# EchoBot I

In [1]: USER: Hello!

Out[1]: BOT: I can hear you, you said: 'Hello!'

In [2]: USER: How are you?

Out[2]: BOT: I can hear you, you said: 'How are you?'

## EchoBot II

In [1]: def respond(message):

: return "I can hear you! you said: {}".format(message)

In [2]: def send\_message(message):

: # calls respond() to get response

In [3]: send\_message("hello!")

Out[3]: USER: hello!

...: BOT : I can hear you! you said: hello!

## EchoBot III

In [1]: import time

In [2]: time.sleep(0.5)

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

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!

 +100 XP

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**

**70 XP**

**Instructions**

**70 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.

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

Notice the {0} inside bot\_template and user\_template? These act as placeholders and will get replaced by the argument you pass when you call use the .format() method. For example, user\_template.format("Hello!") will result in the following string: "USER: Hello!".

**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**

**70 XP**

**Instructions**

**70 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.

[**Show Answer (-70 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of bot\_template.format(). Did you correctly specify the first argument? Expected response, but got message.

# 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

 +70 XP

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.

[**Take Hint (-15 XP)**](javascript:void(0))

**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.

[**Take Hint (-15 XP)**](javascript:void(0))

# 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 [1]: …

In [2]: Instructions 2/2

…

... Take Hint (-15 XP)

File "<stdin>", line 1

Instructions 2/2

^

SyntaxError: invalid syntax

In [3]:

In [3]: respond("what's your name?")

Out[3]: 'my name is Greg'

In [4]: respond("what's today's weather?")

Out[4]: 'the weather is cloudy'

In [5]:

**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.

[**Take Hint (-15 XP)**](javascript:void(0))

**Incorrect Submission**

Check the first if statement. Did you correctly specify the body? Did you call random.choice()?

**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.

[**Take Hint (-15 XP)**](javascript:void(0))

In [1]: send\_message

Out[1]: <function \_\_main\_\_.send\_message>

In [2]: respond

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

respond

NameError: name 'respond' is not defined

In [3]: # 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 [4]: respond

Out[4]: <function \_\_main\_\_.respond>

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

Out[5]: 'my name is Greg'

In [6]: respond("what's your name?")

Out[6]: 'I go by Greg'

In [7]: respond("what's your name?")

Out[7]: 'my name is Greg'

In [8]: respond("what's your name?")

Out[8]: 'I go by Greg'

In [9]: respond("what's your name?")

Out[9]: 'they call me Greg'

In [10]:

**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'.

[**Take Hint (-30 XP)**](javascript:void(0))

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")

In [1]: responses

Out[1]:

{'question': ["I don't know :(", 'you tell me!'],

'statement': ['tell me more!',

'why do you think that?',

'how long have you felt this way?',

'I find that extremely interesting',

'can you back that up?',

'oh wow!',

':)']}

<script.py> output:

USER : what's today's weather?

BOT : I don't know :(

USER : what's today's weather?

BOT : you tell me!

USER : I love building chatbots

BOT : can you back that up?

USER : I love building chatbots

BOT : tell me more!

In [2]:

## 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.

Take Hint (-30 XP)

# 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"))

In [1]: re

Out[1]: <module 're' from '/usr/lib/python3.6/re.py'>

In [2]: response

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

response

NameError: name 'response' is not defined

In [3]: random.choice()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

random.choice()

TypeError: choice() missing 1 required positional argument: 'seq'

In [4]: rules

Out[4]:

{'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}']}

<script.py> output:

What about your last birthday

In [5]:

 +100 XP

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'.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of re.sub(). Did you correctly specify the first argument? Expected me, but got I.

# 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"))

Traceback (most recent call last):

File "script.py", line 23, in <module>

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

File "script.py", line 11, in replace\_pronouns

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

NameError: name 'phrase' is not defined

<script.py> output:

your last birthday

when me went to florida

i had your own castle

+100 XP

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!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check the definition of respond(). Did you correctly specify the body? Did you call match\_rule()?

Check your call of replace\_pronouns(). Did you correctly specify the first argument? Expected phrase, but got message.

Check the first if statement. Did you correctly specify the body? Did you call response.format()?

# 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

Traceback (most recent call last):

File "script.py", line 16, in <module>

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

File "script.py", line 43, in send\_message

response = respond(message)

File "script.py", line 4, in respond

response, phrase = match\_rule

TypeError: 'function' object is not iterable

<script.py> output:

USER : do you remember your last birthday

BOT : do you reyoumber your last birthday

USER : do you think humans should be worried about AI

BOT : No chance

USER : I want a robot friend

BOT : i want a robot friend

USER : what if you could be anything you wanted

BOT : what if me could be anything me wanted

<script.py> output:

USER : do you remember your last birthday

BOT : your last birthday

USER : do you think humans should be worried about AI

BOT : humans should be worried about ai

USER : I want a robot friend

BOT : a robot friend

USER : what if you could be anything you wanted

BOT : you could be anything you wanted

<script.py> output:

USER : do you remember your last birthday

BOT : Why haven't you been able to forget do you reyoumber your last birthday

USER : do you think humans should be worried about AI

BOT : if do me think humans should be worried about ai? Absolutely.

USER : I want a robot friend

BOT : Why do you want i want a robot friend

USER : what if you could be anything you wanted

BOT : Really--if what if me could be anything me wanted

<script.py> output:

USER : do you remember your last birthday

BOT : What about my last birthday

USER : do you think humans should be worried about AI

BOT : No chance

USER : I want a robot friend

BOT : What's stopping you from getting a robot friend

USER : what if you could be anything you wanted

BOT : Really--if me could be anything me wanted

In [1]:

 +100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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] = '|'.join(keys)

# Print the patterns

print(patterns)

**Incorrect Submission**

Check the first for loop. Did you correctly specify the iterable part? Did you call keywords.items()?

Check the first for loop. Did you correctly specify the body? Did you call re.compile()?

