**EchoBot I**

Hello, World!

You'll begin learning how to build chatbots in Python by writing two functions to build the simplest bot possible: EchoBot. EchoBot just responds by replying with the same message it receives.

In this exercise, you'll define a function that responds to a user's message. In the next exercise, you'll complete EchoBot by writing a function to send a message to the bot.

**Instructions**

**100 XP**

* Write a function called respond() with a single parameter message which returns the bot's response. To do this, concatenate the strings "I can hear you! You said: " and message.
* Store the concatenated strings in bot\_message, and return this result.

bot\_template = "BOT : {0}"

user\_template = "USER : {0}"

# Define a function that responds to a user's message: respond

def respond(message):

# Concatenate the user's message to the end of a standard bot respone

bot\_message = "I can hear you! You said: " + message

# Return the result

return bot\_message

# Test function

print(respond("hello!"))

<script.py> output:

I can hear you! You said: hello!

Excellent! Now to build a function to send a message to the bot.

**EchoBot II**

Having written your respond() function, you'll now define a function called send\_message() with a single parameter message which logs the message and the bot's response.

**Instructions**

**100 XP**

* Use the user\_template string's .format() method to include the user's message into the user template, and print the result.
* Call the respond() function with the message passed in and save the result as response.
* Log the bot's response using the bot\_template string's .format() method.
* Send the message "hello" to the bot.

# Create templates

bot\_template = "BOT : {0}"

user\_template = "USER : {0}"

# Define a function that sends a message to the bot: send\_message

def send\_message(message):

# Print user\_template including the user\_message

print(user\_template.format(message))

# Get the bot's response to the message

response = respond(message)

# Print the bot template including the bot's response.

print(bot\_template.format(response))

# Send a message to the bot

send\_message("hello")

<script.py> output:

USER : hello

BOT : I can hear you! You said: hello

Boom! You just built your first chatbot!

**Chitchat**

Now you're going to leave the simple EchoBot behind and create a bot which can answer simple questions such as "What's your name?" and "What's today's weather?"

You'll use a dictionary with these questions as keys and the correct responses as values.

This means the bot will only respond correctly if the message matches *exactly*, which is a big limitation. In later exercises you will create much more robust solutions.

The send\_message() function has already been defined for you, as well as the bot\_template and user\_template variables.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Define a respond() function which takes in a message argument, checks if the message has a pre-defined response, and returns the response in the responses dictionary if there is a match, or the "default" message otherwise.

# Define variables

name = "Greg"

weather = "cloudy"

# Define a dictionary with the predefined responses

responses = {

"what's your name?": "my name is {0}".format(name),

"what's today's weather?": "the weather is {0}".format(weather),

"default": "default message"

}

# Return the matching response if there is one, default otherwise

def respond(message):

# Check if the message is in the responses

if message in responses:

# Return the matching message

bot\_message = responses[message]

else:

# Return the "default" message

bot\_message = responses["default"]

return bot\_message

**Instructions 2/2**

**50 XP**

* [2](javascript:void(0))
* Well Done! Your bot is now able to answer some simple questions. Hit 'Run Code' and call send\_message() (which utilizes the new respond() function) in the IPython Shell with the following questions:
  + "what's today's weather?"
  + "what's your name?"
  + "what's your favorite color?"
* Hit 'Submit Answer' when you are done.

# Define variables

name = "Greg"

weather = "cloudy"

# Define a dictionary with the predefined responses

responses = {

"what's your name?": "my name is {0}".format(name),

"what's today's weather?": "the weather is {0}".format(weather),

"default": "default message"

}

# Return the matching response if there is one, default otherwise

def respond(message):

# Check if the message is in the responses

if message in responses:

# Return the matching message

bot\_message = responses[message]

else:

# Return the "default" message

bot\_message = responses["default"]

return bot\_message

In [3]: respond("what's today's weather?")

Out[3]: 'the weather is cloudy'

In [4]: respond("what's your name?")

