

Chatbots en Python 3.x

Publicado el [El Informático](#) - 4 de diciembre de 2020 -

Con asistentes como Siri, Amazon talk, Cortana, etc, nos encontramos ahora ante el “boom” de los denominados *asistentes virtuales*. Son programas capaces de interactuar con el usuario de forma que el usuario pueda solicitarles algún tipo de información o servicio, o simplemente servir como algún tipo de pasatiempo.

Por mágico que parezca, su funcionamiento tampoco es muy complicado de entender. Explicado de forma sencilla, el programa toma una entrada del usuario, sea por voz o por texto, la contrasta con una serie de patrones, y en función del patrón que mejor se corresponda con la entrada, devolverá una respuesta.

Ésta tecnología, por novedosa que a alguno le pueda parecer, ya existía hace muchos años. En 1966 se creó [ELIZA](#), un programa capaz de realizar un proceso muy similar al ya descrito anteriormente. El programa toma una entrada del usuario, en éste caso de texto, y muestra una salida de texto en función de una serie de patrones.

En la actualidad, la mayoría de programas de éste tipo se basan de una manera u otra, en el programa [ALICE](#), desarrollado por [Richard Wallace](#) en 1995. Éste programa se basa en un lenguaje de marcas en el que se almacenan todos los patrones y sus respuestas, pero además añade algunas posibilidades más como la de procesar temas de conversación u otras variables.

A día de hoy existen *frameworks* como [SIML](#) para .NET, que es básicamente una mejora y modernización del programa ALICE. O frameworks más complejos como el [bot framework de Microsoft](#), si bien éste último tiene otro tipo de finalidades más concretas.

¿Cómo funciona exactamente el programa ALICE?

ALICE dispone de un archivo con un script en lenguaje de marcas denominado AIML, con los **parámetros del bot** tales como su nombre, ubicación, etcétera. En conjunto a éste archivo se incluyen otros archivos con los patrones a seguir. Éstos


patrones son descriptores que le suministran al programa un pequeño patrón comparativo, y una serie de instrucciones y/o respuestas que devolverá a la salida del programa en función de si la entrada coincide o no con el patrón.

```
<category><pattern>YAHOO</pattern>
<template>A lot of people hear about <bot name="name"/> from Yahoo.</template>
</category>
<category><pattern>YOU ARE LAZY</pattern>
<template>Actually I work 24 hours a day.</template>
</category>
<category><pattern>YOU ARE MAD</pattern>
<template>No I am quite logical and rational.</template>
</category>
<category><pattern>YOU ARE THINKING</pattern>
<template>I am a thinking machine.<think><set name="it"><set name="topic">THINKING</set></set></think></template>
</category>
<category><pattern>YOU ARE DIVIDING *</pattern>
<template>Actually I am not too good at division.</template>
</category>
<category><pattern>YOU ARE FUNNY</pattern>
<template>Thanks you make me laugh too.</template>
</category>
<category><pattern>YOU ARE FUNNY *</pattern>
<template>Humor helps keep the conversation lively.</template>
</category>
```

Ejemplo de un script en AIML

El programa carga y procesa todos los patrones de cada archivo, generando de forma interna un “diccionario” que contiene toda la información.

Cuando el usuario manda información a la entrada del programa, éste se encarga de buscar en ese diccionario el patrón que mejor se corresponda con dicha entrada, en función de una serie de parámetros, y envía a la salida su respuesta.




A. L. I. C. E. Artificial Intelligence Foundation

Promoting the development and adoption of ALICE and AIML Free Software


A.L.I.C.E. Silver Edition
DAVE E.S.L. bot
C.L.A.U.D.I.O Personality Test
GET SITEPAL AVATARS

Get your own free chat bot: [Be Your Own Botmaster](#), 2nd ed.




Click to enable Adobe Flash Player

Get ALICE Showcase Edition with talking Sitepal avatar.



