# pytlas Documentation

***Release 5.1.3***

**Julien LEICHER**

**Feb 07, 2020**

# 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 to enable back and forth communications between a user and its agent.

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 *Interpreter* which may need additional installation steps.

#### snips

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

Given the language you want your assistant to understand, it will need to download additional resources. Fortunately, you don't have to do it manually since *v5.0.0*, pytlas will automatically try to download them when fitting the interpreter with a language it doesn't already know.

## 1.2 Usage

### 1.2.1 Using the pytlas CLI

pytlas include a basic CLI interface to interact with the system that you may use when developping skills.

After cloning the git repository, this line will start the pytlas REPL by using the configuration file *example/pytlas.ini*. It will load all data and fit the engine before starting the interactive prompt.

```
$ cd example
$ pytlas -c pytlas.ini 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.understanding.snips import SnipsInterpreter
import os

# 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.
```

(continues on next page)

(continued from previous page)

```
@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:
  """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)
```

(continues on next page)

(continued from previous page)

```
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 loaded by your instance (when *Using the pytlas CLI*).

There's only two parts that your skill should always define to make it work correctly, *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 sentences of how a user might trigger your code. The *Interpreter* will use them to train and extract meaningful content on unknown inputs.

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.

**Warning:** At runtime, all trainings will be merged into a single dictionary. So don't forget to namespace your intents and entities when that makes sense.

### 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 Builtin intents

pytlas does not ship with training data at all. There's currently one intent which needs data: *__cancel__*. You should provide training data to recognize this intent, something as simple as this:

```
from pytlas import training

@training('en')
def en_data(): return """
%[__cancel__]
  cancel
  abandon the command
"""
```

### 2.1.3 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 (with or without slots defined for example)
- Use optional synonyms (with ~[your synonym?] in intents)

## 2.2 Handler

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

Your handler will received one and only argument, 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 intent 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.understanding.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. Once filled by the user, your handler will be called again with the updated slots.

```
from pytlas import intent

@intent('lights_on')
```

(continues on next page)

(continued from previous page)

```
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()
```

### 2.2.5 Builtin intents

For now, there's only one builtin intent that you want to handle which is *__fallback__*. It will be called if an intent has been recognized but no handler have been found to fulfill the request.

## 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')
```

(continues on next page)

(continued from previous page)

```
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__))

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()
```

(continues on next page)

(continued from previous page)

```
    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 way 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 and use `pytlas.
↪skill.Setting` instance in the `settings` property.

@meta()
def register(_): return {
  'name': _('lights'),
  'description': _('Control some lights'),
  'version': '1.0.0',
  'author': 'Julien LEICHER',
  'settings': [
    'lights.setting_one', # represents the 'setting_one' key in the 'lights' section
  ],
}

@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.

**class** pytlas.conversing.request.**Request**(*agent: Agent*, *intent: pytlas.understanding.intent.Intent*, *module_translations: Dict[str, str] = None*)

Tiny wrapper which represents a request sent to a skill handler.

_(*text: str*) → str

Gets the translated value of the given text.

**Parameters** **text** (*str*) – Text to translate

**Returns** Translated text or source text if no translation has been found

**Return type** str

**_d**(*date: datetime.datetime*, *date_only=False*, *time_only=False*, ***options*) → str

Helper to localize given date using the agent current language.

**Parameters**

- **date** (*datetime*) – Date to format accordingly to the user language
- **date_only** (*bool*) – Only format the date part

- **time_only** (*bool*) – Only format the time part
- **options** (*dict*) – Additional options such as *format* to give to Babel

**Returns** Localized string representing the date

**Return type** str

**agent = None**
Agent proxy used to communicate back with the agent

**id = None**
Unique id of the request

**intent = None**
Intent associated with the request

**lang = None**
Request language as extracted from the agent

## 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 facility to enable developers to retrieve settings value.

The *pytlas.settings* module exposes a *SettingsStore* class and a global instance of this class in its *CONFIG* property.

The *SettingsStore* read settings from 3 sources:

- A ConfigParser instance
- System environment variables
- *additional_lookup* property used at construction

So when you request a value from a store object for a section **pytlas** and a key **my_setting**, it will first try to find a key in *additional_lookup* matching *PYTLAS_MY_SETTING*, if not found, it will look for the same key in the OS environment variables and if it's still can't find a match, it will look in the *ConfigParser* instance for the section and the key provided.

This order make it easy to override config file settings by using environment variables or, as seen below, using agent metadata.

The store also provides a wide range of methods to retrieve configuration values casted to a particular type.

