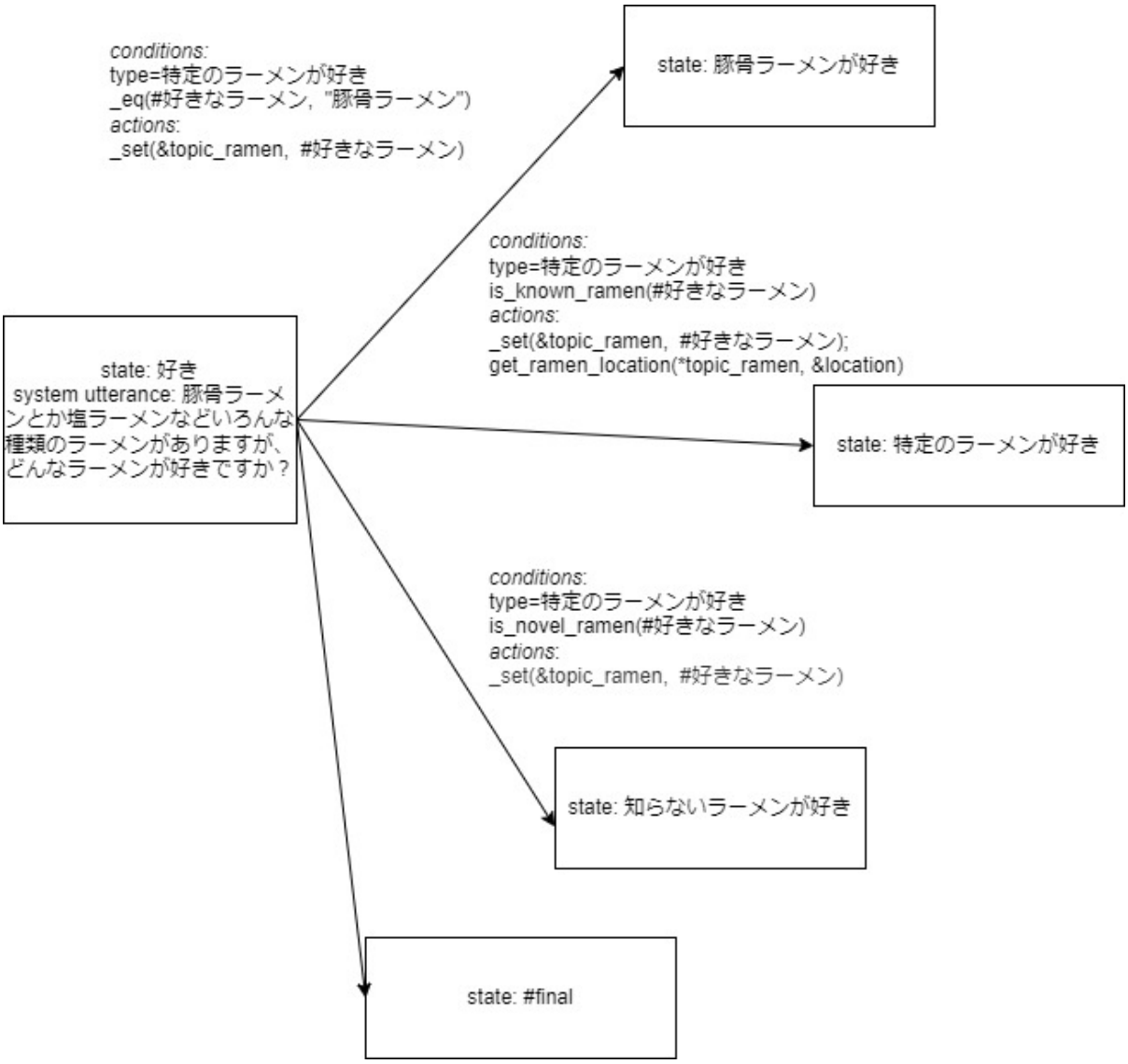
_set は第 2 引数の値を第 1 引数に代入する処理を行います．_set(&topic_ramen，#好きなラーメン) の第 1 引数&topic_ramen は，部文脈情報の topic_ramen キーの意味で，この関数呼び出しは文脈情報の topic_ramen に#好きなラーメンスロットの値をセットします．文脈情報の値は，*<キー名>で，条件やアクションの中で取り出せます．

get_ramen_location(*topic_ramen，&location) は，開発者の作成した関数の呼び出しです．get_ramen_location は scenario_functions.py で定義されています．この関数は第 1 引数の値であるラーメンの種類が名物である場所を検索し，文脈情報の第 2 引数で指定されたキーの値にセットします．例えば topic_ramen キーの値が味噌ラーメンの場合，味噌ラーメンが名物である場所を検索し，その値が札幌であれば，文脈情報の location の値を札幌に設定する，という処理を行います．