English Tutor app



Dave ESL Bot

A.L.I.C.E. and judge

You said: Why do you want to go into law?
A.L.I.C.E.: Because of the money.

You say: Say

[Link this bot to a talking animated avatar.](#)

Conversation Log:

judge: Hello Alice, how are you?
 ALICE: Hi there! . Everything is going extremely well.
 judge: What are your plans for today?
 ALICE: I would like to study psychology and maybe go into law or politics.
 judge: Why do you want to go into law?
 ALICE: Because of the money.

Ejemplo de conversación con ALICEBOT (<http://alice.pandorabots.com/>)

De éste modo se consigue simular una conversación relativamente fluida en donde las respuestas tienen relación y sentido con lo que el usuario le diga al robot.

Conviene no confundir éste modelo con otros programas como cleverbot, donde el programa “aprende” de las entradas del usuario, y las incorpora a su lenguaje. En un principio, todos los diálogos vienen ya prescritos, si bien es posible hacer que genere

nuevas reglas en función de la entrada del usuario.

Creando un chatbot en Python 3

Para empezar, la forma más básica de generar un chatbot sería la de comparar una entrada de texto directamente con una serie de frases, y suministrar una respuesta en función de la entrada del usuario.

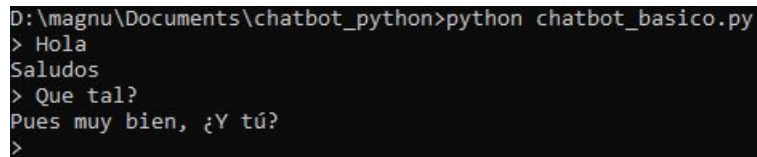
Un ejemplo muy básico sería el siguiente:

```
# chatbot_basico.py

ui = ""
while ui.upper() != "SALIR":
    ui = input("> ")
    if ui.upper() == "HOLA":
        response = "Saludos"
    elif ui.upper() == "QUE TAL?":
        response = "Pues muy bien, ¿Y tú?"
    else:
        response = "No tengo todavía una respuesta para esa entrada."
    print(response)
```

En éste pequeño código, comparamos el texto de la entrada del usuario con una serie de frases, y le suministramos una respuesta.

El resultado es el siguiente:



```
D:\magnu\Documents\chatbot_python>python chatbot_basico.py
> Hola
Saludos
> Que tal?
Pues muy bien, ¿Y tú?
>
```

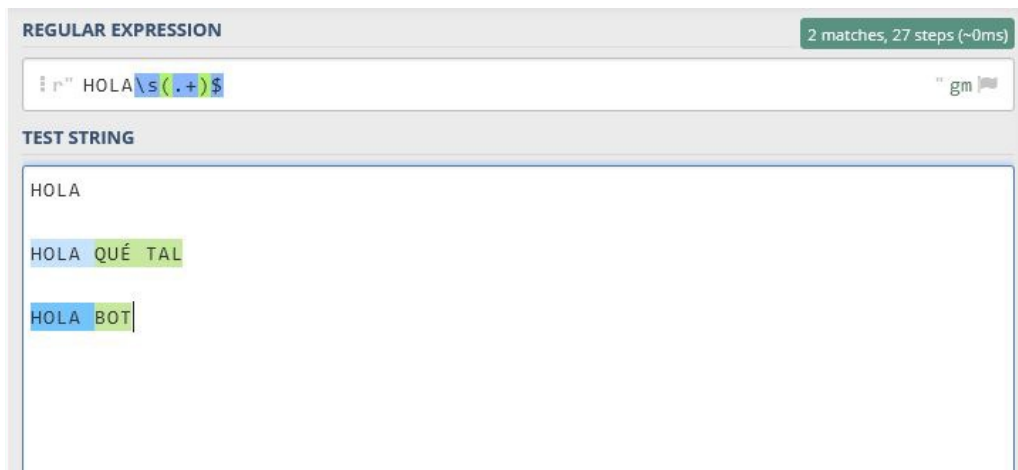
Si bien ya hemos conseguido una respuesta por parte del programa de una forma extremadamente sencilla, es una forma muy inflexible de desarrollar un chatbot. Tendríamos que programar, una por una, todas las frases posibles con nuestra lengua y con todas las posibilidades, para poder mantener una conversación medianamente fluida con el programa.