Out[4]: 'my name is Greg'

In [5]: respond("what's your favorite color?")

Out[5]: 'default message'

**Adding variety**

It can get a little boring hearing the same old answers over and over. In this exercise, you'll add some variation. If you ask your bot how it's feeling, the likelihood that it responds with "oh I'm great!" or "I'm very sad today" should be equal.

Here, you'll use the random module - specifically random.choice(ls) - which randomly selects an element from a list ls.

A dictionary called responses, which maps each message to a list of possible responses, has been defined for you.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Import the random module.
* If the message is in responses, use random.choice() in the respond() function to choose a random matching response.
* If the message is not in responses, choose a random default response.

# Import the random module

import random

name = "Greg"

weather = "cloudy"

# Define a dictionary containing a list of responses for each message

responses = {

"what's your name?": [

"my name is {0}".format(name),

"they call me {0}".format(name),

"I go by {0}".format(name)

],

"what's today's weather?": [

"the weather is {0}".format(weather),

"it's {0} today".format(weather)

],

"default": ["default message"]

}

# Use random.choice() to choose a matching response

def respond(message):

# Check if the message is in the responses

if message in responses:

# Return a random matching response

bot\_message = random.choice(responses[message])

else:

# Return a random "default" response

bot\_message = random.choice(responses["default"])

return bot\_message

**Instructions 2/2**

**50 XP**

* [2](javascript:void(0))
* Great job! Adding some variety makes your bot much more fun to talk to. Now, hit 'Run Code' and use send\_message() (which utilizes the new respond() function) to ask the bot "what's your name?" 3 times.
* Hit 'Submit Answer' when you're done.

# Import the random module

import random

name = "Greg"

weather = "cloudy"

# Define a dictionary containing a list of responses for each message

responses = {

"what's your name?": [

"my name is {0}".format(name),

"they call me {0}".format(name),

"I go by {0}".format(name)

],

"what's today's weather?": [

"the weather is {0}".format(weather),

"it's {0} today".format(weather)

],

"default": ["default message"]

}

# Use random.choice() to choose a matching response

def respond(message):

if message in responses:

bot\_message = random.choice(responses[message])

else:

bot\_message = random.choice(responses["default"])

return bot\_message

In [2]: send\_message("what's your name?")

USER : what's your name?

BOT : I go by Greg

In [3]: send\_message("what's your name?")

USER : what's your name?

BOT : I go by Greg

In [4]: send\_message("what's your name?")

USER : what's your name?

BOT : they call me Greg

**ELIZA I: asking questions**

Asking questions is a great way to create an engaging conversation. Here, you'll create the very first hint of ELIZA's famous personality, by responding to statements with a question and responding to questions with answers.

A dictionary of responses with "question" and "statement" as keys and lists of appropriate responses as values has already been defined for you. Explore this in the Shell with responses.keys() and responses["question"].

**Instructions**

**100 XP**

* Define a respond() function which takes in message as an argument, and uses the string's .endswith() method to check if a message ends with a question mark.
* If the message does end with a question mark, choose a random "question" from the responses dictionary. Else, choose a random "statement" from the responses.
* Send the bot multiple messages with and without a question mark - these have been provided for you. If you want to experiment further in the Shell, be sure to first hit 'Run Code'.

import random

def respond(message):

# Check for a question mark

if message.endswith('?'):

# Return a random question

return random.choice(responses["question"])

# Return a random statement

return random.choice(responses["statement"])

# Send messages ending in a question mark

send\_message("what's today's weather?")

send\_message("what's today's weather?")

# Send messages which don't end with a question mark

send\_message("I love building chatbots")

send\_message("I love building chatbots")

<script.py> output:

USER : what's today's weather?

BOT : you tell me!

USER : what's today's weather?

BOT : I don't know :(

USER : I love building chatbots

BOT : oh wow!

USER : I love building chatbots

BOT : tell me more!

Nice! Asking the user questions is a great way to keep them talking.