# 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)

The keywords dictionary:

{'greet': ['hello', 'hi', 'hey'], 'goodbye': ['bye', 'farewell'], 'thankyou': ['thank', 'thx']}

<script.py> output:

{}

<script.py> output:

{}

<script.py> output:

{'greet': 'hello|hi|hey', 'goodbye': 'bye|farewell', 'thankyou': 'thank|thx'}

<script.py> output:

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

In [1]:

 +100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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 re.search(pattern, 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

In [1]:

 +100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check the definition of find\_name(). Did you correctly specify the body? Did you call re.compile() twice?

**Hint**

* Use | to see if "name" **or** "call" occur.
* Use the pattern '[A-Z]{1}[a-z]\*' to find capitalized words.

Incorrect Submission

Check the first if statement. Did you correctly specify the condition? Expected name\_keyword.search(message), but got name\_keyword.search(name\_pattern, message).

 +100 XP

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.

<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!

In [1]:

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you call len()?

Check your call of len(). Did you correctly specify the first argument? Running it generated an error: Argument 'string' has incorrect type (expected str, got list)

Check your call of nlp(). Did you correctly specify the first argument? Expected sentence, but got sentences.

# 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

Traceback (most recent call last):

File "script.py", line 16, in <module>

doc = nlp(sentences)

File "script.py", line 337, in \_\_call\_\_

doc = self.make\_doc(text)

File "script.py", line 365, in make\_doc

return self.tokenizer(text)

TypeError: Argument 'string' has incorrect type (expected str, got list)

In [1]:

+100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import SVC

from sklearn.svm import SVC as svc

# Create a support vector classifier

clf = svc(c=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)))

**Incorrect Submission**

Did you import SVC from sklearn.svm?

Did you define the variable clf without errors?

Check your call of clf.fit(). Did you correctly specify the first argument? Running it generated an error: name 'x\_train' is not defined.

Traceback (most recent call last):

File "script.py", line 5, in <module>

clf = svc(c=1)

TypeError: \_\_init\_\_() got an unexpected keyword argument 'c'

Traceback (most recent call last):

File "script.py", line 5, in <module>

clf = SVC(c=1)

TypeError: \_\_init\_\_() got an unexpected keyword argument 'c'

Traceback (most recent call last):

File "script.py", line 8, in <module>

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

NameError: name 'x\_train' is not defined

<script.py> output:

Predicted 162 correctly out of 201 test examples

In [1]:

 +100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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:

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

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

In [1]:

+100 XP

Great work. Your bot can now make use of spaCy's built-in entity recognizer.

**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**

**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.

[**Take Hint (-30 XP)**](javascript:void(0))

# 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) == "color":

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)

**Incorrect Submission**

Did you correctly iterate over the word's ancestors?

Did you correctly check for an "item" entity?

# 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)

Traceback (most recent call last):

File "script.py", line 24, in <module>

assign\_colors(doc)

File "script.py", line 20, in assign\_colors

item = find\_parent\_item(word)

File "script.py", line 7, in find\_parent\_item

for parent in word:

TypeError: 'spacy.tokens.token.Token' object is not iterable

<script.py> output:

item: None has color : red

item: None has color : blue

<script.py> output:

item: None has color : jacket

item: None has color : jeans

<script.py> output:

item: jacket has color : red

item: jeans has color : blue

In [1]:

 +100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you define the variable config without errors?

# 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"))

In [1]: # 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()

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"))

Traceback (most recent call last):

File "<stdin>", line 17, in <module>

interpreter = trainer.train(training\_data)

File "<stdin>", line 149, in train

components.validate\_arguments(self.pipeline, context)

File "<stdin>", line 78, in validate\_arguments

"The `backend` configuration key is "

ValueError: Can not train an empty pipeline. Make sure to specify a proper pipeline in the configuration using the `pipeline` key.The `backend` configuration key is NOT supported anymore.

Traceback (most recent call last):

File "script.py", line 10, in <module>

config = RasaNLUConfig(args)

File "script.py", line 78, in \_\_init\_\_

with io.open(filename, encoding='utf-8') as f:

TypeError: expected str, bytes or os.PathLike object, not dict

<script.py> output:

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

{'intent': {'name': 'restaurant\_search', 'confidence': 0.6627604390878398}, 'entities': [{'start': 18, 'end': 25, 'value': 'mexican', 'entity': 'cuisine', 'extractor': 'ner\_crf'}, {'start': 44, 'end': 49, 'value': 'north', 'entity': 'location', 'extractor': 'ner\_crf'}], '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"}

In [2]:

 +100 XP

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.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you define the variable interpreter without errors?

# 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?"))

Traceback (most recent call last):

File "script.py", line 18, in <module>

interpreter = trainer.train(load\_data("./training\_data.json"))

File "script.py", line 350, in load\_data

files = resolve\_data\_files(resource\_name)

File "script.py", line 341, in resolve\_data\_files

"{}".format(e))

ValueError: Invalid training data file / folder specified. Could not locate the resource '/tmp/tmp\_26gaoko/training\_data.json'.

<script.py> output:

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

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

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

In [1]:

 +100 XP

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

**SQL basics**

Time to begin writing queries for your first hotel booking chatbot! The database has been loaded as "hotels.db" and a cursor, which has access to the database, has already been defined for you as cursor.

Three queries are provided below. Your job is to identify which query returns ONLY the "Hotel California".

You can test each query below by calling the cursor's .execute() method and passing the query in as a string. Then, you can print the results by calling the cursor's .fetchall() method, which takes no arguments.

**Instructions**

**50 XP**

**Instructions**

**50 XP**

**Possible Answers**

* 

SELECT name from hotels where price = 'expensive' AND area = 'center'

* 

SELECT name from hotels where price = 'mid' AND area = 'north'

* 

SELECT name from hotels where price = 'expensive'

Submit Answer

[**Take Hint (-15 XP)**](javascript:void(0))

**Incorrect Submission**

Not quite. This query does not return any hotels.

Incorrect. This query does not return "Hotel California".

