# AIWolfPy v0.4.9

Kei Harada (cash)

# Table of Contents

# AIWolfPyとは

- aiwolf.orgさんの人狼知能サーバーに、pythonから接続するためのパッケージです
  - http://aiwolf.org
- 本家はJavaですが、自力でサーバーにTCP/IP接続できれば他言語でも受け付けてくれます
  - しかし、通信プロトコルをまとめたドキュメントはない(Javaで開発するとそんなのいらない)上、いつの間にかアップデートされるので、自分で通信されているJSONを読み解く必要があります
  - これを頑張った結果をまとめたのがAIWolfPyです
  - ついでに色々つけています

# AIwolfの魅力

- 何が面白いか？
  - 人狼：面白い！
  - AI開発：面白い！
  - ボット開発：面白い！
  - プロコン：面白い！
  - 将棋・囲碁はやばい人がいるけど、人狼はまだまだブルーオーシャン

面白い！

- 学術的な意義（個人の意見です）
  - そもそも完全情報ゲームではないので色々新しい
  - 多人数なので、２人のゲームにはない「説得」「騙す」が重要

# Pythonの魅力

- 世界的にメジャーなスクリプト言語、Google他で活用
- コード量が短くて済む
- インデントが必須なので、（最初は面倒だが）コードが見やすい
- 日本語・英語共に情報が充実していて、検索すればだいたいわかる
- 機械学習、特に深層学習のライブラリが充実
- 何も考えずにfor文とか書くと遅い(体感10倍)けど、良いパッケージは内部的にC等の高速なものを呼んでいて、うまく使えば速い

# 人狼知能大会は１ファイル作れば

# OK！そうPythonならね

- １ファイル自分で作れば動きます（！）
- ./hogehoge.py でスクリプトを実行するので、
  このスクリプト内でAgentオブジェクトを作成して、作成したAgentを引数に入れて
  aiwolfpy.connect_parse
- 大会参加の場合は名前（登録チーム名）に注意

# Chap. 1. Agentの動かし方

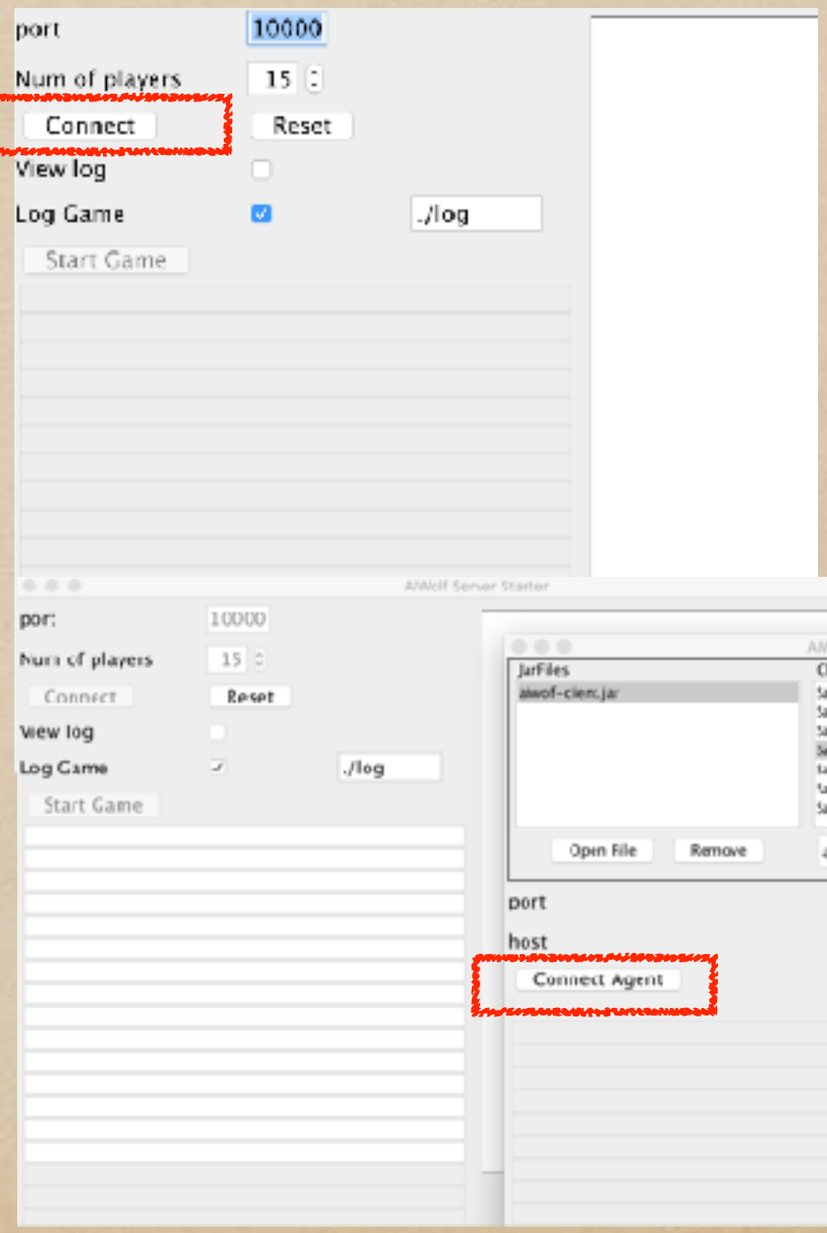
- 1. 1. 動作環境
- 1. 2. AIWolf Serverの起動
- 1.3. Python Agentの接続

