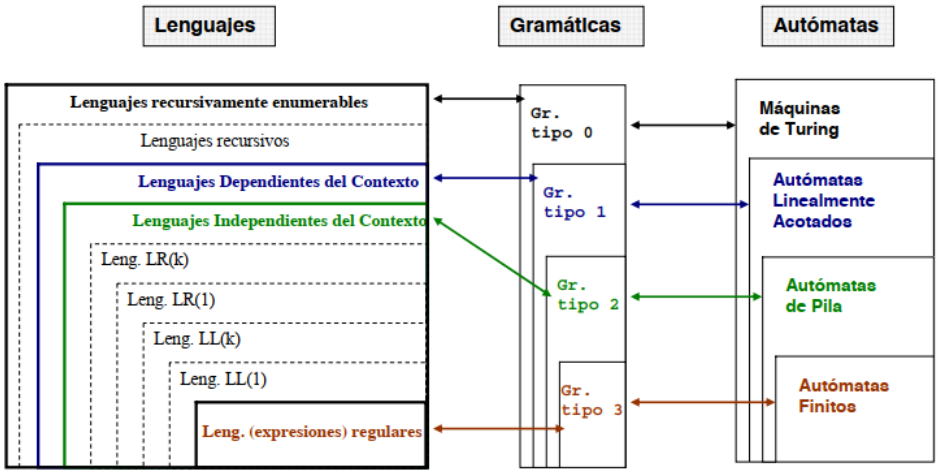
😎 Expresiones Regulares en Python

Introducción

> **Las expresiones regulares, también llamadas regex, son un conjunto de caracteres que forman un patrón de búsqueda, y que están normalizados por medio de una sintaxis específica. Se usan normalmente en aplicaciones que implican procesamiento de una gran cantidad de texto.**

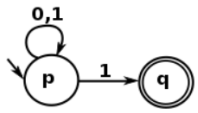
Chomsky clasificó las gramáticas formales (y los lenguajes que éstas generan) de acuerdo a una jerarquía de 4 niveles. A cada nivel de gramática se le puede asociar de forma natural un conjunto de lenguajes que serán los que esas gramáticas generan, pero además, se le puede asociar una clase de *autómatas* formada por aquellos que podrán reconocer a dichos lenguajes.



Cada nivel de lenguaje se corresponde con un tipo de autómata. Por ejemplo, dado un lenguaje de tipo 3 siempre será posible encontrar un autómata finito que reconozca dicho lenguaje, es decir, que permita determinar si una determinada palabra pertenece o no al lenguaje. Si el lenguaje es de tipo 2 será necesario utilizar un autómata más complejo, concretamente un autómata de pila.

Y cada nivel contiene al anterior. Por ejemplo, cualquier lenguaje de tipo 3 es a su vez un lenguaje de tipo 2, es decir ($L3 \subsetneq L2 \subsetneq L1 \subsetneq L0$). De la misma forma, un autómata finito puede considerarse como un caso particular de autómata de pila y éste como un caso particular de máquina de Turing.

Las expresiones regulares expresan un lenguaje definido por una gramática regular que se puede resolver con un autómata finito no determinista (AFND), donde la coincidencia está representada por los estados. Ejemplo simple:



El lenguaje de Expresión Regular es una representación textual de tal autómata. Ese último ejemplo es expresado por la siguiente regex: **^[01]*1$** Que coincide con cualquier cadena que comience con 0 ó 1, repitiendo 0 o más veces, y que termine con 1. En otras palabras, es una expresión regular para hacer coincidir los números impares en su representación binaria.