 +50 XP

Great work! You've got the basics of SQL down.

In [1]: c = sqlite3.connect('hotels.db').cursor()

In [2]: c.execute("SELECT name from hotels where price = 'expensive' AND area = 'center'"); c.fetchall()

ERROR! Session/line number was not unique in database. History logging moved to new session 5

Out[2]: []

In [3]: c.execute("SELECT name from hotels where price = 'mid' AND area = 'north'"); c.fetchall()

Out[3]: [('Hotel California',)]

In [4]: c.execute("SELECT name from hotels where price = 'expensive'"); c.fetchall()

Out[4]: []

In [5]: In[2]

Out[5]: 'c.execute("SELECT name from hotels where price = \'expensive\' AND area = \'center\'"); c.fetchall()'

In [6]: exec(In[2])

In [7]: c.execute("SELECT name from hotels where price = 'expensive' AND area = 'center'"); c.fetchall()

Out[7]: []

In [8]:

**SQL statements in Python**

It's time to begin writing SQL queries! In this exercise, your job is to run a query against the hotels database to find all the expensive hotels in the south. The connection to the database has been created for you, along with a cursor c.

As Alan described in the video, you should be careful about SQL injection. Here, you'll pass parameters the safe way: As an extra tuple argument to the .execute() method. This ensures malicious code can't be injected into your query.

**Instructions**

**100 XP**

* Define a tuple t of strings "south" and "hi" for the area and price.
* Execute the query using the cursor's .execute() method. You're looking for **all** of the fields for **all** hotels where the area is "south" and the price is "hi".
* Print the results using the cursor's .fetchall() method.

[**Take Hint (-30 XP)**](javascript:void(0))

**Exercise**

**Exercise**

**SQL statements in Python**

It's time to begin writing SQL queries! In this exercise, your job is to run a query against the hotels database to find all the expensive hotels in the south. The connection to the database has been created for you, along with a cursor c.

As Alan described in the video, you should be careful about SQL injection. Here, you'll pass parameters the safe way: As an extra tuple argument to the .execute() method. This ensures malicious code can't be injected into your query.

**Instructions**

**100 XP**

* Define a tuple t of strings "south" and "hi" for the area and price.
* Execute the query using the cursor's .execute() method. You're looking for **all** of the fields for **all** hotels where the area is "south" and the price is "hi".
* Print the results using the cursor's .fetchall() method.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import sqlite3

import sqlite3

# Open connection to DB

conn = sqlite3.connect('hotels.db')

# Create a cursor

c = conn.cursor()

# Define area and price

area, price = "south", "hi"

t = (area, price)

# Execute the query

c.execute('SELECT \* FROM hotels WHERE area="south" AND price ="hi"')

# Print the results

print(c.fetchall())

<script.py> output:

[('Grand Hotel', 'hi', 'south', 5)]

In [1]:

 +100 XP

Nice! According to our database, the Grand Hotel is the only high-end hotel in the south.

**Creating queries from parameters**

Now you're going to implement a more powerful function for querying the hotels database. The goal is for that function to take arguments that can later be specified by other parts of your code.

More specifically, your job is to define a find\_hotels() function which takes a single argument - a dictionary of column names and values - and returns a list of matching hotels from the database.

**Instructions**

**100 XP**

* A filters list has been created for you. Join this list together with the strings " WHERE " and " and ".
* Create a tuple of the values of the params dictionary.
* Create a connection and cursor to "hotels.db" and then execute the query, just as in the previous exercise.
* Return the results of the query.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of tuple(). Did you correctly specify the first argument? Did you call params.values()?

Check your call of sqlite3.connect(). Did you correctly specify the first argument? Expected 'hotels.db', but got "\_\_\_\_".

Check your call of c.execute(). Did you correctly specify the first argument? Expected query, but got 'SELECT \* FROM hotels WHERE area="south" AND price ="hi"'

Did you correctly specify the body? Check your call of c.execute(). Did you specify the second argument?

Check your call of c.execute(). Did you correctly specify the second argument? Expected t, but got 'SELECT \* FROM hotels WHERE area="south" AND price ="hi"'

Check the definition of find\_hotels(). To verify it, we reran find\_hotels({"area": "south", "price":"lo"}). Expected [('Cozy Cottage', 'lo', 'south', 2)], but got None.

Check the definition of find\_hotels(). To verify it, we reran find\_hotels({"area": "south", "price":"lo"}). Running the higlighted expression generated an error: function takes at most 2 arguments (3 given).

# Define find\_hotels()

def find\_hotels(params):

# Create the base query

query = 'SELECT \* FROM hotels'

# Add filter clauses for each of the parameters

if len(params) > 0:

filters = ["{}=?".format(k) for k in params]

query += " WHERE " + " and ".join(filters)

# Create the tuple of values

t = tuple(params.values())

# Open connection to DB

conn = sqlite3.connect("hotels.db")

# Create a cursor

c = conn.cursor()

# Execute the query

c.execute(query, t)

# Return the results

return (c.fetchall())

 +100 XP

Super! You've now got a function that can find matching hotels for any area and price range combination. You'll practice using it in the next exercise!

**Exercise**

**Exercise**

**Using your custom function to find hotels**

Here, you'll see your find\_hotels() function in action! Recall that it accepts a single argument, params, which is a dictionary of column names and values.

**Instructions**

**100 XP**

* Create the params dictionary with the column names (keys) "area" and "price", with corresponding values "south" and "lo".
* Use the find\_hotels() function along with your params dictionary to find all inexpensive hotels in the South.

[**Take Hint (-30 XP)**](javascript:void(0))

<script.py> output:

[('Cozy Cottage', 'lo', 'south', 2)]

In [1]:

**Exercise**

**Exercise**

**Creating SQL from natural language**

Now you'll write a respond() function that can handle messages like "I want an expensive hotel in the south of town" and respond appropriately according to the number of matching results in a database. This is an important functionality for any database-backed chatbot.