### 遷移の記述のまとめ

まとめると，1 行目は，状態が好きの時，「豚骨ラーメンとか塩ラーメンなどいろんな種類のラーメンがありますが，どんなラーメンが好きですか？」を発話し，次のユーザの発話の言語理解結果のタイプが特定のラーメンが好きで，好きなラーメンスロットの値が豚骨ラーメンであれば，条件が満たされて遷移が行われ，好きなラーメンスロットの値，すなわち，豚骨ラーメンが，文脈情報の topic_ramen の値にセットされ，豚骨ラーメンが好き状態に移行します．条件が満たされない場合，2 行目の条件が調べられます．

これを図示すると以下のようになります．




### 特殊な状態名

状態名には特殊なものがあります．

`#prep` は，対話が始まる前の状態で，セッション開始後，条件判定とアクションが実行されます．

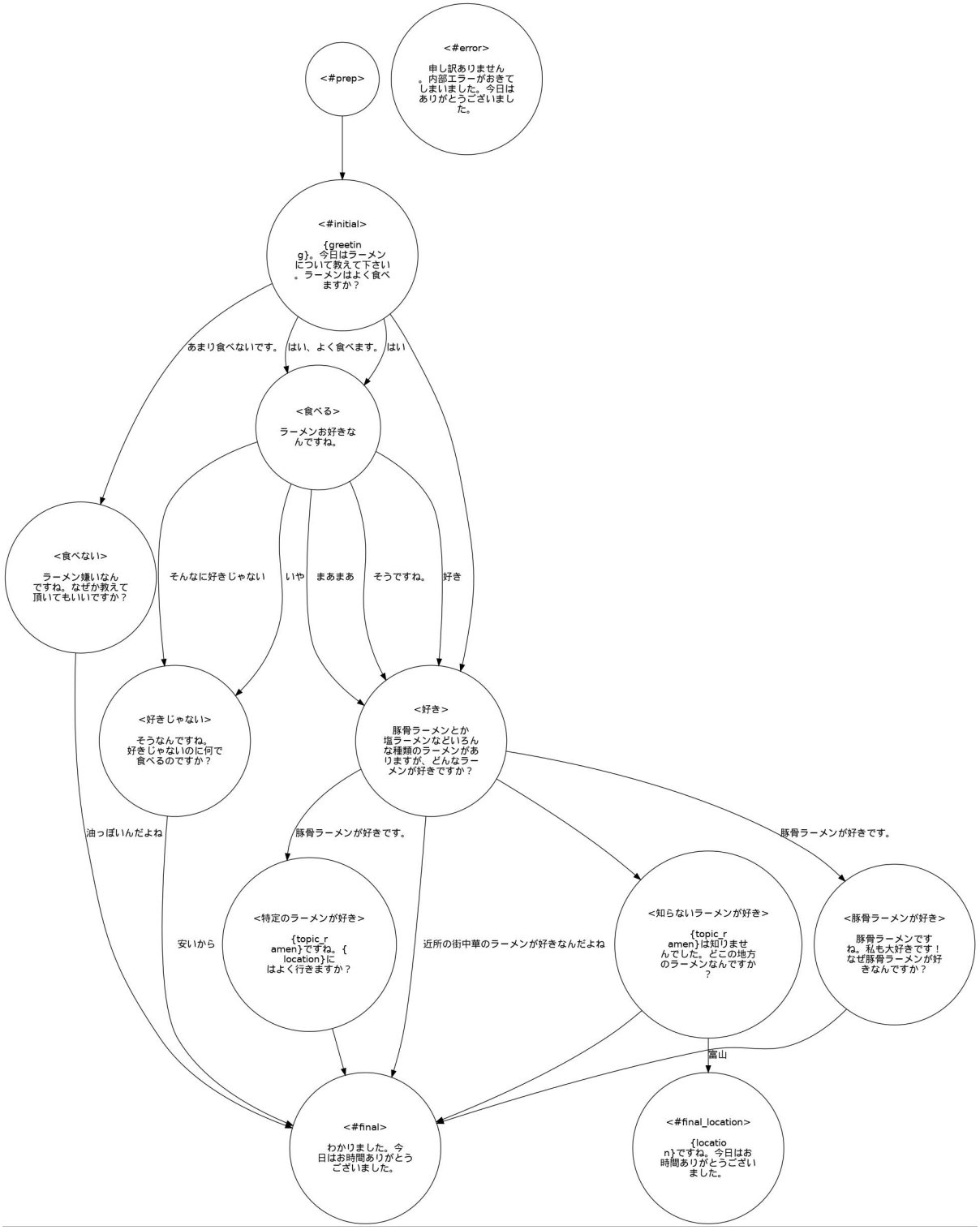
`#initial` は，最初のユーザ発話を生成する状態です．

`#final` で始まる名前の状態は，最終状態です．ブロックの出力の `final` に `True` を入れて返すので，対話が終了します．

`#error` は，内部エラーが起きた場合に遷移する状態です．これもブロックの出力の `final` に `True` を入れて返します．

### シナリオグラフ

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, `system utterance` 列のシステム発話と `user utterance example` 列のユーザ発話例を利用したグラフ (シナリオグラフ_scenario_graph.jpg) を出力します．以下が本アプリケーションのシナリオグラフです．



### 3.4.5 N-Best 言語理解結果の利用

本アプリケーションでは，LR-CRF 言語理解ブロックが 5-Best の理解結果を出力するようになっています．これは，コンフィギュレーションファイルの以下の num_candidates 要素で指定されています．

```
blocks:  # bclock list
  - ....
  - name: understander
    ....
    num_candidates: 3
  - ....
```

STN Manager ブロックでは，遷移の条件を調べるときに上位の言語理解結果から順に調べ，条件を満たすものがあれば．その結果を用いてアクションを実行して，次の状態に遷移します．

### 3.4.6 シンプルアプリケーションを利用したアプリケーション構築

#### 概要

{}3.3.2 章と同様に，シンプルアプリケーションを利用したアプリケーション構築法を説明します．

- sample_apps/simple_ja をディレクトリ毎コピーします．DialBB のディレクトリとは全く関係ないディレクトリで構いません．
- 各ファイルを編集します．. これらのファイルの名前を変更しても構いません．
- 以下のコマンドで起動します．

  ```
