INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

MATERIA: Teoría Computacional.

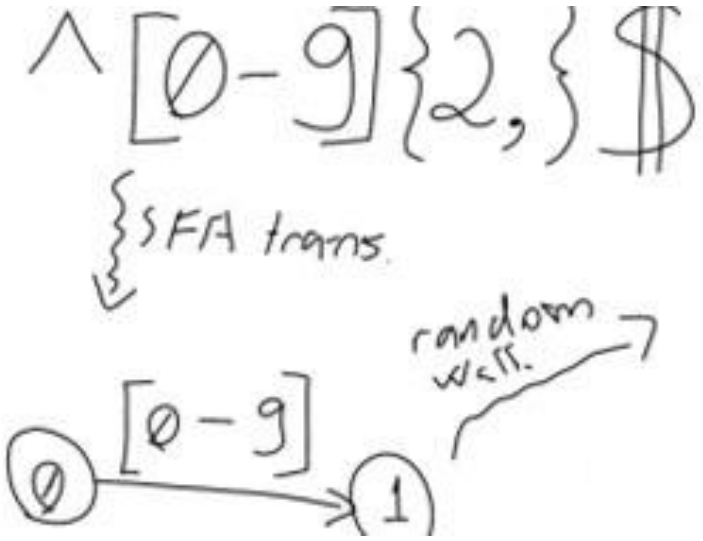
PRÁCTICA: Práctica 3. Expresiones Regulares.

ALUMNO: Meza Vargas Brandon David.

PROFESOR: Jorge Luis Rosas Trigueros.

FECHA PRÁCTICA: 06-nov-2020

FECHA DE ENTREGA: 13-nov-2020



MARCO TEÓRICO

Expresiones Regulares

Las expresiones regulares, a menudo llamada también regex, son unas secuencias de caracteres que forma un patrón de búsqueda, las cuales son formalizadas por medio de una sintaxis específica. Los patrones se interpretan como un conjunto de instrucciones, que luego se ejecutan sobre un texto de entrada para producir un subconjunto o una versión modificada del texto original. Las expresiones regulares pueden incluir patrones de coincidencia literal, de repetición, de composición, de ramificación, y otras sofisticadas reglas de reconocimiento de texto.

¿Cómo funciona una expresión regular?

Una expresión regular puede estar formada, o bien exclusivamente por caracteres normales (como *abc*), o bien por una combinación de caracteres normales y metacaracteres (como *ab*c*). Los metacaracteres describen ciertas construcciones o disposiciones de caracteres: por ejemplo, si un carácter debe estar en el inicio de la línea o si un carácter solo debe o puede aparecer exactamente una vez, más veces o menos.

Metacaracteres

Metacaracteres – delimitadores: Esta clase de metacaracteres nos permite delimitar dónde queremos buscar los patrones de búsqueda. Ellos son:

Metacaracter	Descripción
^	inicio de línea.
$	fin de línea.
\A	inicio de texto.
\Z	fin de texto.
.	cualquier carácter en la línea.
\b	encuentra límite de palabra.

Metacaracter	Descripción
\B	encuentra distinto a límite de palabra

Metacaracteres – clases predefinidas: Estas son clases predefinidas que nos facilitan la utilización de las expresiones regulares. Ellos son:

Metacaracter	Descripción
\w	un carácter alfanumérico (incluye "_").
\W	un carácter no alfanumérico.
\d	un carácter numérico.
\D	un carácter no numérico.
\s	cualquier espacio (lo mismo que [\t\n\r\f]).
\S	un no espacio.

Metacaracteres – iteradores: Cualquier elemento de una expresión regular puede ser seguido por otro tipo de metacaracteres, los iteradores. Usando estos metacaracteres se puede especificar el número de ocurrencias del carácter previo, de un metacaracter o de una sobrexpresión. Ellos son:

Metacaracter	Descripción
*	cero o más, similar a {0,}.
+	una o más, similar a {1,}.
?	cero o una, similar a {0,1}.
{n}	exactamente n veces.
{n,}	por lo menos n veces.
{n,m}	por lo menos n pero no más de m veces.
*?	cero o más, similar a {0,}?.
+?	una o más, similar a {1,}?.
??	cero o una, similar a {0,1}?.
{n}?	exactamente n veces.
{n,}?	por lo menos n veces.
{n,m}?	por lo menos n pero no más de m veces.

