pytlas Documentation

Release 4.0.6

Julien LEICHER

Aug 27, 2019

Contents

pytlas is an open-source **python 3** assistant library built for people and made to be super easy to setup and understand.

Its goal is to make easy to **map natural language sentences** to **python function handlers**. It also manages a conversation with the help of a finite state machine.

Ever wanted to develop your own Alexa, Siri or Google Assistant and host it yourself? This is possible!

Warning: **pytlas** being a library, it does not handle speech recognition and synthesis. It only handles text inputs. If you want it to be able to interact with the voice, you must write a *Client* which will call the library internally.

CHAPTER 1

Getting started

Here is the basic steps to get you started quickly with **pytlas**.

1.1 Installation

There's multiple way to install pytlas. You're free to pick the one that better fit your needs.

Note: Whatever installation you choose, you may need additional setup related to the interpreter you have decided to use. See *Choosing your interpreter* below for more information.

In the following examples, pytlas is installed with the extra require **snips** which is the official interpreter used by pytlas.

Warning: On a Raspberry PI, if you wish to install *snips*, you will have to follow the instructions here to install *rust* and *setuptools_rust* before running below commands.

1.1.1 From pypi

```
$ pip install pytlas[snips]
```

Note: The *[snips]* mention here represents an extra require and will download *snips_nlu* for you.

1.1.2 From source

```
$ git clone https://github.com/atlassistant/pytlas.git
$ cd pytlas
$ pip install -e .[snips]
```

1.1.3 Choosing your interpreter

in order to understand natural language, pytlas is backed by **interpreters** which may need additional installation steps.

snips

The official interpreter supported use the fantastic snips-nlu python library.

Given the language you want your assistant to understand, you may need to download additional resources using the following command:

```
$ snips-nlu download en
```

to download only needed english resources or:

```
$ snips-nlu download-all-languages
```

to download all language resources.

1.2 Usage

1.2.1 Using the pytlas CLI

pytlas include a basic CLI interface to interact with the system.

This line will start the pytlas REPL with skills located in the example/skills/ directory (in the git repository). It will load all data and fit the engine before starting the interactive prompt. The *-c cache/* lets the interpreter save its trained data to this folder to speed up the loading at the next launch if training data has not changed since.

```
$ cd example
$ pytlas -c cache/ repl
```

1.2.2 Using the library

Here is a snippet which cover the basics of using pytlas inside your own program :

```
# pytlas is fairly easy to understand.
# It will take raw user inputs, parse them and call appropriate handlers with
# parsed slots values. It will also manage the conversation states so skills can
# ask for user inputs if they need to.

from pytlas import Agent, intent, training
from pytlas.interpreters.snips import SnipsInterpreter
import os
```

(continues on next page)

(continued from previous page)

```
# Here, we register a sentence as training data for the specified language
# Those training sample are written using a simple DSL named chatl. It make it
# back-end agnostic and is much more readable than raw dataset needed by NLU
# engines.
#
# Those data will be parsed by `pychatl` to output the correct dataset use for the fit
# part.

@training('en')
def en_data(): return """
%[lights_on]
  turn the @[room]'s lights on would you
  turn lights on in the @[room] and @[room]
  lights on in @[room] and @[room] please
  turn on the lights in @[room]
  turn the lights on in @[room]
  enlight me in @[room]
  lights on in @[room] and [room]

~[basement]
  cellar

@[room](extensible=false)
  living room
  kitchen
  bedroom
  ~[basement]

"""

# Here we are registering a function (with the intent decorator) as an handler
# for the intent 'lights_on'.
#
# So when a user input will be parsed as a 'lights_on' intent by the interpreter,
# this handler will be called with a special `Request` object which contains the
# agent (which triggered this handler) and the intent with its slots.

@intent('lights_on')
def on_intent_lights_on(request):

  # With the request object, we can communicate back with the `answer` method
  # or the `ask` method if we need more user input. Here we are joining on each
  # slot `value` because a slot can have multiple values.

  # This is where you should call the actual code managing the lights

  request.agent.answer('Turning lights on in %s' % ', '.join([v.value for v in␣
↪request.intent.slot('room')]))

  # When using the `answer` method, you should call the `done` method as well. This is
  # useful because a skill could communicate multiple answers at different intervals
  # (ie. when fetching the information elsewhere).

  return request.agent.done()

class Client:
```