**ELIZA II: Extracting key phrases**

The really clever thing about ELIZA is the way the program *appears* to understand what you told it by occasionally including phrases uttered by the user in its responses.

In this exercise, you will match messages against some common patterns and extract phrases using re.search(). A dictionary called rules has already been defined, which matches the following patterns:

* "do you think (.\*)"
* "do you remember (.\*)"
* "I want (.\*)"
* "if (.\*)"

Inspect this dictionary in the Shell before starting the exercise.

**Instructions**

**100 XP**

* Iterate over the rules dictionary using its .items() method, with pattern and responses as your iterator variables.
* Use re.search() with the pattern and message to create a match object.
* If there is a match, use random.choice() to pick a response.
* If '{0}' is in that response, use the match object's .group() method with index 1 to retrieve a phrase.

# Define match\_rule()

def match\_rule(rules, message):

response, phrase = "default", None

# Iterate over the rules dictionary

for pattern , responses in rules.items():

# Create a match object

match = re.search(pattern , message)

if match is not None:

# Choose a random response

response = random.choice(responses)

if '{0}' in response:

phrase = match.group(1)

# Return the response and phrase

return response.format(phrase)

# Test match\_rule

print(match\_rule(rules, "do you remember your last birthday"))

<script.py> output:

Why haven't you been able to forget your last birthday

Well Done! You're putting your knowledge of regex to work!

**ELIZA III: Pronouns**

To make responses grammatically coherent, you'll want to transform the extracted phrases from first to second person and vice versa. In English, conjugating verbs is easy, and simply swapping "me" and 'you', "my" and "your" works in *most* cases.

In this exercise, you'll define a function called replace\_pronouns() which uses re.sub() to map "me" and "my" to "you" and "your" (and vice versa) in a string.

**Instructions**

**100 XP**

* If 'me' is in message, use re.sub() to replace it with 'you'.
* If 'my' is in message, replace it with 'your'.
* If 'your' is in message, replace it with 'my'.
* If 'you' is in message, replace it with 'me'.

# Define replace\_pronouns()

def replace\_pronouns(message):

message = message.lower()

if 'me' in message:

# Replace 'me' with 'you'

return re.sub('me' , 'you' , message)

if 'my' in message:

# Replace 'my' with 'your'

return re.sub('my' , 'your' , message)

if 'your' in message:

# Replace 'your' with 'my'

return re.sub('your' , 'my' , message)

if 'you' in message:

# Replace 'you' with 'me'

return re.sub('you' , 'me' , message)

return message

print(replace\_pronouns("my last birthday"))

print(replace\_pronouns("when you went to Florida"))

print(replace\_pronouns("I had my own castle"))

<script.py> output:

your last birthday

when me went to florida

i had your own castle

Great! Your bot can now refer to itself! However, as you can see, one of the pitfalls of manually conjugating verbs is that they may not always be grammatically correct.

**ELIZA IV: Putting it all together**

Now you're going to put everything from the previous exercises together and experience the magic! The match\_rule(), send\_message(), and replace\_pronouns() functions have already been defined, and the rules dictionary is available in your workspace.

Your job here is to write a function called respond() with a single argument message which creates an appropriate response to be handled by send\_message().

**Instructions**

**100 XP**

* Get a response and phrase by calling match\_rule() with the rules dictionary and message.
* Check if the response is a template by seeing if it includes the string '{0}'. If it does:
  + Use the replace\_pronouns() function on phrase.
  + Include the phrase by using .format() on response and overriding the value of response.
* Hit 'Submit Answer' to see how the bot responds to the provided messages!