Your find\_hotels() function from the previous exercises has already been defined for you, along with a Rasa NLU interpreter object, which can handle hotel queries, and a list of responses, which you can explore in the Shell.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Use the .parse() method of interpreter to extract the "entities" in the message.
* Find matching hotels using the params dictionary and find\_hotels() function.
* Use the min() function to choose the right index for the response to send. In this case, n is the number of results.
* Select the appropriate response from the responses list and insert the names of hotels using the .format() method.

[**Take Hint (-15 XP)**](javascript:void(0))

* Excellent! You've built a chatbot that can interpret the results of your hotel DB queries. Now, call the respond() function with the message "I want an expensive hotel in the south of town". Place it inside a call to print() so that you can see the response of your bot in the shell.

# Define respond()

def respond(message):

# Extract the entities

entities = interpreter.parse(message)["entities"]

# Initialize an empty params dictionary

params = {}

# Fill the dictionary with entities

for ent in entities:

params[ent["entity"]] = str(ent["value"])

# Find hotels that match the dictionary

results = find\_hotels(params)

# Get the names of the hotels and index of the response

names = [r[0] for r in results]

n = min(len(results),3)

# Select the nth element of the responses array

return responses[n].format(\*names)

# Test the respond() function

respond("I want an expensive hotel in the south of town")

**Refining your search**

Now you'll write a bot that allows users to add filters incrementally, just in case they don't specify all of their preferences in one message.

To do this, initialize an empty dictionary params outside of your respond() function (as opposed to inside the function, like in the previous exercise). Your respond() function will take in this dictionary as an argument.

**Instructions**

**100 XP**

* Define a respond() function that accepts two arguments - a message **and** a dictionary of params - and returns two results - the message to send to the user and the updated params dictionary.
* Extract "entities" from the message using the .parse() method of the interpreter, exactly like you did in the previous exercise.
* Find the hotels that match params using your find\_hotels() function.
* Initialize the params dictionary outside the respond() function and hit 'Submit Answer' to pass the messages to the bot.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Running the higlighted expression generated an error: name '\_\_\_\_' is not defined.

# Define a respond function, taking the message and existing params as input

def respond(message, params):

# Extract the entities

entities = interpreter.parse(message)["entities"]

# Fill the dictionary with entities

for ent in entities:

params[ent["entity"]] = str(ent["value"])

# Find the hotels

results = find\_hotels(params)

names = [r[0] for r in results]

n = min(len(results), 3)

# Return the appropriate response

return responses[n].format(\*names), params

# Initialize params dictionary

params = {}

# Pass the messages to the bot

for message in ["I want an expensive hotel", "in the north of town"]:

print("USER: {}".format(message))

response, params = respond(message, params)

print("BOT: {}".format(response))

<script.py> output:

USER: I want an expensive hotel

Traceback (most recent call last):

File "script.py", line 23, in <module>

response, params = respond(message, params)

File "script.py", line 10, in respond

results = \_\_\_\_

NameError: name '\_\_\_\_' is not defined

<script.py> output:

USER: I want an expensive hotel

BOT: Grand Hotel is one option, but I know others too :)

USER: in the north of town

BOT: Ben's BnB is a great hotel!

In [1]:

 +100 XP

Great! Your chatbot can now help users even when they split their preferences over a few messages.

**Exercise**

**Exercise**

**Basic negation**

Quite often, you'll find your users telling you what they *don't* want - and that's important to understand! In general, negation is a difficult problem in NLP. Here, we'll take a very simple approach that works for many cases.

A list of tests called tests has been defined for you. Explore it in the Shell - you'll find that each test is a tuple consisting of:

* A string containing a message with entities.
* A dictionary containing the entities as keys and a Boolean saying whether they are negated as the key.

Your job is to define a function called negated\_ents() which looks for negated entities in a message.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Using list comprehension, check if the words "south" or "north" appear in the message and extract those entities.
* Split the sentence into chunks ending with each entity. To do this:
  + Use the .index() method of phrase to find the starting index of each entity e and add the entity's length to it to find the index of the end of the entity.
  + Starting with start=0, take slices of the string from start to end for each end in ends. Append each slice of the sentence to the list, chunks. Ensure you update your starting position with each iteration.
* For each entity, if "not" or "n't" appears in the chunk, consider this entity negated.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Running the higlighted expression generated an error: slice indices must be integers or None or have an \_\_index\_\_ method

Traceback (most recent call last):

File "script.py", line 34, in <module>

print(negated\_ents(test[0]) == test[1])

File "script.py", line 15, in negated\_ents

chunks.append(phrase[start:end])

TypeError: slice indices must be integers or None or have an \_\_index\_\_ method

In [1]:

**Hint**

* To compute ents, pass in ["south", "north"] as the iterable to the provided list comprehension.
* Concatenate phrase.index(e) and len(e) to calculate ends.
* Use the .append() method on chunks and index phrase from start to end.
* If "not" or "n't" is in the chunk, consider the entity negated by setting result[ent] to False.

**Incorrect Submission**

Check the first if statement. Did you correctly specify the body? The system wants to check the first if statement but hasn't found it.

**Incorrect Submission**

Running the higlighted expression generated an error: name '\_\_\_\_' is not defined.

**Incorrect Submission**

Did you give the entity the key False if "not" or "n't" appears in the chunk?

Check the definition of negated\_ents(). To verify it, we reran negated\_ents("no I don't want to be in the south"). Expected {'south': False}, but got {'south': True}.