```
from pytlas import intent
from pytlas.settings import CONFIG

# Load a setting file
CONFIG.load_from_file('file/path/pytlas.ini')

# Get a string
CONFIG.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
# CONFIG.getint
# CONFIG.getfloat
# CONFIG.getlist
# CONFIG.getbool
# CONFIG.getpath

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

@intent('my_intent')
def my_handler(r):
  # Inside an handler, you can use the `agent.settings` property which is a␣
↪`SettingsStore`
  # instance extending the global one with the agent metadata.
  #
  # It is useful to allow agent to override some settings such as api keys.
  #
  # The following line will returns the settings from agent meta first, if not found,
  # from env variables and if still not found, from the loaded config file.
  r.agent.settings.get('openweather_key', 'a default value', section='pytlas.weather')
```

## 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
```

(continues on next page)

(continued from previous page)

```
"""

@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()

# 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 pytlas *SkillsManager* class make it easy to add, update and remove skills to and from your pytlas skills directory.

**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 Using the CLI

### 3.1.1 Listing

List all loaded skills metadata and tries to translate them.

```
$ pytlas skills list
```

### 3.1.2 Installing

Install one or more skills from a git repository. If you use a relative name such as *owner/repo*, it will be resolved as *https://github.com/owner/repo* but you can use an absolute URL such as *https://gitlab.com/owner/repo*.

```
$ pytlas skills add atlassistant/pytlas-weather atlassistant/pytlas-help
```

### 3.1.3 Updating

Updates one, more, or all skills.

```
$ pytlas skills update atlassistant/pytlas-weather atlassistant/pytlas-help
$ pytlas skills update # Will try to update all skills in the skills directory
```

### 3.1.4 Removing

Remove one or more skills.