Comando grep en Linux

El comando grep perteneciente a la familia Unix es una de las herramientas más versátiles y útiles disponibles. Este busca un patrón que definamos en un archivo de texto. En otras palabras, con grep en Linux puedes buscar una palabra o patrón y se imprimirán la línea o líneas que la contengan. Una de las utilidades de grep es el uso de expresiones regulares.

Expresiones Regulares en Python

En la librería estándar de Python podemos encontrar el modulo re, el cual nos proporciona todas las operaciones necesarias para trabajar con las expresiones regulares, así que debemos importar el módulo re: import re.

Buscando Coincidencias:

Una vez que hemos importado el módulo, podemos empezar a tratar de buscar coincidencias con un determinado patrón de búsqueda. Para hacer esto, primero debemos compilar nuestra expresión regular en un objeto de patrones de Python, el cual posee métodos para diversas operaciones como los siguientes:

- match(): El cual determinada si la regex tiene coincidencias en el comienzo del texto.
- search(): El cual escanea todo el texto buscando cualquier ubicación donde haya una coincidencia.
- findall(): El cual encuentra todos los subtextos donde haya una coincidencia y nos devuelve estas coincidencias como una lista.
- finditer(): El cual es similar al anterior pero en lugar de devolvernos una lista nos devuelve un iterador.

Cunado hay coincidencias, Python nos devuelve un Objeto de coincidencia con sus propios métodos:

group(): El cual devuelve el texto que coincide con la expresión regular.

start(): El cual devuelve la posición inicial de la coincidencia.

end(): El cual devuelve la posición final de la coincidencia.

span(): El cual devuelve una tupla con la posición inicial y final de la coincidencia.

Modificando el texto de entrada:

Además de buscar coincidencias de nuestro patrón de búsqueda en un texto, podemos utilizar ese mismo patrón para realizar modificaciones al texto de entrada. Para estos casos podemos utilizar los siguientes métodos:

- split(): El cual divide el texto en una lista, realizando las divisiones del texto en cada lugar donde se cumple con la expresión regular.

- sub(): El cual encuentra todos los subtextos donde existe una coincidencia con la expresion regular y luego los reemplaza con un nuevo texto.
- subn(): El cual es similar al anterior pero además de devolver el nuevo texto, también devuelve el número de reemplazos que realizó.

Banderas de compilación

Las banderas de compilación permiten modificar algunos aspectos de cómo funcionan las expresiones regulares.

Algunas de las banderas de compilación que podemos encontrar son:

- IGNORECASE, I: Para realizar búsquedas sin tener en cuenta las minúsculas o mayúsculas.
- VERBOSE, X: Que habilita el modo verborrágico, el cual permite organizar el patrón de búsqueda de una forma que sea más sencilla de entender y leer.
- ASCII, A: Que hace que las secuencias de escape \w, \b, \s and \d funciones para coincidencias con los caracteres ASCII.
- DOTALL, S: La cual hace que el metacaracter. funcione para cualquier caracter, incluyendo el las líneas nuevas.
- LOCALE, L: Esta opción hace que \w, \W, \b, \B, \s, y \S dependientes de la localización actual.
- MULTILINE, M: Que habilita la coincidencia en múltiples líneas, afectando el funcionamiento de los metacaracteres ^ and $.

MATERIAL Y EQUIPO

Para la realización de esta práctica son necesarias las siguientes herramientas:

- Un equipo de cómputo que cumpla con los requerimientos para el uso del lenguaje de programación Python.
- Tener instalado el lenguaje de programación Python.
- Contar con un IDE para programar con Python, cualquiera es útil.
- Contar con una terminal de Linux, ya sea de internet o usando una distribución de Linux.
- Internet.