  export PYTHONPATH=<DialBB ディレクトリ>;python run_server.py <コンフィギュレーションファイル>
  ```

#### 変更するファイル

- simple-nlu-knowledge-ja.xlsx

  LR-CRF 言語理解ブロックで用いる知識を編集します．

- simple-scenario-ja.xlsx

  シナリオを編集します．

- scenario_functions.py

  シナリオに付け加えた関数（条件関数，アクション関数）の定義を行います．（5.5.6 章参照．）

  各関数の定義では，シナリオで用いる引数にプラスして，文脈情報を表す辞書型の引数を付加する必要があります．一般的には，context: Dict[str, Any] とします．文脈情報には，あらかじめ登録されている情報と対話中にアクション関数によって登録される情報があります．詳細は 5.5.4 章を見てください．

- `config.yml`

  基本的な機能のみを用いるならば，あまり変更する必要はないと思います．

## 3.5 実験アプリケーション

### 3.5.1 概要

ChatGPT による言語理解とネットワークベース対話管理を軸に，組み込みブロックの様々な機能を含んでいるものです．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）
- *ChatGPT Understander* （*ChatGPT* 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）

シンプルアプリケーションとの大きな違いは，ChatGPT 言語理解ブロックと spaCy 固有表現抽出ブロックを用いているところと，STN Manager ブロックの先進的な機能を用いているところです．

### 3.5.2 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/simple_ja` ディレクトリ（フォルダ）にあります．そのディレクトリ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．

- `lab_app_nlu_knowledge_ja.xlsx`

  ChatGPT 言語理解ブロックで用いる知識を記述したものです．

- `lab_app_scenario_ja.xlsx`

  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

- `test_requests.json`

  テストリクエスト（4.7 章）のファイルです．

### 3.5.3 ChatGPT 言語理解ブロック

ChatGPT の JSON モードを用いて言語理解を行います．言語理解用の知識は LR-CRF 言語理解ブロックと同じ形式のものを用います．これを Few-shot learning に用いて，入力テキストに対して言語理解を行います．入出力 LR-CRF 言語理解ブロックも同じです．

