WW (https://worldwritable.com/)

World Writable (https://worldwritable.com/)

Twitter (//twitter.com/@liza) LinkedIn (//linkedin.com/in/lizadaly)

Chatbot Fundamentals

An interactive guide to writing bots in Python

Part 4 of Natural Language Processing for Programmers (https://worldwritable.com/natural-language-processing-for-programmers-c21a4aff3cb9)

By Liza Daly (https://lizadaly.com/)

> "It is said that to explain is to explain away. This maxim is nowhere so well fulfilled as in the area of computer programming, especially in what is called heuristic programming and artificial intelligence...Once a particular program is unmasked, once its inner workings are explained in language sufficiently plain to induce understanding, its magic crumbles away; it stands revealed as a mere collection of procedures, each quite comprehensible. The observer says to himself, *I could have written that*."
>
> — Joseph Weizenbaum, ELIZA (1966) (https://www.csee.umbc.edu/courses/331/papers/eliza.html)

In this brief tutorial I'll walk you through using a popular Python language library to construct a simple chatbot that evaluates and responds to user input. It won't fool your friends, and for a production system you'll want to consider one of the existing bot platforms or frameworks, but these examples should help you think through the design and engineering challenges of a conversational UI.

About this tutorial

The source code presented here is interactive. You are strongly encouraged to modify the Python code — right in your browser—and experiment with the outcomes. (You may get a lot of error messages, but I promise you can't permanently break anything!) See Technical details below for more information on how the live code is implemented.

The boundaries of a bot

When you begin work on a conversational UI, even a trivial one, you'll need to answer these fundamental design questions:

1. Domain knowledge: What does a user expect this bot to understand?

2. Personality: What tone or vocabulary does the bot employ?

Domain knowledge

True artificial intelligence does not exist, so while some AIs can imitate humans quite convincingly or answer some kinds of factual questions, all bots are restricted to a subset of topics or conversational gambits. IBM's Jeopardy-playing Watson (http://www.nytimes.com/2011/02/17/science/17jeopardy-watson.html?pagewanted=print) "knew" facts and could construct realistic natural language responses, but it couldn't schedule your meetings or deliver your groceries. Simpler commercial bots like SlackBot (https://get.slack.help/hc/en-us/articles/202026038-Slackbot-your-assistant-notepad-programmable-bot) can successfully help users set up their Slack accounts, but aren't designed to engage you in open-ended dialogue.

Personality

Bots have historically been personified as something less than fully human to excuse their rote responses and frustrating lack of comprehension. This can be an opportunity for creativity and playful invention—the first bot I helped design was modelled after a famous parrot (http://inky.org/if/alex.html)—but it can also be a minefield of unexamined assumptions. It's disappointing that so many bots are personified as female (https://medium.com/@veronica/bots-and-gender-4ed9865fe2f2#.3u6tdsxlf) or teenagers (https://www.theguardian.com/technology/2016/mar/30/microsoft-racist-sexist-chatbot-twitter-drugs), as if those groups were naturally subservient (http://aworkinglibrary.com/writing/bots/) or not fully human. It's probably better for everyone if your bot is personified simply as itself—a computer program—or something truly non-human.

Often the dual axes of domain and personality align: in the program ELIZA, the *domain* was a therapy session, and the bot's *personality* was that of a Rogerian therapist (https://en.wikipedia.org/wiki/Person-centered_therapy). Domain and personality don't necessarily need to be tightly coupled, though—an ecommerce bot needs to know about products, sizing, and order status, but that domain doesn't imply any particular kind of personality. A shopping bot could have the persona of a helpful person, a cheerful kitten, or have no personality at all.

Meet "Brobot" 

Having warned you away from human personifications, I'm going to break my own rule and create a bot with a particular set of well-known personality traits and interaction models. I'll show you some introductory level chatbot techniques by writing software modeled after the **dialectical capabilities of a brogrammer (http://www.bloomberg.com/news/articles/2012-03-01/the-rise-of-the-brogrammer)**.

In this tutorial you can interact with Brobot by talking with it, and in some examples, you can override selected examples of its code to observe the effect on its behavior.