# 1.1. 動作環境

- Python
  - python2系 2.7(動作確認は2.7.12)
  - python3系 3.4 ～(動作確認は3.5.2)
    - 大会の実行環境は2.7系だったと思いますが、最新版の詳細は運営さんに確認してください
- 標準パッケージ + numpy, spicy, pandas, scikit-learn
  - Anaconda入れておけば大丈夫です
- Java実行環境
  - 手元で対戦させるのにないと不便です

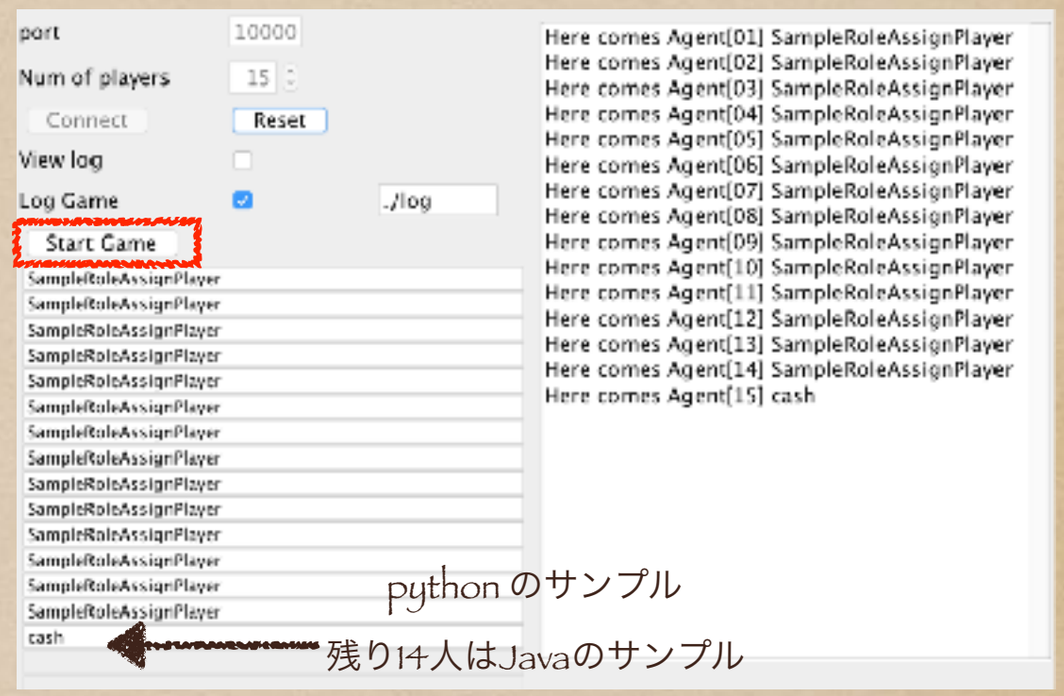
# 1.2. AIWolf Serverの起動

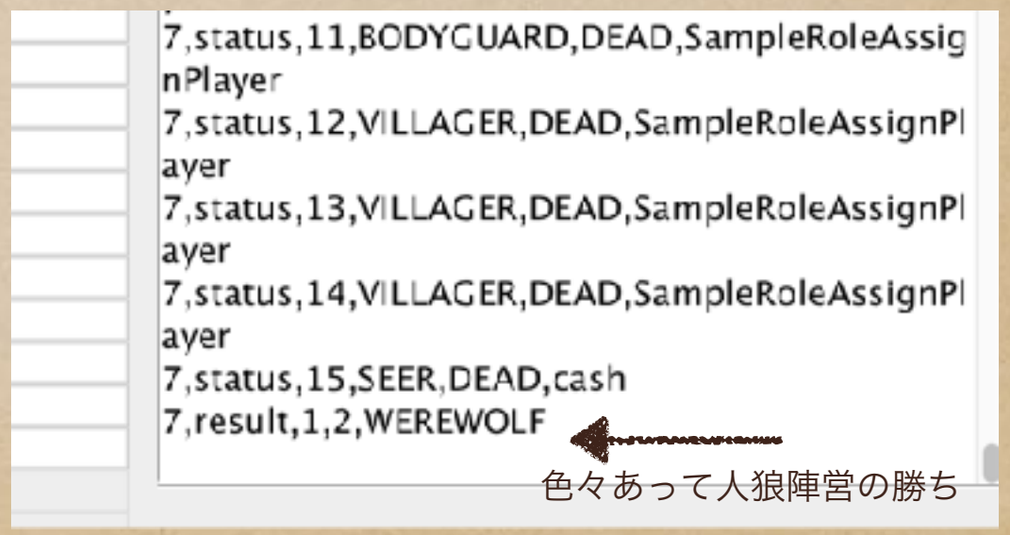
- http://aiwolf.org/server から最新版(執筆時点でAIWolf-ver0.4.9)をダウンロード
  - 適当な場所に展開して、必要であれば以下のファイルに実行権限を付与
    - StartServer.sh, StartGUIClient.sh (Windowsの場合は同名のbatファイル)
- StartServer.shを実行, Connectを押す
- StartGUIClient.shを実行
  - JarFilesでaiwolf-client.jarを選ぶ(初回はOpen File から選択)