DESARROLLO

Primera parte.

En primer lugar, se realizaron cuatro ejercicios de expresiones regulares, los cuales están en la página: https://regex.sketchengine.co.uk/

El primer ejercicio consta de hacer que se seleccionen todas las palabras de la izquierda, pero ninguna de la derecha, la solución a este problema lo vemos en la figura 1;

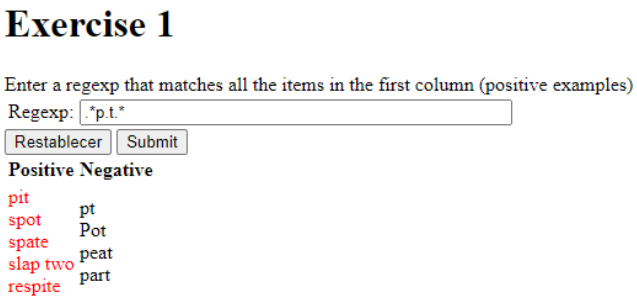


Figura 1. Ejercicio 1 de expresiones regulares.

En este primer ejercicio, personalmente, estaba algo confundido sobre como usar de manera practica las expresiones regulares y se me complico un poco, pero el profesor lo explico después y de esta forma los demás problemas fueron más sencillos.

El segundo ejercicio consta de lo mismo que el primero, la solución la tenemos en la figura 2;

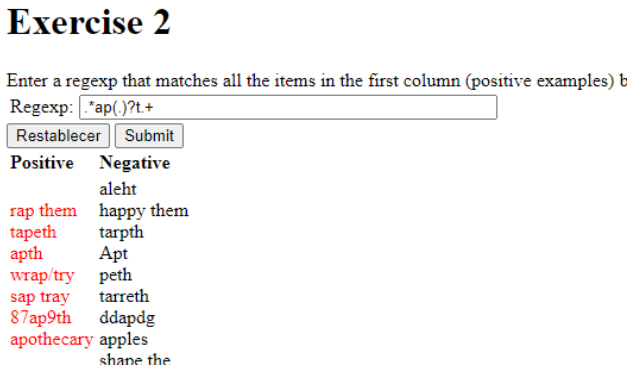


Figura 2. Ejercicio 2 de expresiones regulares.

En lo personal, este fue el mas complicado de los 4, pues no se me iluminaba la cadena ***apth*** pero, después de pensar un rato, me di cuenta que usando el operador ? para que un carácter sea opcional y así se ilumine.

El ejercicio 3, al igual que los dos anteriores, es de rellenar las cadenas de la izquierda, pero no las de la derecha, la solución la tenemos en la figura 3;

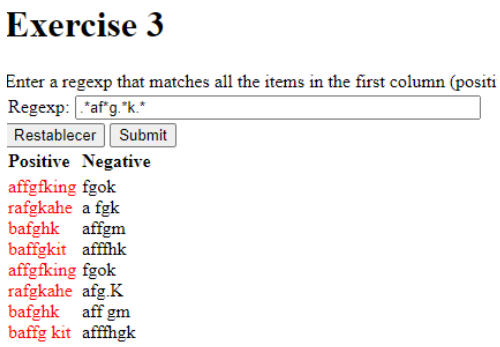


Figura 3. Ejercicio 3 de expresiones regulares.

El cuarto y ultimo ejercicio de esta parte de la practica consiste en indicar cuando una cadena termina y cuando empieza, la solución la podemos ver en la figura 4;