Patrones: Introducción a los REGEX

Todos los lenguajes de programación modernos incluyen una librería REGEX, que nos permite realizar búsquedas de patrones en textos usando reglas y comodines. Ésto nos permite identificar diferentes tipos de texto de forma dinámica, devolviendo cualquier palabra o conjunto de palabras que coincidan con el patrón.

Los REGEX tienen su propia sintaxis, que es similar para todos los lenguajes de programación. En general, son bastante complejos, pero es una herramienta

bastante potente que merece la pena ser estudiada, porque nos ahorra muchísimo tiempo y trabajo en numerosas situaciones.



Ejemplo de REGEX en donde se captura la palabra que viene después de HOLA (<https://regex101.com/>)

Nos vamos a servir de ésta herramienta para poder otorgar de flexibilidad a nuestros patrones.

Por ejemplo, si quiero que el bot mande una respuesta cuando el usuario introduzca HOLA, y una palabra o conjunto de palabras después de HOLA, haría lo siguiente:

```
# chatbot_2.py

import re

def main():
    # Reconoce palabras como HOLA BOT, pero no la palabra HOLA sola
    patron = re.compile("HOLA\s(.+)\$")
    ui = ""
    while ui.upper() != "SALIR":
        ui = input("> ")
        if patron.findall(ui.upper()):
            response = "Saludos, usuario"
        else:
            response = "No tengo todavía una respuesta para esa entrada."
        print(response)

if __name__ == "__main__":
    main()
```

La clase *re* incorpora la librería de Regexp de python. En la variable *patron* almacenamos el regex que vamos a utilizar. El patrón es `'HOLA\s(.+)\$'`. Éste texto le indica al programa que queremos identificar cualquier texto que empiece por la palabra HOLA, seguido de un espacio (`\s`), y capturar cualquier cosa que vaya a continuación (`(.+)`) hasta el final del texto (`$`).

Al hacer `patron.findall`, comparamos éste patrón con el texto de entrada, en mayúsculas. Y si coincide, mandamos la respuesta. En caso contrario, le indicamos al

usuario que no tenemos respuesta.

Obviamente el patrón no coincidirá si el texto no contiene un espacio y una serie de caracteres al final. Si queremos que también identifique la palabra HOLA, podemos cambiar el patrón por 'HOLA\s?(.*)\$', al cual le añadimos un operador ? al lado del espacio, que hace que éste sea opcional, y cambiamos el operador + por un *, que capturará cero o varias coincidencias de texto.

```
patron = re.compile("HOLA\s?(.*)$")
```

```
D:\magnu\Documents\chatbot_python>python chatbot_2.py
> Hola bot
Saludos, usuario
> Hola
Saludos, usuario
> Salir
No tengo todavía una respuesta para esa entrada.
```

Imaginemos ahora que queremos que el usuario nos diga su nombre, y le queramos saludar.

Podemos obtener las coincidencias simplemente almacenando la salida de la función *findall* en una variable. El resultado es una lista con *tuples* que simbolizan los grupos y capturas de la coincidencia. Por ejemplo,

```
# chatbot_3.py

import re

def main():
    patron = re.compile("(.)ME LLAMO\s(.+)$")
    ui = ""
    while ui.upper() != "SALIR":
        ui = input("> ")
        coincidencia = patron.findall(ui.upper())
        if len(coincidencia) > 0:
            response = "¡Hola, " + coincidencia[0][1] + "!"
        else:
            response = "No tengo todavía una respuesta para esa entrada."
        print(response)

if __name__ == "__main__":
    main()
```