# Define negated\_ents()

def negated\_ents(phrase):

# Extract the entities using keyword matching

ents = [e for e in ["south", "north"] if e in phrase]

# Find the index of the final character of each entity

ends = sorted([phrase.index(e) + len(e) for e in ents])

# Initialise a list to store sentence chunks

chunks = []

# Take slices of the sentence up to and including each entitiy

start = 0

for end in ends:

chunks.append(phrase[start:end])

start = end

result = {}

# Iterate over the chunks and look for entities

for chunk in chunks:

for ent in ents:

if ent in chunk:

# If the entity contains a negation, assign the key to be False

if "not" in chunk or "n't" in chunk:

result[ent] = False

else:

result[ent] = True

result[ent] = not ("not" in chunk or "n't" in chunk)

return result

# Check that the entities are correctly assigned as True or False

for test in tests:

print(negated\_ents(test[0]) == test[1])

# Define negated\_ents()

def negated\_ents(phrase):

# Extract the entities using keyword matching

ents = [e for e in ["south", "north"] if e in phrase]

# Find the index of the final character of each entity

ends = sorted([phrase.index(e) + len(e) for e in ents])

# Initialise a list to store sentence chunks

chunks = []

# Take slices of the sentence up to and including each entitiy

start = 0

for end in ends:

chunks.append(phrase[start:end])

start = end

result = {}

# Iterate over the chunks and look for entities

for chunk in chunks:

for ent in ents:

if ent in chunk:

# If the entity contains a negation, assign the key to be False

if "not" in chunk or "n't" in chunk:

result[ent] = False

else:

result[ent] = True

return result

# Check that the entities are correctly assigned as True or False

for test in tests:

print(negated\_ents(test[0]) == test[1])

<script.py> output:

False

True

True

True

Traceback (most recent call last):

File "script.py", line 35, in <module>

print(negated\_ents(test[0]) == test[1])

File "script.py", line 26, in negated\_ents

result[ent] = \_\_\_\_

NameError: name '\_\_\_\_' is not defined

<script.py> output:

False

True

True

True

<script.py> output:

False

True

True

True

<script.py> output:

True

True

True

True

In [1]:

**Filtering with excluded slots**

Now you're going to put together some of the ideas from previous exercises in order to allow users to tell your bot about what they do and do not want, split across multiple messages.

The negated\_ents() function has already been defined for you. Additionally, a slightly tweaked version of the find\_hotels() function, which accepts a neg\_params dictionary in addition to a params dictionary, has been defined.

**Instructions**

**100 XP**

* Define a respond() function which accepts a message, params, and neg\_params as arguments.
* Use the negated\_ents() function with message and ent\_vals as arguments. Store the result in negated.
* Use the tweaked find\_hotels() function with the params and neg\_params dictionaries as arguments to find matching hotels. Store the result in results.
* Initialize the params and neg\_params dictionaries outside the respond() function and hit 'Submit Answer' to see the bot's responses!

[**Take Hint (-30 XP)**](javascript:void(0))

# Define the respond function

def respond(message, params, neg\_params):

# Extract the entities

entities = interpreter.parse(message)["entities"]

ent\_vals = [e["value"] for e in entities]

# Look for negated entities

negated = negated\_ents(message, ent\_vals)

for ent in entities:

if ent["value"] in negated and negated[ent["value"]]:

neg\_params[ent["entity"]] = str(ent["value"])

else:

params[ent["entity"]] = str(ent["value"])

# Find the hotels

results = find\_hotels(params, neg\_params)

names = [r[0] for r in results]

n = min(len(results),3)

# Return the correct response

return responses[n].format(\*names), params, neg\_params

# Initialize params and neg\_params

params, neg\_params = {}, {}

# Pass the messages to the bot

for message in ["I want a cheap hotel", "but not in the north of town"]:

print("USER: {}".format(message))

response, params, neg\_params = respond(message, params, neg\_params)

print("BOT: {}".format(response))

<script.py> output:

USER: I want a cheap hotel

BOT: Cozy Cottage is a great hotel!

USER: but not in the north of town

BOT: I'm sorry :( I couldn't find anything like that

In [1]:

 +100 XP

Great work! Your bot can now handle just about any sequence of requests, with positive or negative preferences.

**Exercise**

**Exercise**

**Form filling**

You'll often want your bot to guide users through a series of steps, such as when they're placing an order.

In this exercise, you'll begin building a bot that lets users order coffee. They can choose between two types: Colombian and Kenyan. If the user provides unexpected input, your bot will handle this differently depending on where they are in the flow.

Your job here is to identify the appropriate state and next state based on the intents and response messages provided. For example, if the intent is "order", then the state changes from INIT to CHOOSE\_COFFEE.

A function send\_message(policy, state, message) has already been defined for you. It takes the policy, the current state, and message as arguments, and returns the new state as a result. Additionally, an interpret(message) function, similar to the one Alan described in the video, has been pre-defined for you.

**Instructions**

**100 XP**

* Define three states: INIT with value 0, CHOOSE\_COFFEE with value 1, and ORDERED with value 2.
* Create a dictionary called policy with tuples as keys and values. Each key is a tuple containing a state and an intent, and each value is a tuple containing the next state and the response message. The messages have been filled in for you. Your job is to fill in the states.
* Instantiate a variable state with the value INIT.
* For each of the messages, call the send\_message() function, passing in the policy, state, and message.

[**Take Hint (-30 XP)**](javascript:void(0))

# Define the INIT state

INIT = 0

# Define the CHOOSE\_COFFEE state

CHOOSE\_COFFEE = 1

# Define the ORDERED state

ORDERED = 2

# Define the policy rules

policy = {

(INIT, "order"): (CHOOSE\_COFFEE, "ok, Colombian or Kenyan?"),

(INIT, "none"): (INIT, "I'm sorry - I'm not sure how to help you"),

(CHOOSE\_COFFEE, "specify\_coffee"): (ORDERED, "perfect, the beans are on their way!"),

(CHOOSE\_COFFEE, "none"): (CHOOSE\_COFFEE, "I'm sorry - would you like Colombian or Kenyan?"),

}

# Create the list of messages

messages = [

"I'd like to become a professional dancer",

"well then I'd like to order some coffee",

"my favourite animal is a zebra",

"kenyan"

]

# Call send\_message() for each message

state = INIT

for message in messages:

state = send\_message(policy, state, message)

<script.py> output:

USER : I'd like to become a professional dancer

BOT : I'm sorry - I'm not sure how to help you

USER : well then I'd like to order some coffee

BOT : ok, Colombian or Kenyan?