```
$ pytlas skills remove atlassistant/pytlas-weather atlassistant/pytlas-help
```

## 3.2 Using the SkillsManager class

**class** pytlas.supporting.**SkillsManager**(*directory: str*, *lang='en'*, *default_git_url='https://github.com/'*, *handlers_store: pytlas.handling.skill.HandlersStore = None*, *metas_store: pytlas.handling.skill.MetasStore = None*)

The SkillsManager handles skill installation, updates, listing and removal. It can be used with the built-in CLI or used as a library.

**get**() → List[pytlas.handling.skill.Meta]

Retrieve currently loaded skills. That means you should first start to imports them by using the *pytlas.handling.importers* namespace.

**Returns** Skills loaded.

**Return type** list of Meta

**install**(**names*) → Tuple[List[str], List[str]]

Install or update given skill names.

**Parameters** **names** (*list of str*) – Skills to install/update

**Returns** Respectively, successful installs and failed ones

**Return type** (list of str, list of str)

**uninstall**(**names*) → Tuple[List[str], List[str]]

Uninstall given skill names.

**Parameters** **names** (*list of str*) – Skills to remove

**Returns** Respectively, successful removes and failed ones

**Return type** (list of str, list of str)

**update**(**names*) → Tuple[List[str], List[str]]

Update given skill names.

**Parameters** **names** (*list of str*) – Skills to update, if no one is given, all skills will be updated

**Returns** Respectively, successful updates and failed ones

**Return type** (list of str, list of str)

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 Understanding

The understanding domain groups all thing related to the understanding of user intents.

### 4.1.1 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.

#### 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*.

#### 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*.

#### 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.

---

`Interpreter.`**`fit`**(*data: dict*) → None
Fit the interpreter with given data.

**Parameters** **`data`** (*dict*) – Training data

`Interpreter.`**`parse`**(*msg: str*, *scopes: List[str] = None*) → List[pytlas.understanding.intent.Intent]
Parses the given raw message and returns parsed intents.

**Parameters**

- **`msg`** (*str*) – Message to parse
- **`scopes`** (*list of str*) – Optional list of scopes used to restrict parsed intents

**Returns** Parsed intents

**Return type** list of Intent

`Interpreter.`**`parse_slot`**(*intent: str*, *slot: str*, *msg: str*) → List[pytlas.understanding.slot.SlotValue]
Parses the given raw message to extract a slot matching given criterias.

**Parameters**

- **`intent`** (*str*) – Name of the current intent
- **`slot`** (*str*) – Name of the current slot to extract
- **`msg`** (*str*) – Raw message to parse

**Returns** Slot values extracted

**Return type** list of SlotValue

### 4.1.2 Trainings store

All training data are registered on a *TrainingsStore* instance, mostly using the *training* decorator.

**`class`** `pytlas.understanding.`**`TrainingsStore`**(*data: dict = None*)
Contains training data.

**`all`**(*lang: str*) → Dict[str, str]
Retrieve all training data in the given language.

It will evaluate all register functions for the given language.

**Parameters** **`lang`** (*str*) – Language to get

**Returns** Dictionary with package name as key and training DSL string as value

**Return type** dict

**get**(*package: str*, *lang: str*) → str
Retrieve training data for a particular package in the given language.

It will evaluate all register functions for the given language.

**Parameters**

- **package** (*str*) – Pacjage
- **lang** (*str*) – Language to get

**Returns** Training data

**Return type** str

**register**(*lang: str*, *func: Callable*, *package: str = None*) → None
Register training data written using the chatl DSL language into the system.

**Parameters**

- **lang** (*str*) – Language for which the training has been made for
- **func** (*func*) – Function to call to return training data written using the chatl DSL
- **package** (*str*) – Optional package name (usually __package__), if not given pytlas will try to determine it based on the call stack

## 4.2 Handling

The handling domain enables skills to register their data such as **meta**, **handlers** and **translations**.

This is where you, as a developer, will spend most of your time, see *Writing skills* for more info.

Basically, you will just declare a python module and use *pytlas* decorators to register some specific components on the running environment.

### 4.2.1 Handlers store

Handlers are register on an instance of an *HandlersStore*, mostly using the *intent* decorator.

**class** pytlas.handling.**HandlersStore**(*data: dict = None*)
Holds skill handlers.

**get**(*intent_name: str*) → Callable
Try to retrieve the handler associated with a particular intent.

**Parameters** **intent_name** (*str*) – Intent to search

**Returns** Handler if found, None otherwise

**Return type** callable

**register**(*intent_name: str*, *func: Callable*, *package: str = None*) → None
Register an intent handler.

**Parameters**

- **intent_name** (*str*) – Name of the intent to handle

- **func** (*callable*) – Handler to be called when the intent is triggered
- **package** (*str*) – Optional package name (usually __package__), if not given pytlas will try to determine it based on the call stack

### 4.2.2 Metas store

Skill meta are registered on a *MetasStore*, mostly using the *meta* decorator.

**class** pytlas.handling.**MetasStore**(*translations_store: pytlas.handling.localization.TranslationsStore = None*, *data: dict = None*)

Hold skill metadatas.

**all**(*lang: str*) → List[pytlas.handling.skill.Meta]
Retrieve all registered meta in the given language.

**Parameters** **lang** (*str*) – Language to use

**Returns** Registered Meta

**Return type** list of Meta

**get**(*package: str*, *lang: str*) → pytlas.handling.skill.Meta
Retrieve a meta for the given package.

**Parameters**

- **package** (*str*) – Package name to retrieve
- **lang** (*str*) – Lang for which you want to retrieve the skill Meta

**Returns** Meta instance or None if not found

**Return type** Meta

**register**(*func: Callable*, *package: str = None*) → None
Register skill package metadata

**Parameters**

- **func** (*func*) – Function which will be called with a function to translate strings using the package translations at runtime
- **package** (*str*) – Optional package name (usually __package__), if not given pytlas will try to determine it based on the call stack

### 4.2.3 Translations store

Translations are registered on a *TranslationsStore* instance, mostly using the *translations* decorator.

**class** pytlas.handling.**TranslationsStore**(*data: dict = None*)

Translations store which holds all translations used by skills.

**all**(*lang: str*) → Dict[str, Dict[str, str]]
Retrieve all translations for all packages in the given language.

**Parameters** **lang** (*str*) – Language for which we want translations

**Returns** Dictionary of package => translations dict in the given language

**Return type** dict

**get** (*package: str*, *lang: str*) → Dict[str, str]
Retrieve all translations for a particular package.

**Parameters**

- **package** (*str*) – Name of the package
- **lang** (*str*) – Language to retrieve