GPT のモデルは，ブロックコンフィギュレーションの `gpt_model` で指定しています．

ChatGPT に言語理解を行わせる場合のプロンプトのテンプレートはデフォルトのものを用いていいます．詳細は 5.4.2 章を参照してください．

### 3.5.4 spaCy 固有表現抽出ブロック

spaCy/GiNZA を用いて固有表現抽出を行います．抽出した結果は，ブロックの出力の `aux_data` に入れて返します．以下が例です．

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

`Person` や `Dish` は spaCy/GiNZA のモデルで定義されている固有表現のクラスです．この結果は STN Manager ブロックの中で，`#Person` や`#Dish` という特殊変数で取り出すことができます．

spaCy/GiNZA のモデルは，ブロックコンフィギュレーションの `model` で指定します．現在 `ja_ginza_electra` が指定されています．

ブロックコンフィギュレーションの `patterns` 要素に，固有表現の例を追加し，抽出しやすくすることができます．現在以下のように指定されています．

```
patterns:
  - label: Person
    pattern: 大谷
  - label: Person
    pattern: ダルビッシュ
```

### 3.5.5 STN Manager の機能

実験アプリケーションでは，シンプルアプリケーションでは用いていない以下の STN Manager の機能を使っています．

### システム発話中の関数呼び出し・特殊変数参照

シンプルアプリケーションでは，システム発話中には，文脈情報の変数しか埋め込んでいませんでしたが，関数呼び出しや特殊変数も埋め込むことができます．

例えば，

```
私は{get_system_name()}です．よろしければお名前を教えて頂けますか？
```

の場合，scenario_fuctions.py で定義されている get_system_name(context) が呼ばれ，その返り値（文字列）が{get_system_name()}に置き換わります．なお，get_system_name は，コンフィギュレーションファイルの system_name の値を返すように定義されています．

```
ありがとうございます．{#NE_Person}さん，今日はラーメンについて教えて下さい．ラーメンはよく食べますか？
```

の場合，{#NE_Person}は，特殊変数#NE_Person の値で置き換えられます．#NE_Person は aux_data の NE_Person の値，すなわち固有表現抽出の Person の値になります．

### シンタクスシュガー

シナリオ内で組み込み関数呼び出しを簡単に記述するためのシンタクスシュガーが用意されています．たとえば confirmation_request="すみません．ラーメンってよく食べますか？"は_set(&confirmation_request, "すみません．ラーメンってよく食べますか？") と同じです．

#好きなラーメン=="豚骨ラーメン"は，_eq(#好きなラーメン, "豚骨ラーメン") と同じで，#NE_Person!=""は，_ne(#NE_Person, "") と同じです．

### リアクション発話生成

シナリオの actions 欄に_reaction="そうなんですね．"があります．_reaction は文脈情報の特殊な変数で，次のシステム発話の先頭にこの変数の値を付加します．たとえば，この後状態好きに遷移すると，システム発話豚骨ラーメンとか塩ラーメンなどいろんな種類のラーメンがありますが，どんなラーメンが好きですか？の先頭に，そうなんですね．が付加され，そうなんですね．豚骨ラーメンとか塩ラーメンなどいろんな種類のラーメンがありますが，どんなラーメンが好きですか？が発話されます．

このようにユーザ発話に対するリアクションを与えることで，ユーザの言ったことを聞いているよ，ということを示すことができ，ユーザ体験が良くなります．

### ChatGPT を用いた発話生成・条件判定

システム発話中にある，$"それまでの会話につづけて，対話を終わらせる発話を 50 文字以内で生成してください．"は，組み込み関数呼び出し_check_with_llm("それまでの会話につづけて，対話を終わらせる発話を 50 文字以内で生成してください．"}のシンタクスシュガーで，ChatGPT を用いて発話を生成します．

また，conditions 欄にある$"ユーザが理由を言ったかどうか判断してください．"は，組み込み関数呼び出し_check_with_llm("ユーザが理由を言ったかどうか判断してください．") のシンタクスシュガーで，これまでの対話から ChatGPT を用いて判定を行い，bool 値で返します．