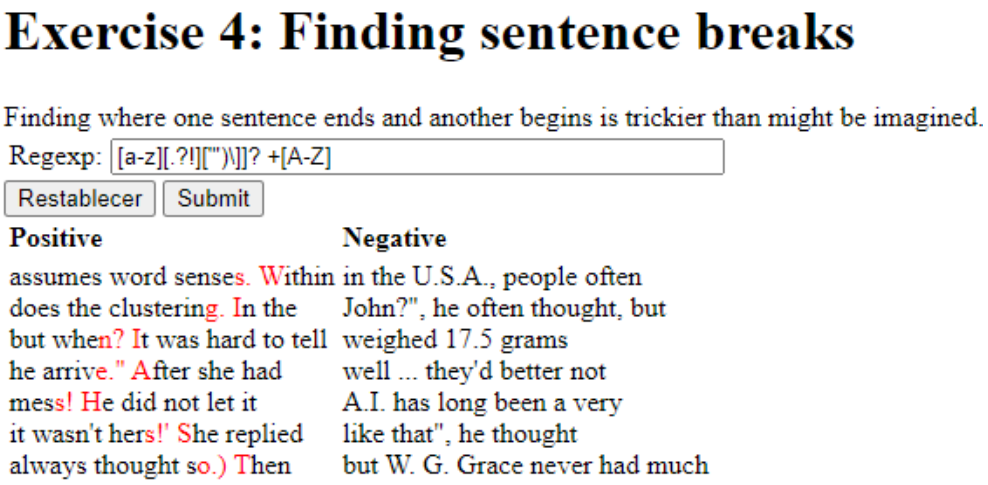


Figura 4. Ejercicio 4 de expresiones regulares.

Este problema parece difícil, pero si no analizamos nos damos cuenta que no lo es, pues solo es dar condiciones de cómo puede acabar o terminar una cadena.

Segunda parte.

La segunda parte de la práctica fue usar el comando grep en Linux, manipulando el himno del politécnico, el maestro nos mostro algunos ejemplos de como usar este comando, por ejemplo; seleccionar las líneas que empiecen con la subcadena Poli (ver figura 5);

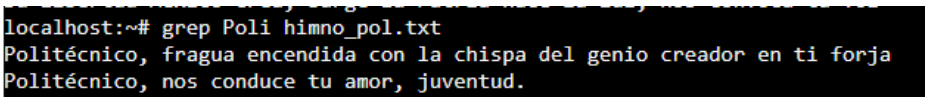


Figura 5. Ejemplo de grep.

Otro ejemplo puede ser elegir palabras que empiecen con c y terminen con a (ver figura 6)

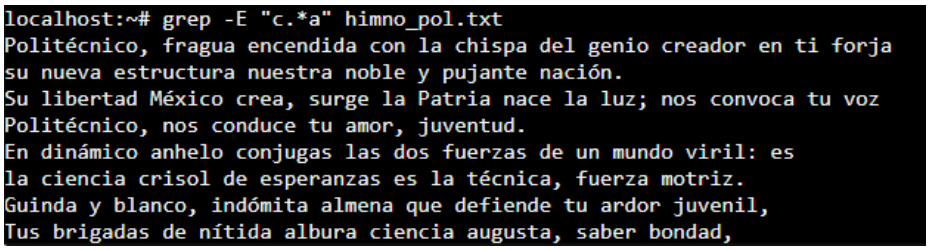


Figura 6. Ejemplo grep

En algunas terminales se seleccionan las palabras que cumplan con las condiciones, en este caso usamos uno de internet y no lo hace, por lo que es algo difícil visualizar los resultados.

El problema propuesto para esta parte de la práctica es Determinar una expresión regular para identificar todas las líneas en las que no aparezca ninguna palabra con dos vocales juntas en el Himno del IPN usando grep (ver figura 7.).