# Define respond()

def respond(message):

# Call match\_rule

response , phrase = match\_rule(rules , message)

if '{0}' in response:

# Replace the pronouns in the phrase

phrase = replace\_pronouns(phrase)

# Include the phrase in the response

response = response.format(phrase)

return response

# Send the messages

send\_message("do you remember your last birthday")

send\_message("do you think humans should be worried about AI")

send\_message("I want a robot friend")

send\_message("what if you could be anything you wanted")

<script.py> output:

USER : do you remember your last birthday

BOT : Did you think I would forget my last birthday

USER : do you think humans should be worried about AI

BOT : No chance

USER : I want a robot friend

BOT : Why do you want a robot friend

USER : what if you could be anything you wanted

BOT : Do you wish that me could be anything me wanted

In [1]: rules

Out[1]:

{'I want (.\*)': ['What would it mean if you got {0}',

'Why do you want {0}',

"What's stopping you from getting {0}"],

'do you remember (.\*)': ['Did you think I would forget {0}',

"Why haven't you been able to forget {0}",

'What about {0}',

'Yes .. and?'],

'do you think (.\*)': ['if {0}? Absolutely.', 'No chance'],

'if (.\*)': ["Do you really think it's likely that {0}",

'Do you wish that {0}',

'What do you think about {0}',

'Really--if {0}']}

Wow! You just built your own version of ELIZA, the most famous chatbot of all!

# Intent classification with regex I

You'll begin by implementing a very simple technique to recognize intents - looking for the presence of keywords.

A dictionary, keywords, has already been defined. It has the intents "greet", "goodbye", and "thankyou" as keys, and lists of keywords as the corresponding values. For example, keywords["greet"] is set to "["hello","hi","hey"].

Also defined is a second dictionary, responses, indicating how the bot should respond to each of these intents. It also has a default response with the key "default".

The function send\_message(), along with the bot and user templates, have also already been defined. Your job in this exercise is to create a dictionary with the intents as keys and regex objects as values.

##### Instructions

**100 XP**

##### Instructions

**100 XP**

* Iterate over the keywords dictionary, using intent and keys as your iterator variables.
* Use '|'.join(keys) to create regular expressions to match at least one of the keywords and pass it to re.compile() to compile the regular expressions into pattern objects. Store the result as the value of the patterns dictionary.

# Define a dictionary of patterns

patterns = {}

# Iterate over the keywords dictionary

for intent , keys in keywords.items():

# Create regular expressions and compile them into pattern objects

patterns[intent] = re.compile('|'.join(keys))

# Print the patterns

print(patterns)

<script.py> output:

{'greet': re.compile('hello|hi|hey'), 'goodbye': re.compile('bye|farewell'), 'thankyou': re.compile('thank|thx')}

Great work! The next step is to define a function to find the intent of a message.

# Intent classification with regex II

With your patterns dictionary created, it's now time to define a function to find the intent of a message.

##### Instructions

**100 XP**

* Iterate over the intents and patterns in the patterns dictionary using its .items() method.
* Use the .search() method of pattern to look for keywords in the message.
* If there is a match, return the corresponding intent.
* Call your match\_intent() function inside respond() with message as the argument and then hit 'Submit Answer' to see how the bot responds to the provided messages.

# Define a function to find the intent of a message

def match\_intent(message):

matched\_intent = None

for intent, pattern in patterns.items():

# Check if the pattern occurs in the message

if pattern.search(message):

matched\_intent = intent

return matched\_intent

# Define a respond function

def respond(message):

# Call the match\_intent function

intent = match\_intent(message)

# Fall back to the default response

key = "default"

if intent in responses:

key = intent

return responses[key]

# Send messages

send\_message("hello!")

send\_message("bye byeee")

send\_message("thanks very much!")

<script.py> output:

USER : hello!

BOT : Hello you! :)

USER : bye byeee

BOT : goodbye for now

USER : thanks very much!

BOT : you are very welcome

Well done! Your bot can now respond to messages even when they don't exactly match.

# Entity extraction with regex

Now you'll use another simple method, this time for finding a person's name in a sentence, such as "hello, my name is David Copperfield".

You'll look for the keywords "name" or "call(ed)", and find capitalized words using regex and assume those are names. Your job in this exercise is to define a find\_name() function to do this.