発話を生成する際の ChatGPT のモデルや温度パラメータ，状況設定，ペルソナはブロックコンフィギュレーションで以下のように指定されています．

```
chatgpt:
  gpt_model: gpt-4o-mini
  temperature: 0.7
  situation:
    - あなたは対話システムで，ユーザと食べ物に関して雑談をしています．
    - ユーザとは初対面です
    - ユーザとは同年代です
    - ユーザとは親しい感じで話します
  persona:
    - 名前は由衣
    - 28 歳
    - 女性
    - ラーメン全般が好き
    - お酒は飲まない
    - IT 会社の web デザイナー
    - 独身
    - 非常にフレンドリーに話す
    - 外交的で陽気
```

### サブダイアローグ

next state 欄に#gosub: 確認: ラーメン食べるか確認完了があります．これは，確認状態に遷移して対話を行った後，ラーメン食べるか確認完了状態に戻るという意味です．確認状態から始まる対話をサブダイアローグと呼びます．サブダイアローグが#exit 状態に移行すると，サブダイアローグから抜けて，ラーメン食べるか確認完了状態に行きます．

ユーザに確認を行うような，いろいろな場面で再利用可能な対話を，サブダイアローグとして用意しておくことで，シナリオ記述を減らすことが可能です．

### スキップ遷移

system utterance 欄に$skip があるとシステム発話を返さず，すぐに条件判定を行って次の遷移を行います．アクションの結果をもとにさらに遷移先を変えたいときなどに使います．

### リピート機能

ブロックコンフィギュレーションで repeat_when_no_available_transitions が指定されています．これが指定されているとき，条件を満たす遷移がなければ，元の状態に戻って同じ発話を繰り返します．この場合，デフォルト遷移がない状態があっても構いません．本アプリケーションの場合，好き状態にはデフォルト遷移がないので，この状態で全く関係ない発話をした場合には，同じシステム発話が繰り返されます．

### 音声入力に対処するための機能

STN Manager には音声入力に対処するための機能があり，ブロックコンフィギュレーションで設定することで利用できます．本アプリケーションでは以下のように設定しています．

```
input_confidence_threshold: 0.5
confirmation_request:
  function_to_generate_utterance: generate_confirmation_request
  acknowledgement_utterance_type: 肯定
  denial_utterance_type: 否定
ignore_out_of_context_barge_in: yes
reaction_to_silence:
  action: repeat
```

これらの意味は 5.5.11 章を見てください．

test_requests.json には音声入力に対応した入力の例が入っています．

# 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，ユーザ発話と付加情報を JSON 形式で受けとり，システム発話と付加情報を 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 を利用する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，サーバのレスポンスに含まれているセッション 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，システムの発話です．
- <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.

  指定されたキーが blackboard にない場合, 該当する input の要素は None になります.

- 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（4.3 章を参照）. 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 (辞書型)

  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 (辞書型)

  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 (文字列)

  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.

- self.config_dir (文字列)

  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ファイル> \
  <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．

## 4.7 テストリクエストを用いたテスト

テストシナリオを用いたテストでは aux_data を入力できません．aux_data を入力するテストを行うには，テストリクエストを用いたテストを行います．

以下のコマンドで実行します．

```
$ python dialbb/util/send_test_requests.py <アプリケーションコンフィギュレーションファイル
> \                <テストリクエストファイル>
```

テストリクエストファイルは以下の形式の JSON ファイルです．

```
[
    [
      <最初のセッションの最初の入力>,
      <最初のセッションの2番目の入力>,
      ...
    ],
    [
      <2番目のセッションの最初の入力>,
      <2番目のセッションの2番目の入力>,
      ...
    ],
    ...
]
```

各入力は，以下の形をしています．

- 各セッションの最初の入力（aux_data がない場合）

```
{ "user_id": <ユーザID>}
```

- 各セッションの最初の入力（aux_data がある場合）

```
{
    "user_id": <ユーザID>,
    "aux_data": <aux_dataの値（辞書）>
}
```

–

- 2 番目以降の入力（aux_data がない場合）

```
{
    "user_id": <ユーザID>,
    "user_utterance": "<ユーザ発話テキスト>"
}
```

- 2 番目以降の入力（aux_data がある場合）

```
{
    "user_id": <ユーザID>,
    "user_utterance": "<ユーザ発話テキスト>",
    "aux_data": <aux_dataの値（辞書）>
}
```

各セッション最初の入力はそのまま，2 番目以降は session_id が付加されて DialBB アプリケーションに入力されます．

user_id は省略しても構いません．その場合，各セッションの最初の入力は，aux_data がない場合，空辞書になります．

# 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．

## 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 LR-CRF Understander（ロジスティック回帰と条件付き確率場を用いた言語理解ブロック）

(dialbb.builtin_blocks.understanding_with_lr_crf.lr_crf_understander.Understander)

ロジスティック回帰と条件付き確率場を用いて，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，ロジスティック回帰と条件付き確率場のモデルを学習します．

実行時は，学習したモデルを用いて言語理解を行います．

### 5.3.1 入出力

- 入力
  - input_text: 入力文字列

  入力文字列は正規化されていると仮定します.