  - SampleRoleAssignPlayerを選択、Connect Agent(15人村の場合は14回)



# 1.3. Python Agentの接続

- https://github.com/k-harada/AIWolfPy から最新版(執筆時点でver0.4.9)をダウンロード
  - 適当な場所に展開して、ターミナル等で展開されたディレクトリに移動して、下記を実行
  - ./python_sample.py -h localhost -p 10000
- AIWolf Server StarterのStart Gameを押す
  - この画面に高速で何か流れて、十数秒で止まって右下のようになったら成功
  - Start Game を押すとまた対戦開始



# Chap. 2. Agentの作り方

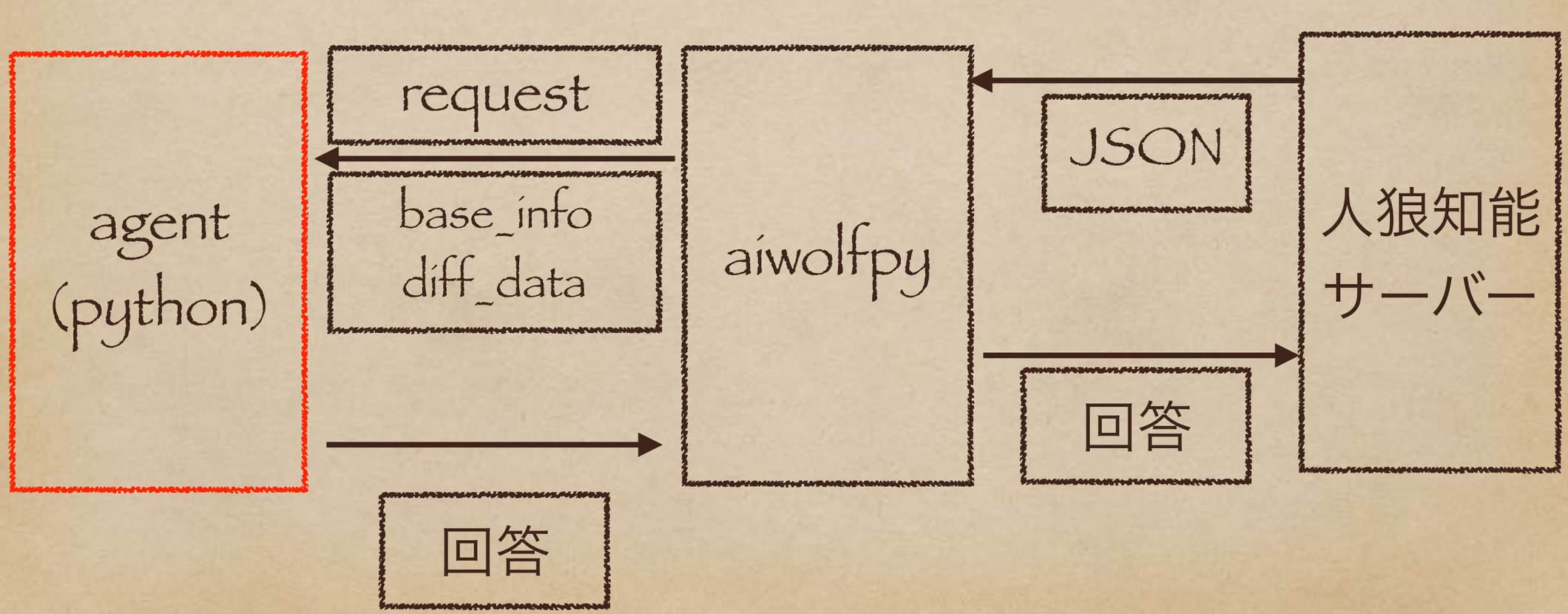
- 2.1. ゲームの流れと実装すべきmethod
  - 2.1.1. ゲームの流れ
  - 2.1.2. 対戦時のデータの流れ
  - 2.1.3. 実装すべきメソッド
- 2.2. データの内容
  - 2.2.1. base_info
  - 2.2.2. diff_data
- 2.3. 過去ログからの機械学習