Start by greeting Brobot:

YOU: Type your chat message Chat

Hopefully, you said something like “Hello” and Brobot said something that sounded like a greeting in reply. For the “greet the robot” use case, we can use simple keyword matching, similar to how ELIZA and other early conversational UIs were modeled. Here’s the relevant code:

```
# Sentences we'll respond with if the user greeted us
GREETING_KEYWORDS = ("hello", "hi", "greetings", "sup", "what's up",)

GREETING_RESPONSES = ["'sup bro", "hey", "*nods*", "hey you get my snap?"]

def check_for_greeting(sentence):
    """If any of the words in the user's input was a greeting, return a greeting re
    for word in sentence.words:
        if word.lower() in GREETING_KEYWORDS:
            return random.choice(GREETING_RESPONSES)
```

This is the simplest possible implementation of a chatbot: it searches the user’s utterance for one or more known keywords and returns one of several possible responses. In practice you won’t want your bot to pick a truly random response—it’s better to cycle through a set of responses and avoid repeats. To keep the tutorial simple I’ve made Brobot completely stateless, so pure randomness will have to do.

Go ahead and modify the code above, right in the browser, to change Brobot’s behavior. Try returning only one response, or responding to more greetings. (If your code has an error, Brobot will pass along the Python message.)

YOU: Run code

Beyond keywords

Python programmers working with NLP have two great high-level libraries to choose from: TextBlob (http://textblob.readthedocs.io/en/dev/) and spaCy (https://spacy.io/). spaCy is easy to use and fast, though it can be memory intensive and doesn’t attempt to cover the whole of statistical NLP. TextBlob wraps the sprawling NLTK library (http://www.nltk.org/) in a very approachable API, so while it can be slower, it’s quite comprehensive. I’ll use TextBlob here, though see my article on text generation (https://worldwritable.com/natural-language-processing-for-programmers-90c4e04dc6de#.ghs8io8vs) for an example using spaCy.

```
    Given a parsed input, find the best pronoun, direct noun, adjective, and ver
    Returns a tuple of pronoun, noun, adjective, verb any of which may be None if t
    pronoun = None
    noun = None
    adjective = None
    verb = None
    for sent in parsed.sentences:
        pronoun = find_pronoun(sent)
        noun = find_noun(sent)
        adjective = find_adjective(sent)
        verb = find_verb(sent)
    logger.info("Pronoun=%s, noun=%s, adjective=%s, verb=%s", pronoun, noun, adject
    return pronoun, noun, adjective, verb
```

The main loop of Brobot performs the following steps:

1. Do some initial pre-processing of the user's text (this is a good place to hook in checks for unsafe input).
2. Ask TextBlob to parse the input for us.
3. Run a series of routines designed to extract the most information from the user's utterance in a structured way.
4. Compose a reply that best matches the user's statement.
5. Perform any post-processing to ensure as best we can that our bot isn't behaving badly.

Each of the `find_*` functions in `find_candidate_parts_of_speech()` consults TextBlob's `sentence.pos_tags` property, which returns the words' parts of speech. (You'll want to consult the Penn Treebank (https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html) reference to map the part-of-speech tag names to the actual grammatical constructs.)

Depending on the bot's domain, you're going to be more interested in some values than others, and you may also want to transform some of the input values or identify synonyms.

Here's some example code that identifies pronouns of interest to us:

```
def find_pronoun(sent):
    """Given a sentence, find a preferred pronoun to respond with. Returns None if
    pronoun is found in the input"""
    pronoun = None

    for word, part_of_speech in sent.pos_tags:
        # Disambiguate pronouns
        if part_of_speech == 'PRP' and word.lower() == 'you':
            pronoun = 'I'
        elif part_of_speech == 'PRP' and word == 'I':
            # If the user mentioned themselves, then they will definitely be the pr
            pronoun = 'You'
    return pronoun
```

I'm going to look for pronouns like "you" or "I" and infer from those that the user wants to talk about themselves or the bot. When identified, I invert them—if the user says "you", Brobot responds with "I".

