Companions/Pythonian2

From QRG Wiki

Pythonian2

Location: qrg/companions/v1/pythonian
Load: TODO

go back to Companions

This page describes the new Pythonian agent, which is a replacement of the original Companions/Python-Agent.

The Pythonian agent uses KQML to communicate with Companion. Through KQML, Pythonian can receive and respond to 'ask' and 'achieve' messages. This is a small subset of the KQML performatives, and support for other performatives may easily be added as necessary. For any unsupported performative, the agent will respond with an error message indicating that the performative is not supported.

The Pythonian agent is built using pykqml, a Python package for building a KQML module. It supports reading and writing KQML messages over a socket connection. Since Companion uses some abnormal socket patterns, the KQMLModule in pykqml has been extended to support the current socket usage.

Contents

System requirements

1. Companion.

2. python 3.x (currently being tested in 3.6.5)

3. pykqml 1.1 (`pip install pykqml==1.1` [1] (https://github.com/bgyori/pykqml))

Give it a test run

There are two examples to try out: Test Agent and NLTK Agent.

Test Agent

The Test Agent is being developed to test out Pythonian. You can use it to see how Pythonian works.

To test this out, you will start Companion, start the Test Agent, and then send a message to the Test Agent from Companion:

1. Start Companion, allowing Companion to listen for new agents. If running from source code, try:

```
(cl-user::start-companion :scheme :mixed)
```

2. Start the Test Agent. In the pythonian directory, run:

```
python test_agent.py
```

3. Send a message:

3.1. From the Session Manager, go to the Commands tab and enter the following message:

```
(achieve :receiver TestAgent :content (task :action (test_achieve data)))
```

To verify the agent received this, look in the window where Pyhtonian is running and look for a debug statement saying something like "testing achieve with _input".

3.2. From the listener, enter the following:

```
(agents::send *facilitator* 'TestAgent '(achieve :content (task :action (test_achieve_return nil))))
```

To verify that Companions is receiving a reply from this, you should see a nil printed out in the listener soon after executing the above command.

If the facilitator "actively refuses" the message, this is because the companion is running with the transport scheme set to :queue. Shut everything down and re-launch the companion using

```
(cl-user::start-companion :scheme :mixed)
```

Note: The executable does this by default, so that shouldn't be an issue there.

NLTK Agent

An example of Pythonian constructs a NLTK Agent. This agent allows Companion to access a small set of NLTK functions. For example, you can use the part-of-speech tagging. Since this example uses nltk, you will need to have nltk. It can easily be installed with the following command

```
pip install nltk
```

More information about installing nltk can be found here: [2] (https://www.nltk.org/install.html)

To test this out, you will start Companion, start the NLTK Agent, and then send a message to the NLTK Agent from Companion:

1. Start Companion, allowing Companion to listen for new agents. Recent executables do this already. If running from code, try:

```
(cl-user::start-companion :scheme :mixed)
```

2. Start the NLTK Agent:

```
python nltk_agent.py
```

3. Send a message: From the Session Manager, go to the Commands tab and enter the following message:

```
(ask-one :receiver NLTKAgent :content (pos_tag "What is the meaning of life?" ?tags))
```

Making a Pythonian agent

If you want Companion to have access to some functionality in python, you can create a Pythonian agent to meet that need. Generally, it is suggested you make a new Pythonian agent and not arbitrarily add to an existing one. For example, if you want functionality from spacy, then it is suggested that you create a new spacy agent. An obvious exception to this is if an existing agent is strongly related to the new desired functionality and uses the same python modules. An example of this case would be extending the nltk agent.

To create a new Pythonian agent, you will create a python class that extends Pythonian. In this class, you will implement the desired functionality. This includes importing any modules you want and adding functions that you will make accessible to Companion.

To instantiate the agent when calling this module, add the following at the end:

```
if __name__ == "__main__":
```

```
a = NLTKAgent(host='localhost', port=9000, localPort=8950, debug=True)
```

Of course, customize the arguments to fit your particular need. Also, soon you will be able to supply many of these arguments at run time.

Receiving knowledge from Companion

Companion may communicate with a Pythonian agent by sending KQML messages to it. The head of each message indicates the performative of the message. The sections below describe the performatives that are currently supported.

achieve

If you want a function to be available to Companion via an achieve message, then you will need to add the function to the class to the achieves that the agent knows about. This can be done in the constructor like so:

```
self.add_achieve('pos_tag', self.pos_tag)
```

The return value of this function will be sent to Companion. The return value should be either something than can be converted to a string or a list, which has contents that are either string-able or a list.

ask-one

If you want to make a query to be available to Companion via an ask message, then you will need to add a function to the class to do the query. To recognize that function as a query, you need to add it to the asks the agent knows about. In addition to the name of the ask (the predicate) and the function to be called, you also need to define the query pattern.

```
self.add_ask('subj_verb', self.subj_verb, '(subj_verb ?text ?subj ?verb)')
```

The function that is called when the ask message is received should return a list. The items in the list will be unified with the variables in the pattern in the order that they appear.

subscribe

The subscribe message tells the pythonian agent to respond (with a tell) with any updates to a given query. The query that an agent is subscribing to needs to be one that the pythonian agent is advertising as one that may be subscribed too (see advertise below).