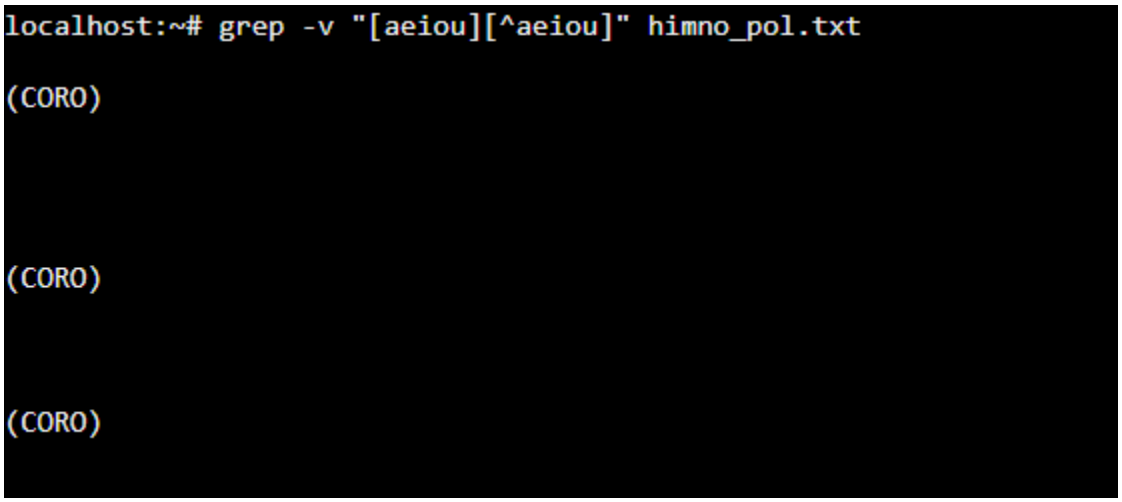


Figura 7. Ejercicio grep.

Esto lo logre con la expresión regular [aeiou][^aeiou] usando grep y -v; el -v funciona como una negación, aquí si me detuve bastante tiempo ya que no pude conseguir una expresión que cumpliera lo que se requería con -E, trate con varias formas y en todas siempre había palabras con dos vocales contiguas, por lo tanto, opte por usar -v con el que llegue al resultado.

Tercer y cuarta parte

En la tercera parte tenemos que seguir un tutorial de como usar las expresiones regulares en Python, parte que se vio en el marco teórico, para así poder hacer la cuarta parte que es determinar expresiones regulares para reconocer RFCs y direcciones IPV4, para verificar que funcionan en grep y Python.

Como parte del tutorial tenemos algunos ejemplos usando algunas de los métodos para buscar coincidencias (ver figura 8);

```python
import re


# compilando la regex
patron = re.compile(r'\bfoo\b')  # busca la palabra foo

texto = """ bar foo bar
foo barbarfoo
foofoo foo bar
"""

# match nos devuelve None porque no hubo coincidencia al comienzo del texto
print(patron.match(texto))

# match encuentra una coindencia en el comienzo del texto
m = patron.match('foo bar')
print(m)

# search nos devuelve la coincidencia en cualquier ubicacion.
s = patron.search(texto)
print(s)

# findall nos devuelve una lista con todas las coincidencias
fa = patron.findall(texto)
print(fa)
```

```
None
<re.Match object; span=(0, 3), match='foo'>
<re.Match object; span=(5, 8), match='foo'>
['foo', 'foo', 'foo']
```

Figura 8. Ejemplos de regex en Python.

En la figura 9 podemos ver algunos ejemplos de expresiones regulares, pero esta vez para modificar el texto de entrada:

```python
# texto de entrada
becquer = """Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Podrá la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
la llama de tu amor."""

# patron para dividir donde no encuentre un caracter alfanumerico
patron = re.compile(r'\W+')

palabras = patron.split(becquer)
palabras[:10]  # 10 primeras palabras

# Cambiando "Podrá" o "podra" por "Puede"
podra = re.compile(r'\b(P|p)odrá\b')
puede = podra.sub("Puede", becquer)
print(puede)
```

```
Puede nublarse el sol eternamente;
Puede secarse en un instante el mar;
Puede romperse el eje de la tierra
como un débil cristal.
¡todo sucederá! Puede la muerte
cubrirme con su fúnebre crespón;
Pero jamás en mí Puede apagarse
la llama de tu amor.
```

Figura 9. Ejemplos de regex con Python.