USER : my favourite animal is a zebra

BOT : I'm sorry - would you like Colombian or Kenyan?

USER : kenyan

BOT : perfect, the beans are on their way!

In [1]:

 +100 XP

Congrats! You just built your first chatbot with a state machine

**Exercise**

**Exercise**

**Asking contextual questions**

Sometimes your users need some help! They will have questions and expect the bot to help them.

In this exercise, you'll allow users to ask the coffee bot to explain the steps to them. As in the previous exercise, the answer they get will depend on where they are in the flow.

**Instructions**

**100 XP**

* Add two rules to your policy\_rules to handle the intent "ask\_explanation" when in the states INIT or CHOOSE\_COFFEE.
* Inside the send\_messages() function, call the send\_message() function with state and msg as arguments to define the new state. Then, hit 'Submit Answer' to send the messages and see the bot's responses.

[**Take Hint (-30 XP)**](javascript:void(0))

# Define the states

INIT=0

CHOOSE\_COFFEE=1

ORDERED=2

# Define the policy rules dictionary

policy\_rules = {

(INIT, "ask\_explanation"): (INIT, "I'm a bot to help you order coffee beans"),

(INIT, "order"): (CHOOSE\_COFFEE, "ok, Colombian or Kenyan?"),

(CHOOSE\_COFFEE, "specify\_coffee"): (ORDERED, "perfect, the beans are on their way!"),

(CHOOSE\_COFFEE, "ask\_explanation"): (CHOOSE\_COFFEE, "We have two kinds of coffee beans - the Kenyan ones make a slightly sweeter coffee, and cost $6. The Brazilian beans make a nutty coffee and cost $5.")

}

# Define send\_messages()

def send\_messages(messages):

state = INIT

for msg in messages:

state = send\_message(msg)

# Send the messages

send\_messages([

"what can you do for me?",

"well then I'd like to order some coffee",

"what do you mean by that?",

"kenyan"

])

**Incorrect Submission**

Check the first for loop. Did you correctly specify the body? Have you specified the arguments for send\_message() using the right syntax?

Traceback (most recent call last):

File "script.py", line 25, in <module>

"kenyan"

File "script.py", line 18, in send\_messages

state = send\_message(msg)

TypeError: send\_message() missing 1 required positional argument: 'message'

In [1]:

help(send\_message)

Help on function send\_message in module \_\_main\_\_:

send\_message(state, message)

<script.py> output:

USER : what can you do for me?

BOT : I'm a bot to help you order coffee beans

USER : well then I'd like to order some coffee

BOT : ok, Colombian or Kenyan?

USER : what do you mean by that?

BOT : We have two kinds of coffee beans - the Kenyan ones make a slightly sweeter coffee, and cost $6. The Brazilian beans make a nutty coffee and cost $5.

USER : kenyan

BOT : perfect, the beans are on their way!

In [2]:

 +100 XP

Good work! Your bot can now modify its answers by considering context

**Exercise**

**Exercise**

**Dealing with rejection**

What happens if you make a suggestion to your user and they don't like it? Your bot will look really silly if it makes the same suggestion again right away.

Here, you're going to modify your respond() function so that it accepts and returns 4 arguments:

* The user message as an argument, and the bot response as the first return value.
* A dictionary params including the entities the user has specified.
* A prev\_suggestions list. When passed to respond(), this should contain the suggestions made in the previous bot message. When returned by respond(), it should contain the current suggestions.
* An excluded list, which contains all of the results your user has already explicitly rejected.

Your function should add the previous suggestions to the excluded list whenever it receives a "deny" intent. It should also filter out excluded suggestions from the response.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Define a respond() function with 4 arguments: message, params, prev\_suggestions, and excluded.
* Interpret the message and store the result in parse\_data.
* The value of the "intent" key of parse\_data is itself a dictionary of key-value pairs. Assign parse\_data["intent"]["name"] to intent, and parse\_data["entities"] to entities.
* If the intent is "deny", use the .extend() method of the excluded list to add prev\_suggestions to it.
* Initialize the empty params dictionary and empty suggestions and excluded lists. Then, hit 'Submit Answer' to send the messages to the bot.

[**Take Hint (-30 XP)**](javascript:void(0))

# Define respond()

def respond(message, params, prev\_suggestions, excluded):

# Interpret the message

parse\_data = interpret(message)

# Extract the intent

intent = parse\_data["intent"]["name"]

# Extract the entities

entities = parse\_data["entities"]

# Add the suggestion to the excluded list if intent is "deny"

if intent == "deny":

excluded.extend(prev\_suggestions)

# Fill the dictionary with entities

for ent in entities:

params[ent["entity"]] = str(ent["value"])

# Find matching hotels

results = [

r

for r in find\_hotels(params, excluded)

if r[0] not in excluded

]

# Extract the suggestions

names = [r[0] for r in results]

n = min(len(results), 3)

suggestions = names[:2]

return responses[n].format(\*names), params, suggestions, excluded

# Initialize the empty dictionary and lists

params, suggestions, excluded = {}, [], []

# Send the messages

for message in ["I want a mid range hotel", "no that doesn't work for me"]:

print("USER: {}".format(message))

response, params, suggestions, excluded = respond(message, params, suggestions, excluded)

print("BOT: {}".format(response))

In [1]: interpret

Out[1]: <function \_\_main\_\_.interpret>

In [2]: excluded

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

excluded

NameError: name 'excluded' is not defined

<script.py> output:

USER: I want a mid range hotel

BOT: Hotel for Dogs is one option, but I know others too :)

USER: no that doesn't work for me

BOT: Grand Hotel is one option, but I know others too :)

In [3]:

 +100 XP

Great! Your bot can now handle negative feedback gracefully.

**Exercise**

**Exercise**

**Pending actions I**

You can really improve the user experience of your bot by asking the user simple yes or no follow-up questions. One easy way to handle these follow-ups is to define *pending* actions which get executed as soon as the user says "yes", and wiped if the user says "no".