# 2.1.1. ゲームの流れ

- ゲームは基本的には、下記のPhaseを、いずれかの陣営が勝利条件を満たすまで繰り返します(0日目はwhisperとdivineのみ)
  - talk: turn制で、生存プレイヤー全員が一斉に発話します
  - vote: 処刑投票、一斉投票です。同票の場合は投票結果を通知された上で再投票、それでも同票の場合は最多得票の中でランダムです
  - whisper: 人狼が2人以上生存している場合にできます、このタイミングのみで、talk中にwhisperなどはできません。初日にCOの相談をしたり、襲撃先や翌日の処刑投票の打ち合わせに使ってください
  - attack/divine/guard: 処刑投票結果を知った上で実行します

# 2.1.2. 対戦時のデータの流れ

- サーバーとのやりとりはaiwolfpy.connent_parseを経由し、agentとはメソッドの呼び出しを通じてやりとりします
- 他のエージェントの行動はサーバーから与えられます



# 2.1.3. 実装すべきメソッド(1)

- 最初はpython_simple_sampleをベースにすることをオススメします
  - ver0.4.9からparse機能付きのconnect_parseを推奨とします
- __init__(self)
- getName(self)
  - 登録したチーム名を返してください
- initialize(self, base_info, diff_data, game_setting)
  - __init__ではないので注意してください。ゲームの初期化です。ここでAgentを初期化する必要はないので、強化学習等をやる方はご活用ください

# 2.1.3. 実装すべきメソッド(2)

- update(self, base_info, diff_data, request)
  - Java版と異なり、request(このupdateの次に呼ばれるメソッド)も付いています。通信としては後続込みの2つで1セットなので、制限時間はupdate+次の処理での計測となります。
  - daily_finishというtalk終了時に呼ばれるリクエストには、単にupdateをし、他のメソッドは呼ばないようにしています（次に必ずvoteが呼ばれるので）もし制限時間を気にする等の理由で必要なら、request == 'DAILY_FINISH'のupdateの後ろに処理を追加してください

# 2.1.3. 実装すべきメソッド(3)

- dayStart(self) : return None
  - updateの中で処理をすることを推奨します
  - 前日の色々な結果が来るので、この瞬間に行いたい特殊処理があればどうぞ
- talk(self), whisper(self) : return (text)
- vote(self), attack(self), divine(self), guard(self) : return (AgentIdx)
  - 返すべきAgentIdxは0始まりではなく1始まりなのでご注意ください
- finish(self) : return None
  - ゲーム終了時に呼ばれるメソッドです
  - なぜか2回呼ばれます。強化学習等ご注意ください