(continues on next page)

(continued from previous page)

```
  """This client is used as a model for an agent. It will receive lifecycle events
  raised by the agent.
  """

  def on_answer(self, text, cards, **meta):
    print (text)

  def on_ask(self, slot, text, choices, **meta):
    print (text)

if __name__ == '__main__':

  # The last piece is the `Interpreter`. This is the part responsible for human
  # language parsing. It parses raw human sentences into something more useful for
  # the program.

  interpreter = SnipsInterpreter('en', cache_directory=os.path.join(os.path.dirname(__
↪file__), 'cache'))

  # Train the interpreter using training data register with the `training` decorator
  # or `pytlas.training.register` function.

  interpreter.fit_from_skill_data()

  # The `Agent` uses the model given to call appropriate lifecycle hooks.

  agent = Agent(interpreter, model=Client())

  # With this next line, this is what happenned:
  #
  # - The message is parsed by the `SnipsInterpreter`
  # - A 'lights_on' intents is retrieved and contains 'kitchen' and 'bedroom' as the
↪'room' slot values
  # - Since the `Agent` is asleep, it will transition to the 'lights_on' state␣
↪immediately
  # - Transitioning to this state call the appropriate handler (at the beginning of␣
↪this file)
  # - 'Turning lights on in kitchen, bedroom' is printed to the terminal by the␣
↪`Client.on_answer` defined above
  # - `done` is called by the skill so the agent transitions back to the 'asleep'␣
↪state

  agent.parse('turn the lights on in kitchen and bedroom please')
```

CHAPTER 2

Writing skills

Writing a skill for **pytlas** is as easy as creating a python module, writing some code that use *pytlas* members and putting it in the skills directory of your instance.

There's only two parts that your skill should always define to make it work, *Training* and *Handler*.

Note: For the rest of this section, I assumed the following directory structure:

```
- skills/
  - your_awesome_skill/
    - __init__.py
```

and we're going to work directly in the *__init__.py* file.

2.1 Training

You should always start by defining example sentence of how a user might trigger your code.

It allows your skill to define which sentences will trigger specific intents so you must provide enough data for it to understand patterns.

Note: You should define training data in all languages that you wish to support in your skill.

2.1.1 Format

It uses a specific **interpreter agnostic format** called chatl that I also maintain. Its goal is to be easy to write and read by humans.

This tiny DSL will be transformed to a format understandable by your interpreter of choice.

So, going back to our skill, let's define some training data:

```
from pytlas import training

@training('en')
def my_data(): return """
%[lights_on]
  turn the @[room]'s lights on would you
  turn lights on in the @[room]
  lights on in @[room] please
  turn on the lights in @[room]
  turn the lights on in @[room]
  enlight me in @[room]

%[lights_off]
  turn the @[room]'s lights off would you
  turn lights off in the @[room]
  lights off in @[room] please
  turn off the lights in @[room]
  turn the lights off in @[room]

~[basement]
  cellar

@[room](extensible=false)
  living room
  kitchen
  bedroom
  ~[basement]
"""
```

Where *%[lights_on]* and *%[lights_off]* define intents, *@[room]* is an entity and *~[basement]* is a synonym.

2.1.2 Best practices

Here is some thoughts about making great training data.

- Use lowercase
- Avoid punctuation
- Give at least 10 sentences per intent
- Provide variety in your samples

2.2 Handler

Handlers are python code that will be executed when an intent has been recognized.

Your handler will received one and only arguments, a *Request* instance which represents the agent and the context for which your handler is being called.

2.2.1 Getting started

Note: The agent in the *Request* is a proxy which maps to an *Agent* so you have access to everything it exposes. Why a proxy you may ask? Because when the action is cancelled by the user, the request is invalidated so any call through the proxy will be dismissed.

Here the basic code you need to have. Calling *request.agent.done* is mandatory to inform the agent that it should returns to its asleep state.

```
from pytlas import intent

# Remember we have defined this handler in the training section with %[lights_on]

@intent('lights_on')
def my_handler(request):
  return request.agent.done()
```

2.2.2 Retrieving slots

Remember, slots are like function arguments that has been extracted by the interpreter.