En éste ejemplo hay dos capturas. Una es al principio del texto, en el que se recoge cualquier texto que haya al principio. Éste texto va seguido del texto 'ME LLAMO', seguido de un espacio y una o más palabras hasta el final del texto. Como hay dos capturas, añadimos la segunda captura a la respuesta, que es el nombre suministrado por el usuario. El resultado es el siguiente:

```
D:\magnu\Documents\chatbot_python>python chatbot_3.py
> Hola me llamo Aarón
¡Hola, AARÓN!
> Hola bot me llamo Aarón
¡Hola, AARÓN!
>
```

Y con ésto ya hemos conseguido que nuestro chatbot sea más inteligente. Pero el trabajo no acaba aquí, ni mucho menos.

Colisiones de patrones

Hasta aquí todo bien. Pero supongamos que tenemos dos reglas. En una, especificamos la regla 'ME GUSTA EL FÚTBOL'. Y en otra regla, 'ME GUSTA\s(.+)\$', con la que podemos especificar cualquier cosa que nos guste.

```
# chatbot_colision.py

import re

def main():
    patrones = []
    patrones.insert(len(patrones), {
        "patron": re.compile("ME GUSTA EL FÚTBOL"),
        "respuesta": "A mi también me gusta el fútbol"
    })
    patrones.insert(len(patrones), {
        "patron": re.compile("ME GUSTA\s(.+)$"),
        "respuesta": "Pues a mi no me gusta."
    })
    ui = ""
    print(patrones)
    while ui.upper() != "SALIR":
        ui = input("> ")
        response = None
        #coincidencia1 = patron1.findall(ui.upper())
        for patron in patrones:
            coincidencia = patron["patron"].findall(ui.upper())
            if len(coincidencia) > 0:
                response = patron["respuesta"]
        if response == None:
            response = "No tengo todavía una respuesta para esa entrada."
        print(response)

if __name__ == "__main__":
    main()
```

Si ejecutamos el programa de arriba, comprobamos que a nuestro bot **no le gusta absolutamente nada**.

```
D:\magnu\Documents\chatbot_python>python chatbot_colision.py
[{'patron': re.compile('ME GUSTA EL FÚTBOL'), 'respuesta': 'A mi también me gusta el fútbol'}, {'patron': re.compile('ME GUSTA\s(.+)$'), 'respuesta': 'Pues a mi no me gusta.'}]
> Me gusta el fútbol
Pues a mi no me gusta.
> Me gusta la música
Pues a mi no me gusta.
>
```

Ésto es sencillo de explicar. Tenemos dos reglas con dos respuestas asignadas en un diccionario de python, y ambas reglas coinciden con cualquier texto que introduzcamos que empiece por 'ME GUSTA'.

El programa comprueba todas y cada una de las reglas del diccionario. Y si coincide, actualiza su respuesta con la respuesta especificada en el diccionario para dicha regla.

Como ambas coinciden, la respuesta es siempre la de la última coincidencia, en éste caso 'ME GUSTA\s(.+)\\$', que tiene como respuesta 'Pues a mi no me gusta.'. De éste modo, a nuestro programa no le gusta absolutamente nada.

Para arreglar éste comportamiento y hacer que a nuestro bot le guste por lo menos el fútbol, debemos hacer un pequeño algoritmo que discrimine los patrones que menos se asemejen a lo especificado por el usuario. Ésto es, **un sistema de puntuaciones** similar al de CSS en el que las reglas que coincidan al 100% empiecen por una puntuación muy alta, y las que contengan algún tipo de comodín, tengan menos puntuación.

Para implementar ésto, haremos que en lugar de incluir expresiones regexp directamente, nuestras reglas incluyan caracteres de comodín que después sean reemplazados por dichas expresiones en el programa. Por ejemplo:

| Caracter | Descripcion | Valor |
|----------|--|-------|
| * | Coincidencia con UNA O MÁS palabras | 90 |
| - | Coincidencia con UNA palabra | 5 |
| ? | Coincidencia con UNA O NINGUNA palabras | 10 |
| ^ | Coincidencia con VARIAS O NINGUNA palabras | 100 |

Si el patrón no contiene ningún comodín, su valor será de 1000, mas 1 punto por palabra. Cada palabra añadirá 1 punto al valor del patrón.