A more sophisticated approach would be to build a dependency tree (https://class.coursera.org/nlp/lecture/175). Dependency grammars describe the relationship among all clauses in a sentence, allowing you to discriminate between (say) the subject and object of a sentence. If your bot needs to know the difference between "dog bites man" and "man bites dog", I recommend using the dependency parsing function of a library like spaCy (https://spacy.io/).

Try adding a special case to allow the user to address 'Brobot' by name in addition to 'you' to set up a response that refers to the bot itself.

YOU: Run code

But enough about me, what do you think of me?

Like all brogrammers, our bot loves to talk about itself, so if the user mentions it anywhere in their input, it'll reply about itself and include a token amount of your input to pretend as if it were listening. This special case routine is fired if the user addressed the bot (by mentioning "you" in their input), and if so, shortcuts all other potential responses:

```
def check_for_comment_about_bot(pronoun, noun, adjective):
    """Check if the user's input was about the bot itself, in which case try to fas
    that feels right based on their input. Returns the new best sentence, or None."
    resp = None
    if pronoun == 'I' and (noun or adjective):
        if noun:
            if random.choice((True, False)):
                resp = random.choice(SELF_VERBS_WITH_NOUN_CAPS_PLURAL).format(**{'n
            else:
                resp = random.choice(SELF_VERBS_WITH_NOUN_LOWER).format(**{'noun':
        else:
            resp = random.choice(SELF_VERBS_WITH_ADJECTIVE).format(**{'adjective':
    return resp

# Template for responses that include a direct noun which is indefinite/uncountable
SELF_VERBS_WITH_NOUN_CAPS_PLURAL = [
```

In a real bot, you'd want to compose responses using a more sophisticated templating engine (http://jinja.pocoo.org/) or maybe even a full-blown Context-Free Grammar (https://worldwritable.com/natural-language-processing-for-programmers-90c4e04dc6de#.duv4q8h8q).

Try coming up with routines that could use more than one term from the user's input and still produce sensible output in most cases. Consider the constraints that tense, spelling, and number agreement will introduce.

YOU: Run code

Constructing a realistic response

The most common case will be that the user supplies sensible input that the program can parse into component words, but none of those words trigger a special case like greeting or referencing the bot. Put another way, the program knows the user said something, but doesn't "understand" what they said, because their input fell outside of its domain knowledge.

In a purely transactional bot, there isn't much to do at this point besides return some help text ("You can ask me about booking a flight, changing a reservation, etc."). In a more conversational bot, you can still manipulate the user's input to generate a successful response, but it's more apt to be one that reflects the bot's personality than its understanding of the world.

In the ELIZA simulation, the bot reflected the user's input back to them in a gently inquiring way. Because this is a brogrammer, it's going to try to neg or dismiss the user. We first check for a special case where the user talked about themselves, and if so negate the verb and assert that whatever they said wasn't true.

If they said anything else, the bot will just mindlessly echo what they said, adding some filler bro-words at the end. Like a real brogrammer, our bot is limited in its intellectual capability and mostly regurgitates aphorisms it saw elsewhere, like LinkedIn.

```
def construct_response(pronoun, noun, verb):
    """No special cases matched, so we're going to try to construct a full sentence
    of the user's input as possible"""
    resp = []

    if pronoun:
        resp.append(pronoun)

    # We always respond in the present tense, and the pronoun will always either be
    # from the user, or 'you' or 'I', in which case we might need to change the ten
    # irregular verbs.
    if verb:
        verb_word = verb[0]
        if verb_word in ('be', 'am', 'is', "'m"):  # This would be an excellent pla
            if pronoun.lower() == 'you':
                # The bot will always tell the person they aren't whatever they sai
```

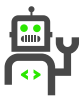
1. If we identified a pronoun from the user, re-use that.

2. In most cases, we pass through the user's verb unchanged.
3. If the verb was "to be" and the user was talking about themselves ("I am a good programmer"), the bot will neg them by inverting the meaning of the verb and claiming they aren't whatever they asserted.
4. Otherwise, just reconstruct the base words from the user's original sentence—subject, verb, object—and add some bro-ish filler.