  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>,
```

(次のページに続く)

(前のページからの続き)

```
  ...,
  <スロット名>: <スロット値>
  }
},
  ....
]
```

### 5.3.2 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．

- 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．

- num_candidates（整数．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．

- canonicalizer

  辞書記述を正規化する際に使うプログラムを指定します．

  - class （文字列）

    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．

- knowledge_google_sheet (ハッシュ)

  - Excel の代わりに Google Sheets を用いる場合の情報を記述します．（Google Sheets 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．

### 5.3.3 言語理解知識

言語理解知識は，以下の 2 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および同義語のリスト |

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．

#### 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．

- `slots`

  発話に含まれるスロット．スロットを以下の形で記述します．

  ```
  <スロット名>=<スロット値>，<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．

  ```
  地方=札幌，好きなラーメン=味噌ラーメン
  ```

  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`

  辞書エントリー名．言語理解結果に含まれます.

- `synonyms`

  同義語を`','` で連結したものです.

## 5.4 ChatGPT Understander （ChatGPT を用いた言語理解ブロック）

(`dialbb.builtin_blocks.understanding_with_chatgpt.chatgpt_understander.Understander`)

OpenAI 社の ChatGPT を用いて，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，プロンプトのユーザ発話タイプのリスト，スロットのリスト，Few shot example に変換します.

実行時は，プロンプトに入力発話を付加して ChatGPT に言語理解を行わせます.

### 5.4.1 入出力

- 入力
  - `input_text`: 入力文字列

  入力文字列は正規化されていると仮定します.

  例："好きなのは醤油"

- 出力
  - `nlu_result`: 言語理解結果（辞書型）

    以下の形式

```
{
  "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"
  }
}
```

### 5.4.2 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.

- 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

  辞書記述を正規化する際に使うプログラムを指定します.

  - class （文字列）

    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.

- 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.

- gpt_model (文字列．デフォルト値は gpt-3.5-turbo）

  ChatGPT のモデルを指定します．gpt-4-turbo などが指定できます．gpt-4 は指定できません.

- prompt_template

  プロンプトテンプレートを書いた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.

  これが指定されていない場合は，dialbb.builtin_blocks.understanding_with_chatgpt.prompt_templates_ja .PROMPT_TEMPLATE_JA （日本語）または，dialbb.builtin_blocks.understanding_with_chatgpt.prompt_templates_en .PROMPT_TEMPLATE_EN （英語）が使われます.

  プロンプトテンプレートは，言語理解を ChatGPT に行わせるプロンプトのテンプレートで，@で始まる以下の変数を含みます.

  - @types 発話タイプの種類を列挙したものです.
  - @slot_definitions スロットの種類を列挙したものです.
  - @examples 発話例と，タイプ，スロットの正解を書いたいわゆる few shot example です.
  - @input 入力発話です.

  これらの変数には，実行時に値が代入されます.

### 5.4.3 言語理解知識

本ブロックの言語理解知識の記述形式は，LR-CRF Understander の言語理解知識の記述形式と全く同じです．詳細は LR-CRF Understander の説明の 5.3.3 章を参照してください.

## 5.5 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"}
```

### 5.5.1 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（dictionary_function 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)

  LR-CRF 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，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．

- multi_party （ブール値．デフォルト値 False．ver. 0.10.0 で追加）

  この値が true のとき，5.5.4 章の対話履歴の内容、および、5.5.6 章の大規模言語モデルを用いる組み込み関数のプロンプトに入る対話履歴で、user_id の値が用いられます。

### 5.5.2 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>}または{<関数呼び出し>}は，対話中にその変数に代入された値や関数呼び出しの結果で置き換えられます．これについては，以下の {numref}realization_in_system_utterance`で詳しく説明します．

  `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，5.5.4 章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，5.5.4 章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．

repeat_when_no_available_transitions が True の場合以外は，デフォルト遷移が必要です．

### 5.5.3 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．

### 5.5.4 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>
  },
  ...
]
```