Note: In a slot, the *value* property will give you back a representation of what have been parsed by the NLU engine in a meaningful way:

- for durations, it will returns a *dateutil.relativedelta* object
- for moneys and temperatures, it returns a *pytlas.interpreters.slot.UnitValue*
- for percentages, a float between 0 and 1
- for exact time, a *datetime.datetime* object,
- for time ranges, a tuple of *datetime.datetime* objects representing the lower and upper bounds
- for anything else, a string

```
from pytlas import intent

# Remember we have defined a slot @[room] in training sentences

@intent('lights_on')
def my_handler(request):
  rooms = request.intent.slot('room')

  # When you retrieve a slot, it's always a list since you can have multiple␣
↪occurences of an entity in the same sentence

  first = rooms.first()
  last = rooms.last()

  # first and last are SlotValue object, if you want to retrieve their value you␣
↪should use the `value` property

  return request.agent.done()
```

2.2.3 Answering

When you need to show something to the user, you should use the *answer* method.

```
from pytlas import intent

@intent('lights_on')
def my_handler(request):
  room = request.intent.slot('room').first().value

  # Turn the lights on !

  # And say it to the user

  request.agent.answer('Turning lights on in %s' % room)

  # You can also give the text parameter an array of strings.
  # If you do so, pytlas will choose one item randomly. This make it easy
  # to provide some variations for your skill handler.
  # request.agent.answer(['Turning lights on in %s' % room, 'Alright, lights on in %s
↪' % room])

  return request.agent.done()
```

2.2.4 Asking

When you need some informations or slot have not been extracted in the original sentence, you can ask the user to fill them.

```
from pytlas import intent

@intent('lights_on')
def my_handler(request):
  room = request.intent.slot('room')

  if not room:
    # Here we ask the user to fill the 'room' slot. That's the only case when you don
↪'t
    # need to call done yourself.
    # Like in the answer text argument, the ask text argument also accept an array of␣
↪strings and
    # pytlas will choose one randomly to provide to the user.
    return request.agent.ask('room', 'Which room?')

  request.agent.answer('Turning lights on in %s' % room)

  return request.agent.done()
```

2.3 Testing your skill

Once you have developed your skill, you should test it. You can launch the pytlas repl and test it manually or (the prefered approach) use some code to trigger agent state and make assertions about how your skill has answered.

In order to help you do the later approach, there's some utilities in the *pytlas* package itself.

Let's consider this tiny skill:

```
from pytlas import training, intent

@training('en')
def en_data(): return """
%[lights_on]
  turn the @[room]'s lights on would you
  turn lights on in the @[room]
  lights on in @[room] please
  turn on the lights in @[room]
  turn the lights on in @[room]
  enlight me in @[room]

~[basement]
  cellar

@[room](extensible=false)
  living room
  kitchen
  bedroom
  ~[basement]
"""

@intent('lights_on')
def on_lights_on(r):
  rooms = req.intent.slot('room')

  if not rooms:
    return req.agent.ask('room', 'For which rooms?')

  req.agent.answer('Turning lights on in %s' % ', '.join(room.value for room in␣
↪rooms))

  return req.agent.done()
```

2.3.1 Writing tests

In order to make assertions, pytlas use the excellent sure library so let's use them here too.

Now, let's create a file *test_lights.py* next to your skill python file.

Warning: Since *create_skill_agent* uses the *SnipsInterpreter*, you must have *snips-nlu* installed and language resources too. See *snips* for more informations.

```
from sure import expect
from pytlas.testing import create_skill_agent
import os

# Let's instantiate an agent specifically designed to make assertions easier.
# It will fit the data with the SnipsInterpreter so you have pretty much what
# will be used in a real case scenario.
agent = create_skill_agent(os.path.dirname(__file__))
```

(continues on next page)