# 2.2.1. base_info


```
{'statusMap': {'3': 'DEAD', '15': 'ALIVE', '1': 'DEAD', '5':
'DEAD', '7': 'DEAD', '12': 'ALIVE', '14': 'DEAD', '2': 'DEAD',
'11': 'ALIVE', '9': 'ALIVE', '10': 'DEAD', '8': 'DEAD', '4':
'DEAD', '6': 'DEAD', '13': 'ALIVE'}, 'remainWhisperMap': {},
'day': 8, 'myRole': 'VILLAGER', 'roleMap': {'15': 'VILLAGER'},
'remainTalkMap': {'9': 10, '13': 10, '15': 10, '11': 10, '12':
10}, 'agentIdx': 15}
```


- dictionary形式で連携します（サンプル→）
  - “agentIdx”: あなたのagentのIDです、対戦中は名前は連携されませんが、100戦の間は変更されません
  - “myRole”: あなたの役職です
  - “roleMap”: 役職のdictionaryです、キーは文字列なので注意してください。基本的には自分の役職しかありませんが、他の人の役職がわかる場合（自分が人狼の場合の他の人狼）はここでわかります
  - “statusMap”: Agentの生死を表すdictionaryです、キーは文字列
  - “remainTalkMap”: 生存Agentのその日の残りの発話可能回数を表すdictionaryです
  - “remainWhisperMap”: Agentのその日の残りのwhisperでの発話可能回数です

# 2.2.2. diff_data(1)

- pandas DataFrame形式で差分を連携します
  - 6列[“day”, “type”, “idx”, “turn”, “agent”, “text”]あります
  - “type”によって内容の入り方が変わります
    - “initialize”, “finish”
    - “talk”, “whisper”
    - “vote”, “attack_vote”
    - “execute”,“dead”
    - “attack”, “divine”, “identify”, “guard”

| | agent | day | idx | text | turn | type |
|---|---|---|---|---|---|---|
| 0 | 15 | 6 | 5 | VOTE Agent[15] | 0 | vote |
| 1 | 10 | 6 | 6 | VOTE Agent[10] | 0 | vote |
| 2 | 10 | 6 | 7 | VOTE Agent[10] | 0 | vote |
| 3 | 10 | 6 | 9 | VOTE Agent[10] | 0 | vote |
| 4 | 9 | 6 | 10 | VOTE Agent[09] | 0 | vote |
| 5 | 15 | 6 | 11 | VOTE Agent[15] | 0 | vote |
| 6 | 10 | 6 | 12 | VOTE Agent[10] | 0 | vote |
| 7 | 11 | 6 | 13 | VOTE Agent[11] | 0 | vote |
| 8 | 13 | 6 | 15 | VOTE Agent[13] | 0 | vote |
| 9 | 10 | 6 | 0 | Over | 0 | execute |
| 10 | 5 | 7 | 0 | Over | 0 | dead |

# 2.2.2. diff_data(2)

- type = “initialize”, “finish”
  - agent = idx = agentIdx
  - initializeの場合はday = 0
  - turn = 0
  - text = comingout文 (例： COMINGOUT Agent[01] SEER)

| | agent | day | idx | text | turn | type |
|---|---|---|---|---|---|---|
| 0 | 3 | 10 | 3 | COMINGOUT Agent[03] SEER | 0 | finish |
| 1 | 15 | 10 | 15 | COMINGOUT Agent[15] VILLAGER | 0 | finish |
| 2 | 1 | 10 | 1 | COMINGOUT Agent[01] BODYGUARD | 0 | finish |
| 3 | 5 | 10 | 5 | COMINGOUT Agent[05] VILLAGER | 0 | finish |
| 4 | 7 | 10 | 7 | COMINGOUT Agent[07] VILLAGER | 0 | finish |
| 5 | 12 | 10 | 12 | COMINGOUT Agent[12] VILLAGER | 0 | finish |
| 6 | 14 | 10 | 14 | COMINGOUT Agent[14] VILLAGER | 0 | finish |
| 7 | 2 | 10 | 2 | COMINGOUT Agent[02] MEDIUM | 0 | finish |
| 8 | 11 | 10 | 11 | COMINGOUT Agent[11] VILLAGER | 0 | finish |
| 9 | 9 | 10 | 9 | COMINGOUT Agent[09] WEREWOLF | 0 | finish |
| 10 | 10 | 10 | 10 | COMINGOUT Agent[10] VILLAGER | 0 | finish |
| 11 | 8 | 10 | 8 | COMINGOUT Agent[08] WEREWOLF | 0 | finish |
| 12 | 4 | 10 | 4 | COMINGOUT Agent[04] WEREWOLF | 0 | finish |
| 13 | 6 | 10 | 6 | COMINGOUT Agent[06] POSSESSED | 0 | finish |
| 14 | 13 | 10 | 13 | COMINGOUT Agent[13] VILLAGER | 0 | finish |