ブロックコンフィギュレーションの multi_party の値が true の時、"user"の代わりに、user_id の値を用います。

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です．*<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．

### 5.5.5 システム発話中の変数や関数呼び出しの扱い

システム発話中の{と}に囲まれた部分の変数や関数呼び出しは，その変数の値や，関数呼び出しの返り値で置き換えられます．

変数は#で始まるものは上記の特殊変数です．それ以外のものは通常の変数で，文脈情報にあるはずのものです．それらの変数が存在しない場合は，置換されず変数名がそのまま使われます．

関数呼び出しの場合，関数は条件やアクションで用いる関数と同じように上記の引数を取ることができます．返り値は文字列でないといけません．

### 5.5.6 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")`
  - `_not_member_of(x, y)`
  - `_num_turns_exceeds(n)`

    文字列 n が表す整数よりもターン数（ユーザの発話回数）が多いとき，True を返します.

    例：`_num_turns_exceeds("10")`
  - `_check_with_llm(task)`

    大規模言語モデル（現在は OpenAI の ChatGPT のみ）を用いて判定をします．後述します.
- 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．

- システム発話内で用いる関数
  - _generate_with_llm(task)

    大規模言語モデル（現在は OpenAI の ChatGPT のみ）を用いて文字列を生成します．後述します．

### 大規模言語モデルを用いた組み込み関数

_check_with_llm(task) および_generate_with_llm(task) は，大規模言語モデル（現在は OpenAI の ChatGPT のみ）と，対話履歴を用いて，条件の判定および文字列の生成を行います．以下が例です．

- 条件判定の例

```
_check_with_llm("ユーザが理由を言ったかどうか判断してください．")
```

- 文字列生成の例

```
_generate_with_llm("それまでの会話につづけて，対話を終わらせる発話を 50 文字以内で生成してください")
```

これらの関数を使うためには，以下の設定が必要です．

- 環境変数 OPENAI_API_KEY に OpenAI の API キーをセットする

  OpenAI の API キーの取得の仕方は Web サイトなどで調べてください．

- ブロックコンフィギュレーションの chatgpt 要素に以下の要素を加える
  - gpt_model （文字列）

    GPT のモデル名です．gpt-4-turbo，gpt-3.5-turbo 等を指定できます．デフォルト値は gpt-3.5-turbo です．gpt-4 は指定できません．

  - temperature (float)

    GPT の温度パラメータです．デフォルト値は 0.7 です．

  - situation （文字列のリスト）

    GPT のプロンプトに書く状況を列挙したものです．

    この要素がない場合，状況は指定されません．

  - persona （文字列のリスト）

    GPT のプロンプトに書くシステムのペルソナを列挙したものです．

この要素がない場合，ペルソナは指定されません．

例：

```
chatgpt:
  gpt_model: gpt-4-turbo
  temperature: 0.7
  situation:
    - あなたは対話システムで，ユーザと食べ物に関して雑談をしています．
    - ユーザとは初対面です
    - ユーザとは同年代です
    - ユーザとは親しい感じで話します
  persona:
    - 名前は由衣
    - 28 歳
    - 女性
    - ラーメン全般が好き
    - お酒は飲まない
    - IT 会社の web デザイナー
    - 独身
    - 非常にフレンドリーに話す
    - 外交的で陽気
```

#### 組み込み関数の簡略記法

組み込み関数の記述を簡単にするために以下の簡略記法（Syntax Sugar) が用意されています．

- <変数名>==<値>

  _eq(<変数名>, <値>) の意味です．

  例：

  ```
  #好きなラーメン=="豚骨ラーメン"
  ```

- <変数名>!=<値>

  _ne(<変数名>, <値>) の意味です．

  例：

  ```
  #NE_Person!=""
  ```

- <変数名>=<値>

  _set(&<変数名>, <値>) の意味です．

  例：

```
user_name=#NE_Person
```

- $<タスク文字列>

  条件として使われた時は，_check_with_llm(<タスク文字列>) の意味で，システム発話中に{}で囲まれて文字列生成として使われた時は，_generate_with_llm(<タスク文字列>) の意味です．

  条件の例：

  ```
  $"ユーザが理由を言ったかどうか判断してください．"
  ```

  文字列生成を含むシステム発話の例：

  ```
  わかりました．{$"それまでの会話につづけて，対話を終わらせる発話を 50 文字以内で生成してください．"}今日はお時間ありがとうございました．
  ```