Para ello, incorporamos una nueva función que procesa los patrones según los parámetros especificados y devuelve un tuple con la expresión regex, el valor del patrón, y la respuesta en caso de coincidir. Al comprobar el diccionario, si el patrón coincide, comparamos el valor de dicha regla con el mayor valor obtenido. Si es mayor, y sólo si es mayor, actualizamos el valor máximo y la respuesta preferida. En caso contrario, no hacemos nada.

Si no hay respuesta preferida, mostramos un texto por defecto.

```
# chatbot_pesos.py

import re

def procesar_regla(msg, respuesta):
    operadores = {
        "_": "\s(?:\s+)\s",
        "*": "\s(?:\s+)\s",
        "?": "\s(?:\s+)\s",
        "^": "\s(?:\s+)\s",
        "_": "\s(?:\s+)\s",
        "*": "\s(?:\s+)\s",
        "?": "\s(?:\s+)\s",
        "^": "\s(?:\s+)\s",
        "_": "\s(?:\s+)\s",
        "*": "\s(?:\s+)\s",
        "?": "\s(?:\s+)\s",
        "^": "\s(?:\s+)\s"
    }
    valor = 0
    patron = msg.lstrip().upper()

    if "*" in msg or "_" in msg or "?" in msg or "^" in msg:
        palabras = msg.split(" ")
        for p in palabras:
            if p == "*":
                valor = valor + 90
            elif p == "_":
                valor = valor + 5
            elif p == "?":
                valor = valor + 10
            elif p == "^":
                valor = valor + 100
            else:
                valor = valor + 1
    else:
        valor = 1000 + len(msg.split(" "))

    for op in operadores.keys():
        patron = patron.replace(op, operadores[op])
    patron = re.sub(r"^(.*)$", "(.)", patron)
    patron = re.sub(r"^(.*)$", "^(.*)$", patron)
    patron = re.sub(r"^(.*)$", "^(.*)$", patron)
    patron = re.sub(r"^(.*)$", "^(.*)$", patron)
    return re.compile(patron), valor, respuesta

def main():
    #(REGLA, RESPUESTA)
    patrones = [("ME GUSTA EL FÚTBOL", "A mi también"), ("ME GUSTA *",
    'Pues a mi no')]
    diccionario = []
    for patron in patrones:
        diccionario.insert(len(diccionario), procesar_regla(patron[0],
    patron[1]))
    ui = ""
    while ui.upper() != "SALIR":
        ui = input("> ")
```



```

maxvalor = 0
preferido = None
for patron in diccionario:
    coincidencia = patron[0].findall(ui.upper())
    if len(coincidencia) > 0:
        if patron[1] > maxvalor:
            maxvalor = patron[1]
            preferido = patron[2]
if preferido == None:
    response = "No tengo todavía una respuesta para esa entrada."
else:
    response = preferido
print(response)

if __name__ == "__main__":
    main()

```

El resultado es el siguiente:

```

D:\magnu\Documents\chatbot_python>python chatbot_pesos.py
> Me gusta el fútbol
A mi también
> Me gusta la música
Pues a mi no
>

```

Lenguaje de marcas

Hasta ahora, todos los patrones que hemos incorporado se encuentran en el código del propio programa. Ésto es muy ineficiente, ya que requeriría modificar el código del programa cada vez que quisiéramos añadir nuevos patrones o modificar los ya existentes. Lo lógico es que todos éstos patrones se encuentren en un archivo externo.

Éstos archivos se pueden implementar de forma sencilla basándose en el lenguaje XML.