(continued from previous page)

```
class TestLights:

  def setup(self):
    # Between each tests, resets the model mock so calls are dismissed and we
    # start on a fresh state.
    agent.model.reset()

  def test_it_should_answer_directly_when_room_is_given(self):
    agent.parse('Turn the lights on in the kitchen please')

    # Retrieve the last call on on_answer (you can also give an integer if you have␣
↪multiple calls in your skill).
    # Here `agent.model.on_answer` is a `pytlas.testing.ModelMock` with some␣
↪utilities to make assertions.
    on_answer = agent.model.on_answer.get_call()

    # And make assertions on argument names
    expect(on_answer.text).to.equal('Turning lights on in kitchen')

  def test_it_should_ask_for_room_when_no_one_is_given(self):
    agent.parse('Turn the lights on')

    on_ask = agent.model.on_ask.get_call()

    expect(on_ask.slot).to.equal('room')
    expect(on_ask.text).to.equal('For which rooms?')

    agent.parse('In the bedroom')

    on_answer = agent.model.on_answer.get_call()

    expect(on_answer.text).to.equal('Turning lights on in bedroom')

    # Since it inherits from `MagicMock`, you can use all methods to make assertions
    agent.model.on_done.assert_called()
```

2.3.2 Launching tests

In order to launch tests, pytlas uses nose, so you may use it to test your skill too.

In your skill directory, just launch the following command:

```
$ python -m nose
..
----------------------------------------------------------------------
Ran 2 tests in 0.016s

OK
```

2.4 Translations

Translating a skill is pretty easy. It works the same as training data. You'll just have to use the decorator on a method which returns a dictionary representing keys and associated translations.

```
from pytlas import translations, intent

@translations('fr')
def my_translations(): return {
  'Turning lights on in %s': "J'allume les lumières dans %s",
}

# Training data are not shown here

@intent('lights_on')
def my_handler(request):
  room = request.intent.slot('room').first().value

  # Do something

  # Here, just use the `request._` to translate the string
  # If you wish to localize a date, we got you covered with the `request._d`

  request.agent.answer(request._('Turning lights on in %s') % room)

  return request.agent.done()
```

2.5 Metadata

Metadata are entirely optional and are mostly use by the tiny skill manager of pytlas to list loaded skills with associated informations.

As a best practice however, you must include it in your skill to provide at least a description of what your skill do and what settings are expected.

```
from pytlas import meta, translations

# Here the function register will be called with a function used to translate
# a string.
# If you prefer, you can also returns a `pytlas.skill.Meta` instance.

@meta()
def register(_): return {
  'name': _('lights'),
  'description': _('Control some lights'),
  'version': '1.0.0',
  'author': 'Julien LEICHER',
  'settings': ['LIGHTS_SETTING_ONE'],
}

@translations('fr')
def fr_translations(): return {
  'lights': 'lumières',
  'Control some lights': 'Contrôle des lumières',
}
```

2.6 Request

It's the object that your handler will receive as it's only argument.

More to come... For now, you better check the source code.

2.7 Hooks

Hooks represents lifecycle events your skill can listen to. At the moment, only 2 hooks are available as decorators.

```
from pytlas import on_agent_created, on_agent_destroyed

@on_agent_created()
def do_some_setup_for(agent):
  # It will be called on agent startup so you have a change to do some
  # stuff in your skill.
  print (agent.meta)

@on_agent_destroyed()
def do_some_cleanup_for(agent):
  # It will be called upon agent destruction.
  print ('Some cleanup stuff could go here!')
```

2.8 Settings

Settings provides a basic handling to enable the user to configure the system.

When you use the *pytlas repl*, you can provide a config file that will be parsed using ConfigParser. If an environment variable matching *SECTION_SETTING* is available, it will override the config file value.

pytlas.settings also provides a wide range of methods to retrieve configuration values casted to a particular type.

```
from pytlas import settings, intent

# Load a setting file
settings.load('file/path/pytlas.conf')

# Get a string
settings.get('openweather_key', 'a default value', section='pytlas.weather')

# If you have exported the env PYTLAS_WEATHER_OPENWEATHER_KEY=apikey, then this
# function will returns "apikey"

# Arguments are the same for other helpers
# settings.getint
# settings.getfloat
# settings.getlist
# settings.getbool
# settings.getpath

# You can also programatically set a setting
settings.set('a key', 'your value', section='pytlas.weather')
```