In this exercise, you're going to define a policy() function which takes the intent as its sole argument and returns two values: The next action to take and a pending action. The policy function should return this pending action when a "yes" or "affirm" intent is returned and should wipe the pending actions if a "no" or "deny" intent is returned.

Here, the interpret(message) function has been defined for you such that if "yes" is in the message, "affirm" is returned, and if "no" is in the message, then "deny" is returned.

**Instructions**

**100 XP**

* Define a function called policy() which takes intent as its argument.
* If the intent is "affirm", return a "do\_pending" action and None.
* If the intent is "deny", return a "Ok" action and None.

[**Take Hint (-30 XP)**](javascript:void(0))

# Define policy()

def policy(intent):

# Return "do\_pending" if the intent is "affirm"

if intent == "affirm":

return "do\_pending", None

# Return "Ok" if the intent is "deny"

if intent == "deny":

return "Ok", None

if intent == "order":

return "Unfortunately, the Kenyan coffee is currently out of stock, would you like to order the Brazilian beans?", "Alright, I've ordered that for you!"

 +100 XP

Excellent work! With a policy() function defined, you can now incorporate it into a send\_message() function.

**Pending actions II**

Having defined your policy() function, it's now time to write a send\_message() function which takes both a pending action and a message as its arguments and leverages the policy() function to determine the bot's response.

Your policy(intent) function from the previous exercise has been pre-loaded.

**Instructions**

**100 XP**

* Define a function called send\_message() which takes in two arguments: pending and message.
* Pass in the interpretation of message as an argument to policy() and unpack the result into the variables action and pending\_action.
* If the action is "do\_pending" and pending is not None, print the pending response. Else, print the action.
* Inside the definition of the send\_messages() function, call your send\_message() function with pending and msg as arguments. Then, hit 'Submit Answer' to send the messages and see the results.

[**Take Hint (-30 XP)**](javascript:void(0))

# Define send\_message()

def send\_message(pending, message):

print("USER : {}".format(message))

action, pending\_action = policy(interpret(message))

if action == "do\_pending" and pending is not None:

print("BOT : {}".format(pending))

else:

print("BOT : {}".format(action))

return pending\_action

# Define send\_messages()

def send\_messages(messages):

pending = None

for msg in messages:

pending = send\_message(pending, msg)

# Send the messages

send\_messages([

"I'd like to order some coffee",

"ok yes please"

])

**Incorrect Submission**

Check your call of policy(). Did you correctly specify the first argument? Did you call interpret()?

Check the definition of send\_message(). Did you correctly specify the body? Have you specified the arguments for policy() using the right syntax?

**Hint**

* To compute the action and pending\_action, be sure to pass interpret(message) as an argument to policy().
* Depending on the value of action and pending, print the appropriate response using the provided .format() methods.

+70 XP

Very well done! Your bot can now follow up on its own suggestions!

**Exercise**

**Exercise**

**Pending state transitions**

You'll often need to briefly deviate from the flow of a conversation, for example to authenticate a user, before returning to the topic of discussion.

In these cases, it's often simpler - and easier to debug - if you save some actions/states as *pending* rather than adding ever more complicated rules.

Here, you're going to define a policy\_rules dictionary, where the keys are tuples of the current state and the received intent, and the values are tuples of the next state, the bot's response, and a state for which to set a pending transition.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Complete the policy\_rules dictionary by filling in the values:
  + A user starts in the INIT state.
  + If the user is in the INIT state and tries to place an order, you should ask for their number and create a pending transition to the AUTHED state.
  + This is the only policy rule which creates a pending transition, so the others simply have a pending state value of None.
* The pending state has been added as the second argument of the send\_message() function, which now returns the new state as well as the pending state. Call this send\_message() function inside send\_messages(), unpacking the output into the variables state and pending.
* Hit 'Submit Answer' to send the messages to the bot!

[**Take Hint (-30 XP)**](javascript:void(0))

**Hint**

* Once the user provides a "number", the state changes to AUTHED.
* Don't confuse the send\_message() function with the send\_messages() function!

**Incorrect Submission**

Did you correctly specify the state and intent for the tuple of the value in the first key-value pair of policy\_rules?

<script.py> output:

USER : I'd like to order some coffee

BOT : you'll have to log in first, what's your phone number?

USER : 555-1234

Traceback (most recent call last):

File "script.py", line 26, in <module>

"kenyan"

File "script.py", line 20, in send\_messages

state, pending = send\_message(state, pending, msg)

File "script.py", line 16, in send\_message

new\_state, response, pending\_state = policy\_rules[(state, interpret(message))]

KeyError: (1, 'number')

<script.py> output:

USER : I'd like to order some coffee

BOT : you'll have to log in first, what's your phone number?

USER : 555-1234

BOT : perfect, welcome back!

BOT : would you like Colombian or Kenyan?

USER : kenyan

BOT : perfect, the beans are on their way!

BOT : would you like Colombian or Kenyan?

In [1]:

+0 XP

Fantastic! You just built a bot that can handle some fairly complicated conversations!

**Exercise**

**Exercise**

**Putting it all together I**

It's time to put everything you've learned in the course together by combining the coffee ordering bot with the ELIZA rules from chapter 1.

To begin, you'll define a function called chitchat\_response(), which calls the predefined function match\_rule() from back in chapter 1. This returns a response if the message matched an ELIZA template, and otherwise, None.

The ELIZA rules are contained in a dictionary called eliza\_rules.

**Instructions**

**100 XP**

* Define a chitchat\_response() function which takes in a message argument.
* Call the match\_rule() function with eliza\_rules and message as arguments. Unpack the output into response and phrase.
* If the response is "default", return None.
* If "{0}" is in the response, replace the pronouns of the phrase using replace\_pronouns(), and then include the phrase in the response by using .format() on response.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check the second if statement. Did you correctly specify the body? Did you call replace\_pronouns()?