A continuación, en la figura 10, podemos ver ejemplos con banderas de compilación;

```
# Ejemplo de VERBOSE
mail = re.compile(r"""
\b              # comienzo de delimitador de palabra
[\w.%+-]        # usuario: Cualquier caracter alfanumerico mas los signos (.%+-)
+@              # seguido de @
[\w.-]          # dominio: Cualquier caracter alfanumerico mas los signos (.-)
+\.             # seguido de .
[a-zA-Z]{2,6}   # dominio de alto nivel: 2 a 6 letras en minúsculas o mayúsculas.
\b              # fin de delimitador de palabra
""", re.X)
```

```
mails = """raul.lopez@relopezbriega.com, Raul Lopez Briega,
foo bar, relopezbriega@relopezbriega.com.ar, raul@github.io,
https://relopezbriega.com.ar, https://relopezbriega.github.io,
python@python, river@riverplate.com.ar, pythonAR@python.pythonAR
"""
```

```
# filtrando los mails con estructura válida
mail.findall(mails)
```

```
['raul.lopez@relopezbriega.com',
 'relopezbriega@relopezbriega.com.ar',
 'raul@github.io',
 'river@riverplate.com.ar']
```

Figura 10. Ejemplos de regex en Python.

Ahora veremos un caso en concreto, que es para validar una URL, la figura 11 nos muestra el código correspondiente en Python y su salida;

```
# Validando una URL
url = re.compile(r"^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?

# vemos que https://relopezbriega.com.ar lo acepta como una url válida.
print(url.search("https://relopezbriega.com.ar"))

# pero https://google.com/un/archivo!.html no la acepta por el carcter !
print(url.search("https://google.com/un/archivo!.html"))
<re.Match object; span=(0, 28), match='https://relopezbriega.com.ar'>
None
```

Figura 11. Validando URL con regex en Python.

Ahora bien, pasaremos a la parte 4 y última de la práctica.

Expresión para reconocer RFCs

Primeramente, tenemos que saber como se forma un RFC, este se forma de la siguiente manera:

Morales: Se compone de 3 letras seguidas por 6 dígitos y 3 caracteres alfanuméricos=12
Físicas: consta de 4 letras seguida por 6 dígitos y 3 caracteres alfanuméricos =13

En la figura 12 vemos un ejemplo de RFC:

Figura 12. Ejemplo de RFC.

Sabiendo como es que se forma un RFC podemos generar una expresión regular para verificarla: **^([A-ZÑ&]{3,4})([0-9]{2}(0[0-9]|1[0-2])(0[1-9]|[12][0-9]|3[01]))([A-Z0-9]{3})**

Ahora expliquemos la expresión regular:

^([A-ZÑ&]{3,4}) Esta parte son los primeros 3 caracteres si se trata de una persona moral o 4 si es de una persona física.

([0-9]{2}(0[0-9]|1[0-2])(0[1-9]|[12][0-9]|3[01])) Esta parte se encarga de los 6 dígitos siguientes; **[0-9]{2}** genera el año, **(0[0-9]|1[0-2])** se encargar de verificar el mes **(0[1-9]|[12][0-9]|3[01])** este grupo verifica el día del mes, por último **([A-Z0-9]{3}** verifica los últimos 3 caracteres que es la homoclave, los cuales son caracteres alfanuméricos.

Una vez implementando nuestra expresión regular en Python, nos queda de la siguiente forma (ver figura 13)

```
exp=('^([A-ZÑ&]{3,4})([0-9]{2}(0[0-9]|1[0-2])(0[1-9]|[12][0-9]|3[01]))'
     '([A-Z0-9]{3})')

rfc= re.compile(exp)

#El RFC MELM8305281H0 es valido
print(rfc.search("MELM8305281H0"))

#Pero el RFC MELM0583281H0
print(rfc.search("MELM0583281H0"))
<re.Match object; span=(0, 13), match='MELM8305281H0'>
None
```

Figura 13. Validando un RFC.

En la figura 113 observamos la salida del código que es el texto en azul, devolviéndonos un match ya que el primer RFC es válido, posteriormente nos da un None, debido al RFC que no es válido.