Gramáticas

Las gramáticas formales definen un lenguaje describiendo cómo se pueden generar las cadenas del lenguaje. Una gramática formal es una cuádrupla $G = (N, T, P, S)$ donde:

- N es un conjunto finito de símbolos no terminales
- T es un conjunto finito de símbolos terminales $N \cap T = \varnothing$
- P es un conjunto finito de producciones
- S es el símbolo distinguido o axioma $S \notin (N \cup T)$

Gramáticas regulares (Tipo 3)

Generan los lenguajes regulares (aquellos reconocidos por un autómata finito). Son las gramáticas más restrictivas. El lado derecho de una producción debe contener un símbolo terminal y, como máximo, un símbolo no terminal. Estas gramáticas pueden ser:

- Lineales a derecha
- Lineales a izquierda

Expresiones regulares

Se denominan expresiones regulares sobre un alfabeto A, a las expresiones que se pueden construir a partir de las siguientes reglas:

- $\varnothing$ es una expresión regular que describe el lenguaje vacío.
- ε es una expresión regular que describe el lenguaje $\{\varepsilon\}$, esto es el lenguaje que contiene únicamente la cadena vacía.
- Para cada símbolo $a \in A$, a es una expresión regular que describe el lenguaje $\{a\}$, esto es el lenguaje que contiene únicamente la cadena a.
- Si r y s son expresiones regulares que describen los lenguajes $L(r)$ y $L(s)$ respectivamente:

1. unión: $r + s$ es una expresión regular que describe el lenguaje $L(r) \cup L(s)$
2. concatenación: $r \ . \ s$ es una expresión regular que describe el lenguaje $L(r).\ L(s)$
3. clausura de Kleene: $r*$ es una expresión regular que describe el lenguaje $L(r)*$

El operador de clausura es el que tiene mayor precedencia, seguido por el operador de concatenación y por último el operador de unión. Las expresiones regulares describen los lenguajes regulares (aquellos reconocidos por autómatas finitos).

Componentes de la sintaxis en Python:

Más información (https://docs.python.org/es/3/library/re.html)

- **Literales: Cualquier caracter se encuentra a sí mismo, a menos que se trate de un metacaracter con significado especial. Una serie de caracteres encuentra esa misma serie en el texto de entrada.**
- **Secuencias de escape: La sintaxis permite utilizar las secuencias de escape que conocemos de otros lenguajes de programación.**

Secuencias de escape en python (https://docs.python.org/es/3/reference/lexical_analysis.html#strings)

Clases de caracteres:	Metacaracteres:
Se pueden especificar encerrando una lista entre corchetes [ ] y encontrará uno cualquiera. Si el primer símbolo después del "[" es "^", encuentra cualquier caracter que no está en la lista.	Son caracteres especiales, sumamente importantes para entender la sintaxis de las expresiones regulares y existen diferentes tipos

Metacaracteres - delimitadores:

Permiten delimitar dónde queremos buscar los patrones de búsqueda. Algunos de ellos son:

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

```
Ejercicio:

Usando regex101.com, encontrar el fin de línea en el siguiente texto:

Las expresiones regulares son una herramienta poderosa en Python para buscar patrones y tokenizar cadenas
de texto.
Al utilizar el módulo re de Python en combinación con una expresión regular adecuada, podemos dividir una
cadena de texto en tokens para su posterior procesamiento en una variedad de contextos.
```

Metacaracteres - clases

Son clases predefinidas que facilitan la utilización de las expresiones regulares. Algunos de ellos son:

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

```
Ejercicio:

Usando regex101.com, encontrar caracteres numéricos en el siguiente texto:

2563 Sea uno de los primeros solicitantes
2564 Sea uno de los primeros solicitantes
2565 Sea uno de los primeros solicitantes
2566 Sea uno de los primeros solicitantes
Nombre: solicitantes, Longitud: 2567, tipo d: objeto
```

Metacaracteres - iteradores

Cualquier elemento de una expresion regular puede ser seguido por otro tipo de metacaracteres, los iteradores.