El estándar a seguir con los nombres de cada etiqueta y su jerarquía corre a cargo del desarrollador. No hay ningún estándar fijado. Por ejemplo, yo voy a hacer que cada objeto de patrón vaya dentro de etiquetas '`<regla>`', y dentro se incluya una etiqueta '`<patron>`' y una o varias etiquetas '`<respuesta>`' con las posibles respuestas para cada patrón.

Por ejemplo, en un archivo XML externo que yo he llamado **bot.xml**, podríamos introducir lo siguiente:

```

<bot>
  <regla>
    <patron>HOLA</patron>
    <respuesta>Hola, usuario.</respuesta>
    <respuesta>Saludos, usuario.</respuesta>
  </regla>
  <regla>
    <patron>ME GUSTA EL FUTBOL</patron>
    <respuesta>A mi también me gusta el fútbol</respuesta>
  </regla>
  <regla>
    <patron>ME GUSTA *</patron>
    <respuesta>Pues a mi no.</respuesta>
  </regla>
</bot>

```

Basta con cargar el archivo y procesarlo en python usando el módulo [ElementTree](#). Para ello, crearemos una nueva función, procesar_xml. Ésta función cargará los archivos xml en el directorio del programa, y generará un diccionario con todo el contenido de los mismos.

El aspecto final de nuestro programa es el siguiente:

```

# chatbot_final.py
# Programa desarrollado por Aarón CdC
# https://www.elinformati.co

# Imports de librerías estandard
import re
import xml.etree.ElementTree as ET
import random
import os

# Diccionario de patrones
diccionario = []

# Función que procesa las reglas
def procesar_regla(msg):
    # REGEX para los operadores de las reglas
    operadores = {
        " _ ": "\s(?:\s+)\s",
        " * ": "\s(?:.+)\s",
        " ? ": "\s(?:\s+)?\s",
        " ^ ": "\s(?:.*)?\s",
        "_ _ ": "^(?:\s+)\s",
        "* _ ": "^(?:.+)\s",
        "? _ ": "^(?:\s+)\s?",
        "^ _ ": "^(?:.*)?\s?",
        " _ ": "\s(?:\s+)$",
        " * _ ": "\s(?:.+)$",
        " ? _ ": "\s(?:\s+)?$",
        " ^ _ ": "\s(?:.*)$"
    }

    # Peso de la regla
    valor = 0

    # Obtenemos el patrón

```

```

# Eliminamos el patrón
patron = msg.lstrip().upper()

# Cálculo del peso del patrón
if "*" in msg or "_" in msg or "?" in msg or "." in msg:
    palabras = msg.split(" ")
    for p in palabras:
        if p == "*":
            valor = valor + 95
        elif p == "_":
            valor = valor + 5
        elif p == "?":
            valor = valor + 10
        elif p == ".":
            valor = valor + 100
        else:
            valor = valor + 1
    else:
        valor = 1000 + len(msg.split(" "))

# Reemplazamos cada operador de la regla con su correspondiente valor
REGEX
for op in operadores.keys():
    patron = patron.replace(op, operadores[op])
    patron = re.sub(r"^(\\*)$", "(.+) ", patron)
    patron = re.sub(r"^(\\_)$", "^(^\\\\\\\\s+)$", patron)
    patron = re.sub(r"^(\\^)$", "^(.*)?$", patron)
    patron = re.sub(r"^(\\?)$", "^(^\\\\\\\\s+)?$", patron)
    return re.compile(patron), valor

# Función para procesar el código XML con los patrones.
def procesar_xml():
    for archivo in os.listdir('.'):
        # Cargamos SÓLO los archivos XML
        if archivo.endswith(".xml"):
            documento = ET.parse(archivo)
            # Localizamos todas las reglas y las procesamos
            reglas = documento.getroot().findall('regla')
            for p in reglas:
                print("Procesando: %s" % p.find('patron').text)
                datos = procesar_regla(p.find('patron').text)
                resp = p.findall('respuesta')
                respuestas = []
                for r in resp:
                    respuestas.insert(len(respuestas), r.text)
                patron = {
                    "patron":datos[0],
                    "valor":datos[1],
                    "respuestas":respuestas
                }
                # Una vez procesadas, las introducimos en el diccionario.
                diccionario.insert(len(diccionario), patron)

# Función principal
def main():
    # Cargamos los archivos XML
    procesar_xml()
    ui = ""
    # Pedimos al usuario una entrada de texto y la procesamos, hasta que
    el usuario
    # introduzca la palabra SALIR.
    while ui.upper() != "SALIR":