To get updates to the query to be sent, each pythonian agent will need some code to notify the pythonian infrastructure of an update on a query. To do this, the developer of the pythonian agent will need to call `update_query(query, value)` , where *query* is a string representing the query and *value* is the new value (or values) for the variables in the query. The *query* string must be an identical string to what is listed as the pattern in registering an ask function. Using the example given in the ask-one performative above, the *query* string `'(subj_verb ?text ?subj ?verb)'` would correctly match the pattern registered for the ask.

To test that a subscribe is working, first advertise the ask (see advertise below). Then add some code that will call *update_query*. On the Companions side, subscribe to the query. An example of the session-reasoner subscribing to junk mail is the following:

```
(agents::subscribe-to-all *sr* '(test_junk_mail ?x) #'print-reply-callback)
```

tell

When the Companion sends a tell to the pythonian agent, it currently logs the message and sends a None in response.

ping

You do not need to do anything for this, as the Pythonian agent will automatically reply to this message.

Sending knowledge to Companion

One of the next functionalities to be added will be allowing the Pythonian agent to automatically send knowledge to Companion without responding to a query. This can be useful when python is being used to collect data. For example, if python is scraping the web for new knowledge, it can be inserted into the KB.

register

Done, but needs to be documented.

There are two types of advertisements. One advertises a query that some other agent may ask of the pythonian agent. The other advertises a query that some other agent may subscribe to. The code for both are in pyhtonian, but only the latter is currently being used (the function call to advertise the former has been commented out).

Any ask may be advertised for subscription by providing a 4th optional argument of True to the `add_ask` function call.

tell

The tell performative is often used in a response message. This is particularly true for ask-one and achieve responses.

Respond with pattern or respond with bindings...

When responding, the results of the python function are "listify"-ed. This means that the results are converted into some lisp-like list structure. If the variable is a list, it is converted to a KQMLList. If the variable is a dictionary, it is converted to an association list. Tuples of length 2 are also converted to an association list, otherwise are treated the same as a list.

insert

To push new knowledge to Companions, a Pythonian agent may use the insert performative. This will take some data (which is the content of the performative) and send it to Companions. On the Companions side, this will be added to working memory and be added to the KB. If you want to have it only go to WM and not the KB, then there is a WM-only flag.

Note that many use cases should probably use subscriptions instead of just pushing data to Companions. Subscriptions allow an agent to indicate that it is looking for certain pieces of knowledge, and when another

agent acquires that knowledge it sends it off to the subscribing agent. This is ideal for asynchronous interactions between the agents, and a good use case is when a human is interacting with the Companion and you want Companion to go off an do something while the interaction continues.

achieve_on_agent

A Pythonian agent can use achieve_on_agent to send a message to a Companion agent to kick off a command via achieve. It takes some data (a string in the form of a plan call) and the name of the agent that should perform the achieve. Of course, the achieve needs to be defined on the Companion's side.

Managing data types

When receiving a message from Companion, the python agent may expect some arguments to be of a particular type (e.g., Boolean). However, KQML does not support types. As a result, there is no (easy) way to automatically convert to the correct type. Instead, there are a handful of conversion functions available to take an item from KQML and convert to the desired type. Below is a list of these conversion functions and the rules for conversion.

convert_to_boolean

Since KQML is based on lisp, and (at least for now) messages are coming from lisp land (i.e., Companion), we use some lisp conventions to determine how a KQML element should be converted to a Boolean.

If the KQML element is `nil` or `()` then `convert_to_boolean` will return `False`. Otherwise, it returns `True`.

convert_to_int

If the KQML element is a KQMLToken or KQMLString, then the internal data of these tokens are cast to an int. Otherwise, the original value is returned.

convert_to_list

If the KQML element is a KQMLList, then the internal list is returned. It currently does not recurse to do any further automatic conversions.

Feature requests

- ability to kill connection/agent without killing python process (i.e. programmatically)
  - I don't understand what this means
- launch agent from Companions - yes, that works. You just need to update companions-internals to define a domain and agent type.
- update state (displaying in session manager)
- documentation
- sending knowledge to Companion via insert (I think -IR)
  - use case: the agent in Minecraft is constantly observing the world. We want it to be able to send updated information to the Companion without waiting for the ask.

    - insert works, but subscriptions would be the right way to do this
- knowledge--> typed language converter
    - integer? *Done*
    - float?
    - other? List is done.

Bug reports

- insert: WM version does not work
- occasional connection issues