**Returns** Translations dictionary

**Return type** dict

**register** (*lang: str*, *func: Callable*, *package: str = None*) → None
Register translations into the store.

**Parameters**

- **lang** (*str*) – Language being loaded
- **func** (*func*) – Function to call to load a dictionary of translations
- **package** (*str*) – Optional package name (usually __package__), if not given pytlas will try to determine it based on the call stack

## 4.3 Conversing

The conversing domain use the *Understanding* and *Handling* domains to trigger python actions from parsed intents and maintain a conversation state.

### 4.3.1 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.

#### Entry point

Agent.**parse** (*msg: str*, ***meta*) → None
Parse a raw message.

The interpreter will be used to determine which intent(s) has been formulated by the user and the state machine will move to the appropriate state calling the right skill handler.

It will also handle some specific intents such as the cancel one and ask states.

**Parameters**

- **msg** (*str*) – Raw message to parse
- **meta** (*dict*) – Optional metadata to add to the request object

**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*.

#### From a skill

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

`Agent.`**`answer`**(*text: str*, *cards: Union[pytlas.handling.card.Card, List[pytlas.handling.card.Card]] = None*, ***meta*) → None

Answer something to the user.

**Parameters**

- **`text`** (*str, list*) – Text to show to the user
- **`cards`** (*list, Card*) – List of Card to show if any
- **`meta`** (*dict*) – Any additional data to pass to the handler

`Agent.`**`ask`**(*slot: str, text: Union[str, List[str]], choices: List[str] = None, **meta*) → None

Ask something to the user.

**Parameters**

- **`slot`** (*str*) – Name of the slot asked for
- **`text`** (*str, list*) – Text to show to the user
- **`choices`** (*list*) – List of available choices
- **`meta`** (*dict*) – Any additional data to pass to the handler

`Agent.`**`done`**(*require_input=False*) → None

Done should be called by skills when they are done with their stuff. It enables threaded scenarii. When asking something to the user, you should not call this method since *ask* end the skill immediately.

**Parameters** **`require_input`** (*bool*) – True if additional informations are needed (mostly use to trigger client input)

`Agent.`**`context`**(*context_name: str*) → None

Switch the agent to the given context name. It will populates the list of reachable scopes so the interpreter will only parse intents defined in this scope.

**Parameters** **`context_name`** (*str*) – Name of the context to switch to (None represents the root one)

### 4.3.2 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 (using its *model* property), 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.

## 4.4 Settings

Settings enables all parts of pytlas to read config data and is already covered in *Settings*.

### 4.4.1 Settings store

The settings store holds config data. The global store is available as *pytlas.settings.CONFIG* property.

**class** pytlas.settings.**SettingsStore**(*config: configparser.ConfigParser = None*, *additional_lookup: Dict[str, object] = None*)
Hold application settings with an internal ConfigParser instance. It provides a lot of utility methods to convert settings to particular representations.

Why? You may ask. Because it starts by looking for the given settings into an optional additional lookup dict, if its not found, it will look in the system environment and finally, it will use the ConfigParser instance which is probably loaded from a configuration file.

And since everything in the env are considered as strings, you can use the provided methods to make things easier.

**get**(*setting: str*, *default: str = None*, *section='pytlas'*) → str
Gets a setting value, if an environment variable is defined, it will take precedence over the value hold in the inner config object.

For example, if you got a setting 'lang' in the 'pytlas' section, defining the environment varialbe PYTLAS_LANG will take precedence.

**Parameters**

- **setting** (*str*) – Name of the configuration option
- **default** (*str*) – Fallback value
- **section** (*str*) – Section to look in

**Returns** Value of the setting

**Return type** str

**getbool**(*setting: str*, *default=False*, *section='pytlas'*) → bool
Gets a boolean value for a setting. It uses the *get* under the hood so the same rules applies.

**Parameters**

- **setting** (*str*) – Name of the configuration option
- **default** (*bool*) – Fallback value
- **section** (*str*) – Section to look in

**Returns** Value of the setting

**Return type** bool

**getfloat**(*setting: str*, *default=0.0*, *section='pytlas'*) → float
Gets a float value for a setting. It uses the *get* under the hood so the same rules applies.

**Parameters**

- **setting** (*str*) – Name of the configuration option
- **default** (*float*) – Fallback value
- **section** (*str*) – Section to look in

**Returns** Value of the setting

**Return type** float

**getint** (*setting: str*, *default=0*, *section='pytlas'*) → int
Gets a int value for a setting. It uses the *get* under the hood so the same rules applies.

**Parameters**

- **setting** (*str*) – Name of the configuration option
- **default** (*int*) – Fallback value
- **section** (*str*) – Section to look in

**Returns** Value of the setting

**Return type** int

**getlist** (*setting: str*, *default=[]*, *section='pytlas'*) → list
Gets a list for a setting. It will split values separated by a comma.

It uses the *get* under the hood so the same rules applies.

**Parameters**

- **setting** (*str*) – Name of the configuration option
- **default** (*list*) – Fallback value
- **section** (*str*) – Section to look in

**Returns** Value of the setting

**Return type** list

**getpath** (*setting: str*, *default: str = None*, *section='pytlas'*) → str
Gets an absolute path for a setting. If the value is not an absolute path, it will be resolved based on the loaded config file directory.

It uses the *get* under the hood so the same rules applies.

**Parameters**

- **setting** (*str*) – Name of the configuration option
- **default** (*str*) – Fallback value
- **section** (*str*) – Section to look in

**Returns** Value of the setting

**Return type** str

**load_from_file** (*path: str*) → None
Load settings from a file.

**Parameters** **path** (*str*) – Name of the file to read

**set** (*setting: str*, *value: object*, *section='pytlas'*) → None
Sets a setting value in the *_data* dictionary so it will take precedence over all the others.

Value will be stringified by this method (since all value can be read from env variables).

**Parameters**

- **setting** (*str*) – Setting key to write
- **value** (*object*) – Value to write
- **section** (*str*) – Section to write to

**to_dict** () → Dict[str, str]
Gets a flat dictionary representation of this store (combining settings from the config and the ones in additional_data).

Each keys will be converted to an env one so it can be used in an agent meta for example.

**Returns** Flat dictionary representing this store

**Return type** dict

**write_to_file** (*path: str*) → None
Write this settings store to a file.

**Parameters** **path** (*str*) – Path to a file to store the resut.

## 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:
    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")
```