Usando estos metacaracteres se puede especificar el número de ocurrencias del caracter previo, de un metacaracter o de una subexpresión. Algunos de ellos son:

Metacaracter	Descripción
*	cero o más, similar a {0,}. indica “cero o más coincidencias” del carácter que viene immediatamente antes.
+	una o más, similar a {1,}. indica “por lo menos una coincidencia”.
?	cero o una, similar a {0,1} indica “como máximo una coincidencia” del carácter que viene inmediatamente antes.
{n}	indica “exactamente n coincidencias” del carácter anterior.
{n,}	por lo menos n veces.
{n,m}	por lo menos n pero no más de m veces. indica “entre n y m coincidencias”.
*?	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.
()	sirven para agrupar los términos y especificar el orden de las operaciones.

Ejercicio:

Usando regex101.com, encontrar una o más 'a' en el siguiente texto:

Las expresiones regulares son patrones utilizados para encontrar y manipular cadenas de texto de manera efectiva y precisa. En informática, son una herramienta fundamental en la búsqueda de patrones en datos y textos. En Python, las expresiones regulares son muy populares debido a su poder y flexibilidad en el análisis de datos.

Más ejemplos:

In [1]:
```
import re

'''
Metacaracter * : ninguna o más veces ese carácter a su izquierda
'''
patrones = ['ho*la']
texto = 'hi hla hola hoola hooola hooooola huuuuulaaaa'

def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))

buscar (patrones, texto)
```

```
['hla', 'hola', 'hoola', 'hooola', 'hooooola']
```

In [2]:
```
'''
Metacaracter +: una o más repeticiones de la letra a su izquierda
'''
patrones = ['ho+']
buscar(patrones, texto)
```

```
['ho', 'hoo', 'hooo', 'hoooooo']
```

In [3]:
```
'''
Metacaracter ?: una o ninguna repetición de la letra a su izquierda
'''
patrones = ['ho?','ho?la']
buscar(patrones, texto)
```

```
['h', 'h', 'ho', 'ho', 'ho', 'ho', 'h']
['hla', 'hola']
```

Repeticiones y rangos

In [4]:
```
'''
Número de repeticiones explicito de la letra a su izquierda: {n}
'''
patrones = ['ho{0}','ho{1}la','ho{3}la']
texto = 'hi hla hola hoola hooola hooooola huuuuulaaaa'

def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))

buscar(patrones, texto)
```

```
['h', 'h', 'h', 'h', 'h', 'h', 'h']
['hola']
['hooola']
```

In [5]:
```
'''
Sintaxis con rango {n, m}
'''
patrones = ['ho{0,1}la', 'ho{1,2}la', 'ho{2,10}la']
buscar(patrones, texto)
```

```
['hla', 'hola']
['hola', 'hoola']
['hoola', 'hooola', 'hoooooola']
```

Conjuntos de caracteres, repeticiones y exclusión

In [6]:
```
'''
Conjuntos de caracteres [ ] - sets
'''
texto = 'hele hala hela hila heli hola hula'
patrones = ['h[ou]la', 'h[aio]la', 'h[aeiou]la']
buscar(patrones, texto)
```

```
['hola', 'hula']
['hala', 'hila', 'hola']
['hala', 'hela', 'hila', 'hola', 'hula']
```

In [7]:
```
'''
Utilizar repeticiones
'''
texto = 'heeele haala heeela hiiiila hoooooola hiiilooo'
patrones = ['h[ae]la', 'h[ae]*la', 'h[io]{3,9}la']
buscar(patrones, texto)
```

```
[]
['haala', 'heeela']
['hiiiila', 'hoooooola']
```

In [8]:
```
'''
Operador de exclusión [^ ] para indicar una búsqueda contraria
'''
texto = 'hala hela hila hilo hola hula'
patrones = ['h[o]la', 'h[^o]la']
buscar(patrones, texto)
```

```
['hola']
['hala', 'hela', 'hila', 'hula']
```