In this code, I manually match all the irregular forms of "to be", but a more flexible approach would be to convert the user's verb to a lemma (https://www.quora.com/Dictionaries-In-WordNet-whats-the-difference-between-a-sense-and-a-lemma). Stems and lemmas are great shortcuts to mapping a range of potential input to some known value; see also senses and similarity matching (https://spacy.io/blog/sense2vec-with-spacy). Both techniques require more horsepower than I could allocate to little Brobot, but don't require much code when using NLP libraries.

Try telling the bot "I am [something]" and verify that it disagrees with you. How could you enhance this behavior?

YOU: Run code



You've got to be kind

The last routine run by any bot should be a filter to limit unpleasant or unsafe output. Just as we should have filtered incoming input to prevent foreign code execution or (maybe) offensive language, we want to ensure that the bot doesn't say things that are harassing or contextually inappropriate. The PR fallout from neglecting this step can be considerable (http://www.theverge.com/2016/3/24/11297050/tay-microsoft-chatbot-racist).

In many ways, this is a doomed exercise from the start. Security experts will confirm that there is no sure-fire way to sanitize unrestricted user input. (For example, I can't truly prevent you from putting destructive Python code into this tutorial, but it's deployed on a transient backend with no permanent storage, no internet access, and nothing connected to me personally.)

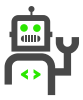
But even if it is *theoretically* impossible to prevent a bad bot, as bot creators we have an ethical obligation to at least try. For Twitter bots, this means not DMing or @-messaging other users. For Slack bots, we should limit the permissions allocated to the bot to prevent it from issuing commands. And for all bots, it means performing checks against offensive words and phrases before allowing the bot to parrot back user input in a harmful way.

```
def filter_response(resp):
    """Don't allow any words to match our filter list"""
    tokenized = resp.split(' ')
    for word in tokenized:
        if '@' in word or '#' in word or '!' in word:
            raise UnacceptableUtteranceException()
        for s in FILTER_WORDS:
            if word.lower().startswith(s):
                raise UnacceptableUtteranceException()
```

A very simple filter against a list of known offensive terms (https://github.com/dariusk/wordfilter) is a good first start, as is removing potentially dangerous characters like '@' or '#' that are meaningful on Twitter.

Confirm that Brobot won't repeat words that being with a hashtag or at-sign. What other kinds of filters would you need in your chat environment to minimize abuse?

YOU: Run code



Go forth and crush it

I covered most of the functional parts of Brobot, but please review the complete source code (https://github.com/lizadaly/brobot/). In most real-world cases, you'll want to move from the prototype stage to a full-blown messaging environment. You may even want to scrap your NLP-based work and start over using existing grammars and libraries for specific chatbots. But I encourage you to start with the fundamentals—I particularly recommend a test-first approach (https://github.com/lizadaly/brobot/blob/master/test_broize.py), as it's a natural fit for conversational UIs.

Technical details

There are several ways to run a Python interpreter in a web browser (http://pypyjs.org/), but those methods typically limit one to the Python native library. That's fine for learning Python itself, but it would preclude tutorials like this that require complex third-party libraries like TextBlob. The journal Nature first pioneered running Jupyter Notebooks in the browser (http://www.nature.com/news/interactive-notebooks-sharing-the-code-1.16261) using Docker as the backend.

This tutorial takes a different approach: AWS Lambda (https://aws.amazon.com/lambda/) provides highly scalable, inexpensive, short-lived Python sessions that can be reached via a lightweight API. Using Lambda eliminates the need for cumbersome Docker container maintenance, and is essentially free for low-traffic use.

Further reading and credits

- An excellent three-part series on ELIZA (http://www.filfre.net/2011/06/eliza-part-1/)
- Robot icon (https://thenounproject.com/term/robot/28693/) originally by Dan Hetteix from the Noun Project
- CodeMirror (https://codemirror.net/) provides the inline text editor
- Visual inspiration for Brobot (http://classicprogrammerpaintings.com/image/143796086956)

- Thanks Dan, Andrew, and Max for feedback and suggestions