- type = “talk”, “whisper”
  - agent = 発話者
  - day = day, idx = talk/whisperのid, turn = talk/whisperのturn
  - text = 発話文そのまま

| | agent | day | idx | text | turn | type |
|---|---|---|---|---|---|---|
| 0 | 15 | 8 | 10 | Over | 2 | talk |
| 1 | 9 | 8 | 11 | Skip | 2 | talk |
| 2 | 11 | 8 | 12 | Skip | 2 | talk |
| 3 | 13 | 8 | 13 | Skip | 2 | talk |
| 4 | 12 | 8 | 14 | Skip | 2 | talk |

# 2.2.2. diff_data(3)

- type = “vote”, “attack_vote”
  - agent = 投票対象, idx = 投票者（紛らわしいので注意）
  - turnは基本的には0, ただし再投票になった場合は1回目の投票のturnが -1
  - text = vote文／attack文
- type = “execute”(処刑), “dead”(朝の死体)
  - agent = 死者
  - idx = 0, turn = 0
  - text = Over

| | agent | day | idx | text | turn | type |
|---|---|---|---|---|---|---|
| 0 | 15 | 6 | 5 | VOTE Agent[15] | 0 | vote |
| 1 | 10 | 6 | 6 | VOTE Agent[10] | 0 | vote |
| 2 | 10 | 6 | 7 | VOTE Agent[10] | 0 | vote |
| 3 | 10 | 6 | 9 | VOTE Agent[10] | 0 | vote |
| 4 | 9 | 6 | 10 | VOTE Agent[09] | 0 | vote |
| 5 | 15 | 6 | 11 | VOTE Agent[15] | 0 | vote |
| 6 | 10 | 6 | 12 | VOTE Agent[10] | 0 | vote |
| 7 | 11 | 6 | 13 | VOTE Agent[11] | 0 | vote |
| 8 | 13 | 6 | 15 | VOTE Agent[13] | 0 | vote |
| 9 | 10 | 6 | 0 | Over | 0 | execute |
| 10 | 5 | 7 | 0 | Over | 0 | dead |

# 2.2.2. diff_data(4)

- type = “divine”, “identify”, “guard”(該当役職のみ、guardは護衛成功を意味しないので注意)
  - agent = 能力の対象, idx = 能力使用者（紛らわしいので注意）
  - turn = 0
  - text = DIVINED / IDENTIFIED / GUARDED 文、占い／霊媒結果はここに入る
  - 例：agent 1 がagent 2を占って人狼だった場合
    - agent = 2, idx = 1, text = DIVINED Agent[02] WEREWOLF
- type = “attack”(襲撃投票の結果の実際の襲撃の内容、襲撃成功を意味しない)
  - agent = 襲撃対象
  - idx = 0, turn = 0
  - text = ATTACK文

# 2.3. 過去ログからの機械学習

- ログの入手方法
  - 過去の大会のログは公式サイトから入手できます(http://aiwolf.org/resource)
  - 大会前に開催される予備予選に参加すると、自分のAgentが参加したゲームのログは、マイページからダウンロードできます
- ログの変換方法
  - aiwolfpy.read_log(file)でaiwolfpyが使用している形式に変換します
    - ただし、全データが入っているため視点がおかしいので、適切に制約をかけてください
  - githubのnotebookディレクトリのサンプルも参考にしてください

# Chap. 3. 大会に参加する方法

- 3.1. 提出ファイルの作成方法
- 3.2. 提出方法

# 3.1. 提出ファイルの作成方法

- javaの場合と同様に、アカウントを作成します
- チーム名をhogeとして、python_simple_sampleをそのまま提出する場合
  - python_simple_sample.pyのmyname = 'cash'をmyname = 'hoge'に変更
  - hogeディレクトリを作成し、その直下にpython_simple_sample.pyとaiwolfpyディレクトリをコピー
  - hogeディレクトリを圧縮してhoge.zipにする
  - ダウンロードしたzipをそのまま出してもダメです