##### Instructions

**100 XP**

* Use re.compile() to create a pattern for checking if "name" or "call" keywords occur.
* Create a pattern for finding capitalized words.
* Use the .findall() method on name\_pattern to retrieve all matching words in message.
* Call your find\_name() function inside respond() and then hit 'Submit Answer' to see how the bot responds to the provided messages.

# Define find\_name()

def find\_name(message):

name = None

# Create a pattern for checking if the keywords occur

name\_keyword = re.compile(r"(name|call)")

# Create a pattern for finding capitalized words

name\_pattern = re.compile('[A-Z]{1}[a-z]\*')

if name\_keyword.search(message):

# Get the matching words in the string

name\_words = name\_pattern.findall(message)

if len(name\_words) > 0:

# Return the name if the keywords are present

name = ' '.join(name\_words)

return name

# Define respond()

def respond(message):

# Find the name

name = find\_name(message)

if name is None:

return "Hi there!"

else:

return "Hello, {0}!".format(name)

# Send messages

send\_message("my name is David Copperfield")

send\_message("call me Ishmael")

send\_message("People call me Cassandra")

<script.py> output:

USER : my name is David Copperfield

BOT : Hello, David Copperfield!

USER : call me Ishmael

BOT : Hello, Ishmael!

USER : People call me Cassandra

BOT : Hello, People Cassandra!

Excellent work! You just built a simple entity recognizer using regex. However, as you can see with the final output of send\_message(), the mix of using regex while making assumptions does have its limitations.

# word vectors with spaCy

In this exercise you'll get your first experience with word vectors! You're going to use the ATIS dataset, which contains thousands of sentences from real people interacting with a flight booking system.

The user utterances are available in the list sentences, and the corresponding intents in labels.

Your job is to create a 2D array X with as many rows as there are sentences in the dataset, where each row is a vector describing that sentence.

##### Instructions

**100 XP**

* Load the spaCy English model by calling spacy.load() with argument 'en'.
* Calculate the length of sentences using len() and the dimensionality of the word vectors using nlp.vocab.vectors\_length.
* For each sentence, call the nlp object with the sentence as the sole argument. Store the result as doc.
* Use the .vector attribute of doc to get the vector representation of each sentence, and store this vector in the appropriate row of X.

# Load the spacy model: nlp

nlp = spacy.load('en')

# Calculate the length of sentences

n\_sentences = len(sentences)

# Calculate the dimensionality of nlp

embedding\_dim = nlp.vocab.vectors\_length

# Initialize the array with zeros: X

X = np.zeros((n\_sentences, embedding\_dim))

# Iterate over the sentences

for idx, sentence in enumerate(sentences):

# Pass each each sentence to the nlp object to create a document

doc = nlp(sentence)

# Save the document's .vector attribute to the corresponding row in X

X[idx, :] = doc.vector

Great job! You can now find vector representations of words and sentences with spaCy.

# Intent classification with sklearn

An array X containing vectors describing each of the sentences in the ATIS dataset has been created for you, along with a 1D array y containing the labels. The labels are integers corresponding to the intents in the dataset. For example, label 0 corresponds to the intent atis\_flight.

Now, you'll use the scikit-learn library to train a classifier on this same dataset. Specifically, you will fit and evaluate a support vector classifier.

##### Instructions

**100 XP**

* Import the SVC class from sklearn.svm.
* Instantiate a classifier clf by calling SVC with a single keyword argument C with value 1.
* Fit the classifier to the training data X\_train and y\_train.
* Predict the labels of the test set, X\_test.

# Import SVC

from sklearn.svm import SVC

# Create a support vector classifier

clf = SVC(1)

# Fit the classifier using the training data

clf.fit(X\_train , y\_train)

# Predict the labels of the test set

y\_pred = clf.predict(X\_test)

# Count the number of correct predictions

n\_correct = 0

for i in range(len(y\_test)):

if y\_pred[i] == y\_test[i]:

n\_correct += 1

print("Predicted {0} correctly out of {1} test examples".format(n\_correct, len(y\_test)))

<script.py> output:

Predicted 162 correctly out of 201 test examples

Boom! You just trained a support vector machine for recognizing intents!