Check your call of response.format(). Did you correctly specify the first argument? Expected phrase, but got response.

# Define chitchat\_response()

def chitchat\_response(message):

# Call match\_rule()

response, phrase = match\_rule(eliza\_rules, message)

# Return none if response is "default"

if response == "default":

return None

if '{0}' in response:

# Replace the pronouns of phrase

phrase = replace\_pronouns(phrase)

# Calculate the response

response = response.format(phrase)

return response

+100 XP

Wow! You've put it all together and your bot can now interleave chit-chat and functional conversation.

**Putting it all together II**

With your chitchat\_response(message) function defined, the next step is to define a send\_message() function. This function should first call chitchat\_response(message) and only use the coffee bot policy if there is no matching message.

**Instructions**

**100 XP**

* Define a send\_message() function which takes in 3 arguments: state, pending, and message.
* Call chitchat\_response(message), storing the result in response. If there is a response, print it and return the state along with None.
* Unpack the policy\_rules dictionary into the variables new\_state, response, and pending\_state. To do this, pass in a tuple consisting of state and interpret(message).
* If pending is not none, extract the new states and response by using pending as the key of policy\_rules.

[**Take Hint (-30 XP)**](javascript:void(0))

# Define send\_message()

def send\_message(state, pending, message):

print("USER : {}".format(message))

response = chitchat\_response(message)

if response is not None:

print("BOT : {}".format(response))

return state, None

# Calculate the new\_state, response, and pending\_state

new\_state, response, pending\_state = policy\_rules[(state, interpret(message))]

print("BOT : {}".format(response))

if pending is not None:

new\_state, response, pending\_state = policy\_rules[pending]

print("BOT : {}".format(response))

if pending\_state is not None:

pending = (pending\_state, interpret(message))

return new\_state, pending

# Define send\_messages()

def send\_messages(messages):

state = INIT

pending = None

for msg in messages:

state, pending = send\_message(state, pending, msg)

# Send the messages

send\_messages([

"I'd like to order some coffee",

"555-12345",

"do you remember when I ordered 1000 kilos by accident?",

"kenyan"

])

<script.py> output:

USER : I'd like to order some coffee

BOT : you'll have to log in first, what's your phone number?

USER : 555-12345

BOT : perfect, welcome back!

BOT : would you like Colombian or Kenyan?

USER : do you remember when I ordered 1000 kilos by accident?

BOT : Yes .. and?

USER : kenyan

BOT : perfect, the beans are on their way!

In [1]:

**Exercise**

**Exercise**

**Generating text with neural networks**

In this final exercise of the course, you're going to generate text using a neural network trained on the scripts of every episode of The Simpsons. Specifically, you'll use a simplified version of the sample\_text() function that Alan described in the video.

It takes in two arguments: seed and temperature. The seed argument is the initial sequence that the network uses to generate the subsequent text, while the temperature argument controls how risky the network is when generating text. At very low temperatures, it just repeats the most common combinations of letters, and at very high temperatures, it generates complete gibberish. In order to ensure fast runtimes, the network in this exercise will only work for a subset of temperature values.

After you finish this exercise, be sure to check out [**this tutorial**](https://www.datacamp.com/community/tutorials/facebook-chatbot-python-deploy) by Alan where he walks you through how to connect a chatbot to Facebook Messenger!

**Instructions**

**100 XP**

* Set the seed to be "i'm gonna punch lenny in the back of the".
* For each of the riskiness values [0.2, 0.5, 1.0, 1.2], call the sample\_text() function with the arguments seed and temperature.

[**Take Hint (-30 XP)**](javascript:void(0))

# Feed the seed text into the neural network

seed = "i'm gonna punch lenny in the back of the"

# Iterate over the different temperature values

for temperature in [0.2, 0.5, 1.0, 1.2]:

print("\nGenerating text with riskiness : {}\n".format(temperature))

# Call the sample\_text function

print(sample\_text(seed, temperature))

<script.py> output:

Generating text with riskiness : 0.2

i'm gonna punch lenny in the back of the been a to the on the man to the mother and the father to simpson the father to with the marge in the for the like the fame to the been to the for my bart the don't was in the like the for the father the father a was the father been a say the been to me the do it and the father been to go. i want to the boy i can the from a man to be the for the been a like the father to make my bart of the father

Generating text with riskiness : 0.5

i'm gonna punch lenny in the back of the kin't she change and i'm all better it and the was the fad a drivera it? what i want to did hey, he would you would in your bus who know is the like and this don't are for your this all for your manset the for it a man is on the see the will they want to know i'm are for one start of that and i got the better this is. it whoce and i don't are on the mater stop in the from a for the be your mileat

Generating text with riskiness : 1.0

i'm gonna punch lenny in the back of the to to macks how screath. firl done we wouldn't wil that kill. of this torshmobote since, i know i ord did, can give crika of sintenn prescoam.whover my me after may? there's right. that up. there's ruining isay.oh.solls.nan'h those off point chuncing car your anal medion.hey, are exallies a off while bea dolk of sure, hello, no in her, we'll rundems... i'm eventy taving me to too the letberngonce

Generating text with riskiness : 1.2

i'm gonna punch lenny in the back of the burear prespe-nakes, 'lisa to isn't that godios.and when be the bowniday' would lochs meine, mind crikvin' suhle ovotaci!..... hey, a poielyfd othe flancer, this in are rightplouten of of we doll hurrs, truelturone? rake inswaydan justy!we scrikent.ow.. by back hous, smadge, the lighel irely.yes, homer. wel'e esasmoy ryelalrs all wronencay...... nank. i wenth makedyk. come on help cerzind, now, n

In [1]:

+100 XP

Very cool! You just generated some text with a neural network. Scroll through the output to see the text generated with different values of the temperature parameter. And congratulations on completing the course! If you're itching to continue learning about chatbots, remember to check out [**this tutorial**](https://www.datacamp.com/community/tutorials/facebook-chatbot-python-deploy) by Alan on how to connect your bot to Facebook Messenger!