(continues on next page)

(continued from previous page)

```
@intent('my_intent')
def my_handler(r):
  # Inside an handler, you can pass agent metadata to `additional_lookup`
  # With this call, agent meta will take precedence over env and file settings, this
  # is useful to allow agent to override some settings such as api keys.
  #
  # If you do so, the `additional_lookup` will be check as if it was the env by using
  # the key PYTLAS_WEATHER_OPENWEATHER_KEY.
  settings.get('openweather_key', 'a default value', section='pytlas.weather',␣
↪additional_lookup=r.agent.meta)
```

2.9 Context

Context are important when dealing with complex skills. It allows you to define in which case your intent should be recognized.

In order to declare a context, you just to have to define your intent with *valid_context/your_intent* and use the *request.agent.context* method to switch at runtime. Here is an example.

```
from pytlas import training, intent

@training('en')
def en_training(): return """
%[start_intent]
  start something right now
  please start a context
  let's dance

%[started_intent/say]
  say something
  talk to me
"""

@intent('start_intent')
def start_handler(request):
  # This line will switch to the context `started_intent` which means that
  # the interpreter will now be able to recognize the `started_intent/say` intent
  # we define earlier.
  #
  # Till we switch to this context, `started_intent/say` could not be triggered.
  request.agent.context('started_intent')

  return request.agent.done()

@intent('started_intent/say')
def say(request)
  request.agent.answer('Hey!')

  # Switch to the root context which is the None one so this handler could not be␣
↪triggered anymore
  request.agent.context(None)

  return request.agent.done()
```

(continues on next page)

(continued from previous page)

```
# You can also override builtin intents such as __fallback__ and __cancel__ for
# your context.
#
# Here the fallback means every sentence not recognized by the interpreter when
# in the `started_intent` context will trigger this handler.
@intent('started_intent/__fallback__')
def fallback(request):
  request.agent.answer('Looks like you said %s' % request.intent.slot('text').first().
↪value)

  return request.agent.done()
```

CHAPTER 3

Managing skills

The PAM (for Pytlas Assistant Manager) module tries to make it easy to add, update and remove skills to and from your pytlas instance.

Note: Since it uses *git* internally to manage skills retrieval and updates, the command should be available in your environment when executing all commands listed below.

3.1 Listing skills

Those methods will list all currently loaded skills with informations taken from the *@meta()* decorator in skills definitions when it founds them.

3.1.1 From CLI

```
$ pytlas skills list
```

3.1.2 From code

```
from pytlas.pam import get_loaded_skills

# Will returns an array of `pytlas.skill.Meta`
skills = get_loaded_skills('en')
```

3.2 Installing skills

Install skills from a Git repository. If no host is given, it will take the value of *PYTLAS_DEFAULT_REPO_URL* which itsef defaults to *https://github.com/*.

3.2.1 From CLI

```
pytlas skills add atlassistant/pytlas-help https://git.yourownserver.com/myorga/my-
↪skill
```

3.2.2 From code

```
from pytlas.pam import install_skills

# The first parameter is the skills directory
# The second parameter is a function to print output messages
skills = install_skills(os.getcwd(), print, 'atlassistant/pytlas-help', 'https://git.
↪yourownserver.com/myorga/my-skill')

# skills now have the list of successfuly installed or updated skills
```

3.3 Updating skills

TODO

3.4 Removing skills

TODO

CHAPTER 4

Core components

In order to develop for pytlas, you should understand how core components fit together to make it understand and call your handlers. If you only want to develop your own skills, you can omit this section and go to *Writing skills*.

The general command flow looks like this:

- The user says **will it rain tomorrow**,
- The user **agent** uses its internal **interpreter** to extract the user intent and slots based on its training data,
- The **agent** call the **skill** handler registered for this specific intent if any,
- The **skill** has now the opportunity to answer or ask something to the user in order to fulfil his request and the agent will use the attached **client** to communicate back with the user.

4.1 Agent

An agent represent the interface between the user and loaded skills. It maintain the conversation state and use an underlying interpreter to understand the user.

The agent is the entry point which will take raw user inputs with its *parse* method and call loaded handlers as needed.