# 3.2. 提出方法

- メモ：なんでもOK
- 言語：Python
- jar/dll/zip
  - hoge.zip
- クラスパス
  - python_simple_sample.py
- エージェント名
  - hoge
- エージェント詳細
  - なんでもOK

# Chap. 4. 大会に参加する方法

- 4.1. aiwolfpyの中身
- 4.2. sampleについて
- 4.3. Tensor5460について

# 4.1. aiwolfpyの中身(1)

- 通信とデータ加工を提供します、実際の呼び方はsimple_sampleを参考にしてください
- __init__.py : パッケージ管理用
- tcpipclient.py : tcp/ip接続機能（旧版、自分でjsonを処理したい方はこちら）
- tcpipclient_parsed.py : tcp/ip接続機能+DataFrame加工（推奨）
- templatetalkfactory.py / templatewhisperfactory.py /contentbuilder.py : 会話構文作成用(contentbuilder.pyの使用を推奨)
- gameinfoparser.py : 通信結果をpandas.DataFrameに集約する
  - 使用しなくても動きますが、自力で通信結果を解読したくないなら使用を推奨します
- read_log.py: サーバーから提供される試合ログ（サイトからダウンロードできる予選等のログを含む）を上記と同様のpandas.DataFrameにする

# 4.1. aiwolfpyの中身(2)

- python_simple_sample.py
  - サンプルエージェント（弱いほう）の実行スクリプトです
  - 真面目に開発するならここからスタートすることをお勧めします
- python_sample.py
  - サンプルエージェント（強いほう）の実行スクリプトです
  - 投票などの行動は直接書いていますが、複雑になってきたら推理機能のように外に書くほうがいいと思います
- aiwolfpy/cash
  - サンプルエージェント（強いほう）用のソースです、Tensor5460もこの中です
  - Predictorはモデル適用です
- notebook
  - 色々やってみたjupyter notebookを置いています、よければ参考にしてください

# 4.2. sampleagentについて

- 色々な計算をしているので、よければ参考にしてください
  - 予備予選で2位(6/24現在)になるくらいは強いです(GAT2016優勝、2016の大会予選2位のものがベース)
  - ただし予備予選中はAgentIdの面でpythonは有利なので、真の実力は割り引いてください
- 強み：tensor5460による高精度の人狼発見ロジック
  - パラメーターは適当ですが人外投票率はトップクラスだと思います
  - notebookを参考に是非機械学習をやってみてください
- 弱み：作戦がほぼなし
  - 人外だと思ったところにVOTE宣言だけしてそのまま投票するので、脅威噛みされやすいようです
  - 人狼の場合に必ず潜伏するので、あまり面白くないかも
  - PPも対応なし。特に5人人狼ではやったほうがいいと思います
  - 対抗は占わない、霊媒ローラーする、処刑できそうなところに投票する、など、色々と工夫のしがいがあると思います

# 4.3.1. Tensor5460の概要

- 15人中3人狼1狂人の配置は5460通り
- 試合中に取得した特徴量(15*K, 15*15*L)を5460通りそれぞれにおいて、どういう特徴になるかを計算する
  - 例：Agent1, 2が占いCOした場合、5460通りの大半のケースにおいて村陣営から占い2COでおかしい
- Tensor5460は例えば15人の占いCO有無を(15次元ベクトル、バイナリ)から各ケースにおいて村、狼、狂の3種が何人COしているか(5460*3)に変換する（実際には同様の特徴がK種類あれば15*Kの行列を5460*3*Kのテンソルにする）
- 例：Agent1がAgent2に処刑投票したから、Agent1が人狼かつAgent2が人狼のケースはちょっと違和感がある
- この場合は15*15のマトリックスを（村、狼、狂）→（村、狼、狂）の3*3パターンに何票入っているか(5460*3*3)に変換する（実際には15*15*Lを5460*3*3*L）