CHAPTER 5

# Migrating

Sometimes, things should be broken for the well being of the library. This is where you will find such changes to help you update your code accordingly.

## 5.1 From 4.* to 5.0.0

Version 5.0.0 is a big overhaul of how things are layed out in the library and as such introduce a lot of breaking changes if you use the library directly.

If all you do is *from pytlas import training, translations, intent, meta*, then you're good to go, nothing has changed, otherwise, keep reading.

The new structure follow a more domain centric approach:

- **understanding**: Contains interpreters, intent, slots and training stuff
- **handling**: Contains handlers, localization, importers and related stuff
- **conversing**: Contains agent and request
- **supporting**: Contains the skills manager
- **testing**: Contains tests related stuff

The **pytlas** root module now only exposes the most common stuff to make more easy for newcomers to use the library. Each submodules also has a public api represented by the *__init__.py* file.

### 5.1.1 Access to settings

```
# WAS
from pytlas.settings import get, getbool # And other getters

# NOW
from pytlas.settings import CONFIG # Represents the global configuration
```

(continues on next page)

(continued from previous page)

```
# And access it like this
# CONFIG.get CONFIG.getbool

# From a skill, you can now use the agent settings which inherits from the global
# configuration and override keys with the agent metadata.
@intent('my-skill')
def my_skill(request):
  request.agent.settings.get('api', section='openweather')
```

### 5.1.2 Registering without decorators

Since the prefered approach is to use the decorators, direct register are not exposed in the main *__init__.py* and should be imported only when needed.

```
# WAS
from pytlas.skill import register as register_intent, register_metadata
from pytlas.localization  import register as register_translations
from pytlas.training import register as register_training
from pytlas.hooks import register as register_hook
# DOES NOT EXIST ANYMORE, use NOW imports
from pytlas import register_intent, register_metadata, register_translations,␣
↪register_training, register_hook

# NOW
from pytlas.handling.skill import GLOBAL_HANDLERS, GLOBAL_METAS
from pytlas.handling.hooks import GLOBAL_HOOKS
from pytlas.handling.localization import GLOBAL_TRANSLATIONS
from pytlas.understanding.training import GLOBAL_TRAININGS

# And use the `register` method on those global stores (ie. GLOBAL_HANDLERS.register)
```

### 5.1.3 Utils

Utilities methods have been splitted by functions:

- *pytlas.datautils*: Data related helpers
- *pytlas.ioutils*: Input/output helpers
- *pytlas.pkgutils*: Package related helpers

and *read_file* as been updated:

```
# WAS
from pytlas.utils import read_file
read_file('data.dsl', relative_to_file=__file__)

# NOW
from pytlas.ioutils import read_file
# relative_to_file is now relative_to and accept a folder too
read_file('data.dsl', relative_to=__file__)
```

### 5.1.4 Testing

*ModelMock.get_call()* now returns the last call by default.

### 5.1.5 Managing skills from code

```
# WAS
from pytlas.pam import get_loaded_skills, install_skills, update_skills, uninstall_
↪skills

# NOW
from pytlas.supporting import SkillsManager

s = SkillsManager('your_skills_directory')
loaded_skills = s.get()
s.install('atlassistant/pytlas-weather', 'another/skill')
s.update('atlassistant/pytlas-weather', 'another/skill')
s.uninstall('atlassistant/pytlas-weather', 'another/skill')
```

# Index

## Symbols



## A



## C



## D



## F



## G



## H



## I



## L



## M



## O



## P



## R

## S



## T



## U



## W