Para probarlo con grep se creo un archivo de texto que contiene las mismas RFCs que se probaron en el programa en Python, que son:

MELM8305281H0

MELM0583281H0

Posteriormente se uso la expresión regular con el comando grep, devolviendo el RFC correcto, esto lo vemos en la figura 14;

```
localhost:~# grep -E "^([A-ZÑ&]{3,4})([0-9]{2}(0[0-9]|1[0-2])(0[1-9]|[12][0-9]|3
[01]))([A-Z0-9]{3})" rfc.txt
MELM8305281H0
```

Figura 14. Verificando RFCs con grep.

Expresión para reconocer direcciones IPV4:

Primero, tengamos en cuenta que una dirección IPV4 se obtiene de 4 octetos que van de 0.0.0.0 hasta 255.255.255.255, de esta forma tenemos la siguiente expresión regular: ***/^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]) $/***

La primera parte de nuestra expresión es: **^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]) \.){3}** esta parte hace que lo que está dentro del grupo pueda ser un octeto que vaya del 0 al 9, del 10 al 99, del 100 al 199 o del 200 al 255, la parte **\.** indicará que cada octeto termine con un punto y el **{3}** indica que se hagan 3 octetos, el cuarto octeto se separa ya que este no contiene un punto al final, la parte del cuarto octeto es indicada con la parte **([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]) $**

Una vez implementando nuestra expresión regular en Python, nos queda de la siguiente forma (ver figura 15):

```
import re

expresion=('^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}'
                '([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$')
ipv4= re.compile(expresion)

#la ip 152.323.32.1442 no es valida

print(ipv4.search("152.323.32.1442"))

#pero la ip 172.168.124.136 si es valida
print(ipv4.search("172.168.124.136"))

None
<re.Match object; span=(0, 15), match='172.168.124.136'>
```

Figura 15. Validando una dirección IPV4

Como vimos en la figura 15, la parte en azul es la salida, en donde nos devuelve un None ya que la primera dirección IPV4 no es válida, pero la segunda si, por lo que devuelve un match.

Para probarlo con grep se creo un archivo de texto que contiene las mismas direcciones que se probaron en el programa en Python, que son:

152.323.32.1442

172.168.124.136

Posteriormente se uso la expresión regular con el comando grep, devolviendo la dirección correcta, esto lo vemos en la figura 16;

```
localhost:~# grep -E "^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([
0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$" ip.txt
172.168.124.136
```

Figura 16. Verificando direcciones IPV4 con grep.

CONCLUSIONES Y RECOMENDACIONES

A partir de los ejercicios realizados en esta práctica, comprendí de una mejor manera como es que funcionan las expresiones regulares, como formarlas y algunas aplicaciones, en este caso validar un rfc o una dirección IPV4.

La principal problemática que se me presento, fue que operadores usar al momento de crear mi expresión regular, pues hay bastantes, pero al hacer la investigación a fondo y ver como es que funciona cada operador en las expresiones regulares, este proceso se me facilito.

Algo que, en mi opinión, hubiera recomendado y recomendaría para practicas posteriores, es que el profesor nos muestre mas ejemplos sobre como usar lo que se verá en la práctica, por ejemplo, en esta práctica sentí que vimos pocos ejemplos y nos falto ver como es que funcionan muchos de los operadores que podemos usar en las expresiones regulares, aun así, el desarrollo de esta fue buena.

BIBLIOGRAFÍA

- Santo Orcero, D. (2003). Perl: expresiones regulares. Mundo Linux: Sólo programadores Linux, (56), 48-52.
- Sguerra, M. D. (2006). Las expresiones regulares. INVENTUM, 1(1), 31-37.
- Gustavo, B. (2020). “Comando grep + ejemplos”. Recuperado de: https://www.hostinger.co/tutoriales/comando-grep-linux/#:~:text=El%20comando%20grep%20perteneciente%20a,o%20l%C3%ADneas%20que%20la%20contengan.
- Raul E. Lopez Briega. (2015). “Expresiones regulares con Python”. Recuperado de: https://relopezbriega.github.io/blog/2015/07/19/expresiones-regulares-con-python/