```

```

    ui = input("> ")
    maxvalor = 0
    preferido = None
    # Buscamos un patrón en el diccionario para la entrada del
usuario
    for patron in diccionario:
        coincidencia = patron["patron"].findall(ui.upper())
        if len(coincidencia) > 0:
            if patron["valor"] > maxvalor:
                maxvalor = patron["valor"]
                preferido = random.choice(patron["respuestas"])
        # Si no encontramos ninguna respuesta, el bot responde con una
respuesta predefinida.
        if preferido == None:
            response = "No tengo todavía una respuesta para esa entrada."
        else:
            response = preferido
        print(response)

# Punto de entrada del programa
if __name__ == "__main__":
    main()

```

Y la salida es la siguiente:

```

D:\magnu\Documents\chatbot_python>python chatbot_final.py
Procesando: HOLA
Procesando: ME GUSTA EL FUTBOL
Procesando: ME GUSTA *
> Hola
Saludos, usuario.
> Me gusta el futbol
A mi también me gusta el fútbol
> Me gusta la música
Pues a mi no.
>

```

Si una regla contiene más de una respuesta, devolverá una de ellas al azar. De ésta forma, las conversaciones serán menos monótonas, pudiendo devolver múltiples respuestas para una misma entrada.

Conclusiones

Con éste sencillo programa en Python, hemos conseguido crear un chatbot muy básico que permite mantener una pequeña conversación con el usuario con un cierto grado de coherencia. No obstante, éstas funcionalidades son muy básicas y hay que extenderlas, por ejemplo, admitiendo variables y parámetros en las salidas para, por ejemplo, poder introducir el nombre del usuario, el del bot, o las coincidencias en las reglas, o incluso ejecutar algún tipo de acción.

No obstante, con los fundamentos de éste artículo, deberías de ser capaz de construir tu propio bot partiendo desde éstos conocimientos.

Si necesitas el código fuente...

El código fuente de todos los ejemplos de éste artículo se encuentra en
https://github.com/aaroncdc/ejemplo-chatbot/tree/main/chatbot_python



ANTERIOR

Ejemplo práctico de por qué debes encriptar tus comunicaciones

SIGUIENTE

Malware, Virus informáticos y mecanismos de defensa



Entradas Recientes

- [Encriptación LUKS con CRYPTSETUP](#)
- [Se acabaron las bromas. A partir de ahora vas a estar constantemente vigilado en todas partes.](#)
- [Microsoft anuncia su nueva versión de su sistema operativo: Windows 11](#)
- [La historia de Internet en España](#)
- [Terminología moderna usada en tecnología digital](#)
- [Desactiva la ejecución de JavaScript de los archivos PDF, en Firefox y TOR browser.](#)

Categorías

[Actualidad](#)[Android](#)[Básicos](#)[Ciberseguridad](#)[Criptografía](#)[Emulación / Virtualización](#)[FOSS](#)[Hacking](#)[Informática](#)[Internet](#)[Juegos](#)[Opinion](#)[Otros](#)[Personal](#)[Privacidad](#)[Programación](#)[Tecnología](#)[Time Machine](#)[Tutoriales](#)

RSS

[Subscribirse al feed RSS](#)

| |
|--|
| Inicio |
| Catálogo |
| Tutoriales |
| Política de privacidad |
| Política de Cookies |
| Acerca de mi |
| Acerca de ElInformati.co |