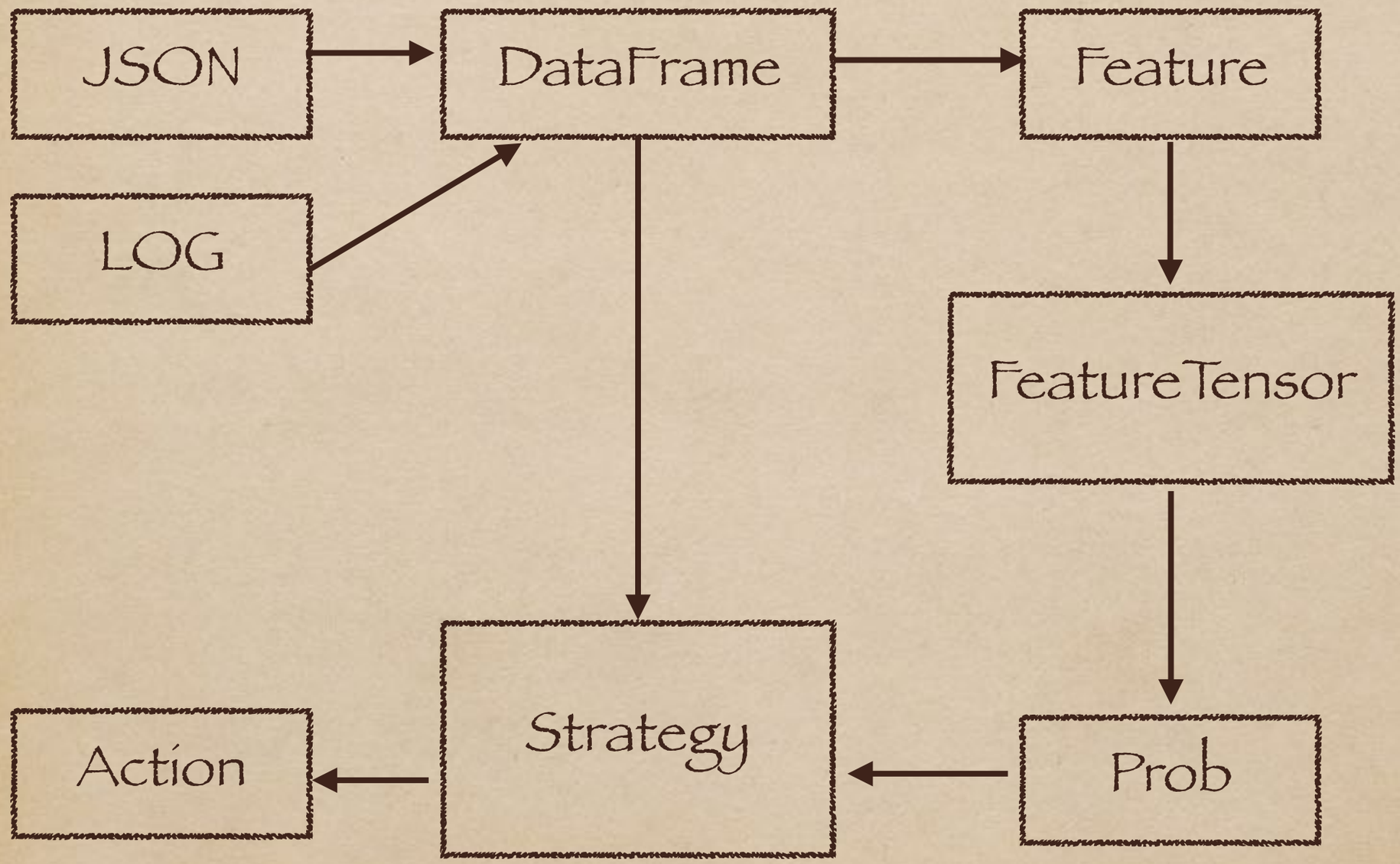
# 4.3.2. Tensor5460の使い方

- Agentの__init__を呼ぶときに__init__で呼び出す
  - 計算装置のイメージです
- 特徴量(15*K, 15*15*L)をTensor5460.apply_tensor_df()に渡すと、5460行の DataFrameを返します
  - numpy.ndarrayを返すメソッドもあります
- ここに統計モデルなどを適用して、各パターンの確率などを計算すると良いです
- RNN, LSTMとかやりたい人は枠組みが違うので自力で頑張ってください
  - もし作ったら公開します

# 5. 今後の開発予定

- 2017夏の大会終了まで
  - バグfixや細かい改善は要望等があればやります
  - サンプルAgentのこれ以上の強化はしない予定です
    - javaのsampleがもっと強くなれば考えますが。。。
- 大会終了後
  - Agentは公開します
- ご質問、ご要望はGithub(k-harada)またはメーリングリストまでお願いします
- pythonでの参加者はもちろん、パッケージ開発者も絶賛募集中です

# 参考：概念図


JSON
LOG
DataFrame
Feature
FeatureTensor
Strategy
Action
Prob