4.1.1 Entry point

parse(*msg*, ***meta*)

This method will use the agent *Interpreter* to extract intents and slots from the raw message given by the user.

Note: More information on *Meta*.

When an intent has been found, it will try to find an handler for this specific intent and call it. It will then manage the conversation, handle cancel and fallback intents and communicate back to the user using its internal *Client*.

4.1.2 From a skill

From a skill perspective, here are the method you will use.

answer(*text*, *cards=None*, ***meta*)
Answer something to the user.

ask(*slot*, *text*, *choices=None*, ***meta*)
Ask for a slot value to the user.

done(*require_input=False*)
Inform the agent that a skill has done its work and it should returns in its asleep state.

context(*context_name*)
Change the current *Context*.

4.2 Interpreter

Interpreter allow pytlas to categorize user intents and to extract slots from raw text. Whatever interpreter you decide to use, it will need training data to be able to understand what's the user intent behind an input sentence.

4.2.1 Intent

An intent represents a user intention.

For example, when I say *what's the weather like?*, my intent is something as *get weather*. When I say *please tell me what's the weather like today*, it maps to the same intent *get weather*.

4.2.2 Slot

A slot is like a parameter value for a function. It represents an entity in the context of an intent.

So when I say *what's the weather like in Paris?*, my intent is *get weather* and the slot *city* should be *Paris*.

4.2.3 Implementing a custom interpreter

If you wish to implement your own interpreter, you must at least extends from *pytlas.interpreters.Interpreter* and implement those methods.

Note: When creating *SlotValue* instance to represent a slot, always remember to sets a value in a meaningfull python representation. See *Retrieving slots* to see what's expected by developers.

---

fit(*data*)
Fit the interpreter with training data.

parse(*msg*, *scopes=None*)
Parse a raw message and returns an intents list. *scopes* is an optional list of allowed intent names.

parse_slot(*intent*, *slot*, *msg*)
Parse a slot for a given context.

4.3 Skill

Skill are where you, as a developer, will spend most your time, see *Writing skills* for more info.

Basically, it's just a python module which uses *pytlas* decorators to register some specific components on the running environment.

4.4 Client

A client is a thin layer used by an agent to communicate with the user. It can be anything such as a tiny CLI (as the one provided), a WebSocket server or a connected speaker.

When provided to an agent, some specific members will be called by the agent on specific lifecycle events:

on_answer(*text*, *cards*, *raw_text*, ***meta*)
: Called when the skill answer something to the user. *cards* is a list of *pytlas.Card* which represents informations that should be presented to the user if possible. Your client should always handle the *text* property at least.

on_ask(*slot*, *text*, *choices*, *raw_text*, ***meta*)
: Called when the skill need some user inputs for the given *slot*. *choices* if set, represents a list of available choices.

on_thinking()
: Called when the agent has called a skill which is handling the request.

on_done(*require_input*)
: Called when a skill has done its work and the agent is going back to the asleep state.

on_context(*context_name*)
: Called when the agent context has changed.

4.5 Meta

When working with **pytlas**, you may find metadata in different places.

Especially in:

- The agent *__init__*, *answer*, *ask* and *parse* methods,
- The *Intent* class

Those metadata represents any non consumed keyword parameters. They are pretty useful when you need to provide additional information but should never be considered mandatory.

Here is a code example for a skill:

```
from pytlas import intent

@intent('get_weather')
def on_weather(r):
  lat = r.intent.meta.get('latitude')
  lng = r.intent.meta.get('longitude')

  if lat and lng:
    # Search using the user position
  else:
```

(continues on next page)

(continued from previous page)

```
    name = r.intent.slot('city').first().value

    if not name:
      return r.agent.ask('city', 'For which city?')

    # Search using a city name

  return r.agent.done()
```

With this definition, if I call the *parse* method with some meta, it will handle my position, else, it will fallback to search the weather for a city:

```
from pytlas import Agent

agent = Agent() # In the real world, you should provide an interpreter and a client

# Meta here will be added to the parsed intent
agent.parse("What's the weather like", latitude=49, longitude=1)

# Will fallback to the city one
agent.parse("What's the weather like in Paris")
```

Index