#### 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，コンフィギュレーションファイルのブロックコンフィギュレーションの function_definition で指定されているモジュールのファイル（Simple アプリケーションでは 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．

### 5.5.7 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．

### 5.5.8 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．

### 5.5.9 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．

### 5.5.10 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．

### 5.5.11 外部データベースへの文脈情報の保存

DialBB アプリケーションを Web サーバとして動作させる場合、リクエストが集中した際にロードバランサを使って複数インスタンスでに処理を分散させる場合、文脈情報を外部 DB（MongoDB) に保存することで、一つのセッションを異なるインスタンスで処理することが可能です。(ver. 0.10.0 で追加)

外部 DB を使うには、ブロックコンフィギュレーションに以下のように context_db 要素を指定します。

```
context_db:
  host: localhost
  port: 27017
  user: admin
  password: password
```

各キーは以下です。

- host （str)

  MongoDB が動作しているホスト名

- port (int. デフォルト値 27017)

  MongoDB のアクセスのためのポート番号

- user （str)

  MongoDB のアクセスのためのユーザ名

- password （str)

  MongoDB のアクセスのためのパスワード

### 5.5.12 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 要素を必ず持ちます．action 要素の値は文字列で"repeat"か"transition"です．action 要素の値が transition の場合，destination 要素が必須です．その値は状態名（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．

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

(ver0.6 で追加，ver0.7 で大幅に変更)

(dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT)

OpenAI 社の ChatGPT を用いて対話を行います．

### 5.6.1 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）

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．

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

### 5.6.2 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．

- user_name （文字列，デフォルト値は"User"）

  ChatGPT のプロンプトに使う文字列です．以下で説明します．

- system_name （文字列，デフォルト値は"System"）

  ChatGPT のプロンプトに使う文字列です．以下で説明します．

- prompt_template （文字列）

  ChatGPT のプロンプトのテンプレートを記述したファイル名です．アプリケーションディレクトリからの相対で記述します．

  プロンプトテンプレートは，システム発話の生成を ChatGPT に行わせるプロンプトのテンプレートで，@で始まる以下の変数を含みます．

  - @dialogue_history 対話の履歴です．実行時に以下のような形の値が代入されます．

    ```
    <ブロックコンフィギュレーションの system_name の値>: <システム発話>
    <ブロックコンフィギュレーションの user_name の値>: <ユーザ発話>
    <ブロックコンフィギュレーションの system_name の値>: <システム発話>
    <ブロックコンフィギュレーションの user_name の値>: <ユーザ発話>
    ...
    <ブロックコンフィギュレーションの system_name の値>: <システム発話>
    <ブロックコンフィギュレーションの user_name の値>: <ユーザ発話>
    ```

- 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, gpt-4-turbo などが指定できます．

### 5.6.3 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，プロンプトテンプレートの@dialogue_history に対話履歴を代入したものを与えて ChatGPT に発話を生成させ，返ってきた文字列をシステム発話として返します．

## 5.7 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．

### 5.7.1 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://spacy.io/models/en#en_core_web_trf-labels

### 5.7.2 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
```

### 5.7.3 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.

# 第6章　Appendix

## 6.1 フロントエンド

DialBB には、Web API にアクセスするための、2 種類のサンプルフロントエンドが付属しています。

### 6.1.1 シンプルなフロントエンド

以下でアクセスできます。

```
http://<ホスト>:<ポート番号>
```

システム発話とユーザ発話を吹き出しで表示します。

aux_data の送信はできません。また、レスポンスに含まれるシステム発話以外の情報は表示されません。

### 6.1.2 デバッグ用フロントエンド

以下でアクセスできます。

```
http://<ホスト>:<ポート番号>/test
```

システム発話とユーザ発話をリスト型式で表示します。

aux_data の送信ができます。また、レスポンスに含まれる aux_data も表示されます。

## 6.2 廃止された機能

### 6.2.1 Snips Understander 組み込みブロック

Snips が Python3.9 以上ではインストールが困難なため，ver. 0.9 で廃止されました．代わりに LR-CRF Understander 組み込みブロックを用いてください．

### 6.2.2 Whitespace Tokenizer 組み込みブロックおよび Sudachi Tokenizer 組み込みブロック

ver. 0.9 で廃止されました．LR-CRF Understander や ChatGPT Understander を使えば Tokenizer ブロックを使う必要はありません．

### 6.2.3 Snips+STN サンプルアプリケーション

ver. 0.9 で廃止されました．