# Using spaCy's entity recognizer

In this exercise, you'll use spaCy's built-in entity recognizer to extract names, dates, and organizations from search queries. The spaCy library has been imported for you, and its English model has been loaded as nlp.

Your job is to define a function called extract\_entities(), which takes in a single argument message and returns a dictionary with the included entity types as keys, and the extracted entities as values. The included entity types are contained in a list called include\_entities.

##### Instructions

**100 XP**

* Create a dictionary called ents to hold the entities by calling dict.fromkeys() with include\_entities as the sole argument.
* Create a spacy document called doc by passing the message to the nlp object.
* Iterate over the entities in the document (doc.ents).
* Check whether the entity's .label\_ is one we are interested in. If so, assign the entity's .text attribute to the corresponding key in the ents dictionary.

###### Hint

* Use nlp(message) to create doc.
* Use ent to iterate over doc.ents, and check if ent.label\_ is in include\_entities. If so, assign ent.text to ents[ent.label\_].

# Define included\_entities

include\_entities = ['DATE', 'ORG', 'PERSON']

# Define extract\_entities()

def extract\_entities(message):

# Create a dict to hold the entities

ents = dict.fromkeys(include\_entities)

# Create a spacy document

doc = nlp(message)

for ent in doc.ents:

if ent.label\_ in include\_entities:

# Save interesting entities

ents[ent.label\_] = ent.text

return ents

print(extract\_entities('friends called Mary who have worked at Google since 2010'))

print(extract\_entities('people who graduated from MIT in 1999'))

<script.py> output:

{'ORG': None, 'PERSON': None, 'DATE': '2010'}

{'ORG': None, 'PERSON': None, 'DATE': None}

# Assigning roles using spaCy's parser

In this exercise you'll use spaCy's powerful syntax parser to assign roles to the entities in your users' messages. To do this, you'll define two functions, find\_parent\_item() and assign\_colors(). In doing so, you'll use a parse tree to assign roles, similar to how Alan did in the video.

Recall that you can access the ancestors of a word using its .ancestors attribute.

##### Instructions

**100 XP**

* Create a spacy document called doc by passing the message "let's see that jacket in red and some blue jeans" to the nlp object.
* In the find\_parent\_item(word) function, iterate over the ancestors of each word until an entity\_type() of "item" is found.
* In the assign\_colors(doc) function, iterate over the doc until an entity\_type of "color" is found. Then, find the parent item of this word.
* Pass in the spacy document to the assign\_colors() function.

# Create the document

doc = nlp("let's see that jacket in red and some blue jeans")

# Iterate over parents in parse tree until an item entity is found

def find\_parent\_item(word):

# Iterate over the word's ancestors

for parent in word.ancestors:

# Check for an "item" entity

if entity\_type(parent) == "item":

return parent.text

return None

# For all color entities, find their parent item

def assign\_colors(doc):

# Iterate over the document

for word in doc:

# Check for "color" entities

if entity\_type(word) == "color":

# Find the parent

item = find\_parent\_item(word)

print("item: {0} has color : {1}".format(item, word))

# Assign the colors

assign\_colors(doc)

<script.py> output:

item: jacket has color : red

item: jeans has color : blue

Nice! Your bot can now figure out simple relationships between entities.

# Rasa NLU

In this exercise, you'll use Rasa NLU to create an interpreter, which parses incoming user messages and returns a set of entities. Your job is to train an interpreter using the MITIE entity recognition model in Rasa NLU.

##### Instructions

**100 XP**

* Create a dictionary called args with a single key "pipeline" with value "spacy\_sklearn".
* Create a config by calling RasaNLUConfig() with the single argument cmdline\_args with value args.
* Create a trainer by calling Trainer() using the configuration as the argument.
* Create a interpreter by calling trainer.train() with the training\_data.

# Import necessary modules

from rasa\_nlu.converters import load\_data

from rasa\_nlu.config import RasaNLUConfig

from rasa\_nlu.model import Trainer

# Create args dictionary

args = {'pipeline' : 'spacy\_sklearn'}

# Create a configuration and trainer

config = RasaNLUConfig(cmdline\_args = args)

trainer = Trainer(config)

# Load the training data

training\_data = load\_data("./training\_data.json")

# Create an interpreter by training the model

interpreter = trainer.train(training\_data)

# Test the interpreter

print(interpreter.parse("I'm looking for a Mexican restaurant in the North of town"))

<script.py> output:

Fitting 2 folds for each of 6 candidates, totalling 12 fits

{'intent': {'name': 'restaurant\_search', 'confidence': 0.6627604390878398}, 'intent\_ranking': [{'name': 'restaurant\_search', 'confidence': 0.6627604390878398}, {'name': 'goodbye', 'confidence': 0.14633725788681204}, {'name': 'affirm', 'confidence': 0.09756426473688806}, {'name': 'greet', 'confidence': 0.09333803828846025}], 'text': "I'm looking for a Mexican restaurant in the North of town", 'entities': [{'entity': 'cuisine', 'extractor': 'ner\_crf', 'start': 18, 'value': 'mexican', 'end': 25}, {'entity': 'location', 'extractor': 'ner\_crf', 'start': 44, 'value': 'north', 'end': 49}]}

Congrats! You just trained an intent and entity recognizer without having to create any arrays.

# Data-efficient entity recognition

Most systems for extracting entities from text are built to extract 'Universal' things like names, dates, and places. But you probably don't have enough training data for your bot to make these systems perform well!

In this exercise, you'll activate the MITIE entity recognizer inside Rasa to extract restaurants-related entities using a very small amount of training data. A dictionary args has already been defined for you, along with a training\_data object.

##### Instructions

**100 XP**

* Create a config by calling RasaNLUConfig() with a single argument cmdline\_args with value {"pipeline": pipeline}.
* Create a trainer and use it to create an interpreter, just as you did in the previous exercise.

# Import necessary modules

from rasa\_nlu.config import RasaNLUConfig

from rasa\_nlu.model import Trainer

pipeline = [

"nlp\_spacy",

"tokenizer\_spacy",

"ner\_crf"

]

# Create a config that uses this pipeline

config = RasaNLUConfig(cmdline\_args = { 'pipeline' : pipeline })

# Create a trainer that uses this config

trainer = Trainer(config)

# Create an interpreter by training the model

interpreter = trainer.train(training\_data)

# Parse some messages

print(interpreter.parse("show me Chinese food in the centre of town"))

print(interpreter.parse("I want an Indian restaurant in the west"))

print(interpreter.parse("are there any good pizza places in the center?"))

<script.py> output:

{'intent': {'name': '', 'confidence': 0.0}, 'text': 'show me Chinese food in the centre of town', 'entities': [{'entity': 'location', 'extractor': 'ner\_crf', 'start': 28, 'value': 'centre', 'end': 34}]}

{'intent': {'name': '', 'confidence': 0.0}, 'text': 'I want an Indian restaurant in the west', 'entities': [{'entity': 'cuisine', 'extractor': 'ner\_crf', 'start': 10, 'value': 'indian', 'end': 16}, {'entity': 'location', 'extractor': 'ner\_crf', 'start': 35, 'value': 'west', 'end': 39}]}

{'intent': {'name': '', 'confidence': 0.0}, 'text': 'are there any good pizza places in the center?', 'entities': [{'entity': 'location', 'extractor': 'ner\_crf', 'start': 39, 'value': 'center', 'end': 45}]}

Very good! You just built a custom entity recogniser with Rasa NLU.