Rangos y códigos escapados

In [9]:
```
'''
Códigos escapados:
\d numérico, \D no numérico
\s espacio en blanco, \S no espacio en blanco
\w alfanumérico, \W no alfanumérico
'''
texto = 'Matemática III - 1er. cuatrimestre - 2022'
patrones = [r'\d+', r'\D+', r'\s', r'\S+', r'\w+', r'\W+']
buscar(patrones, texto)
```

```
['1', '2022']
['Matemática III - ', 'er. cuatrimestre - ']
[' ', ' ', ' ', ' ', ' ', ' ']
['Matemática', 'III', '-', '1er.', 'cuatrimestre', '-', '2022']
['Matemática', 'III', '1er', 'cuatrimestre', '2022']
[' ', ' - ', '. ', ' - ']
```

Métodos que usan expresiones regulares

Existen varios métodos que aceptan expresiones regulares para examinar el contenido de cada elemento de cadena y siguen algunas de las convenciones de la API del módulo **"re"** integrado de Python.

Método	Descripción
match()	Call re.match() on each element, returning a Boolean.
extract()	Call re.match() on each element, returning matched groups as strings.
findall()	Call re.findall() on each element.
replace()	Replace occurrences of pattern with some other string.
contains()	Call re.search() on each element, returning a Boolean.
count()	Count occurrences of pattern.
split()	Equivalent to str.split(), but accepts regexps.
rsplit()	Equivalent to str.rsplit(), but accepts regexps.

Ejemplo:

In [10]:

```
import pandas as pd

famosos = pd.Series(['Jack Nicholson', 'Dustin Hoffman', 'Tom Hanks','Johnny Depp',
                     'Anthony Hopkins','Richard Gere'])

print(f"extract():\n{famosos.str.extract('([A-Za-z]+)')}\n\nfindall():\n{famosos.str.findall(r'^[^AEIOU].*[^aeiou]$')}")
```

```
extract():
         0
0     Jack
1   Dustin
2      Tom
3   Johnny
4  Anthony
5  Richard

findall():
0    [Jack Nicholson]
1    [Dustin Hoffman]
2         [Tom Hanks]
3       [Johnny Depp]
4                  []
5                  []
dtype: object
```

search() escanea todo el texto buscando cualquier ubicación donde haya una coincidencia. Busca el patrón dentro del texto y escribe la posición inicial y final de la ocurrencia.

In [11]:

```
texto = "Las expresiones regulares casi son un lenguaje de \
programación en miniatura para buscar y analizar cadenas. De hecho, \
se han escrito libros enteros sobre las expresiones regulares"

a_buscar = "casi"

x = re.search(a_buscar, texto)

'''
Escribe la posicion inicial y final de la ocurrencia.
'''
print(x.span())
```

```
(26, 30)
```

In [12]:

```
'''
Devuelve el texto que coincide con la expresion regular.
'''
print(x.group())
```

```
casi
```

In [13]:

```
'''
devuelve la posición inicial de la coincidencia.
'''
print(x.start())
```

```
26
```

In [14]:

```
'''
Devuelve la posición final de la coincidencia.
'''
print(x.end())
```

```
30
```

match() determina si la regex tiene coincidencias en el comienzo del texto.

In [15]:

```
texto = "Las expresiones regulares casi son un lenguaje de \
programación en miniatura para buscar y analizar cadenas. De hecho, \
se han escrito libros enteros sobre las expresiones regulares"

a_buscar1= "Las"
'''
Busca el patron dentro del texto
'''
x = re.match(a_buscar1, texto)
print(x.span())
```

```
(0, 3)
```

El siguiente match() dará un error al intentar hacer el print() pues el patrón “expresiones” no se encuentra al principio del texto y por tanto el método match() devuelve Error

In [16]:
```
a_buscar2 = "expresiones"
y = re.match(a_buscar2, texto)
'''
Error!
'''
print(y.span())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [16], in <cell line: 6>()
      2 y = re.match(a_buscar2, texto)
      3 '''
      4 Error!
      5 '''
----> 6 print(y.span())

AttributeError: 'NoneType' object has no attribute 'span'
```

findall() encuentra todos los subtextos donde haya una coincidencia y devuelve estas coincidencias como una lista.

Si agregamos len(re.findall(a_buscar, texto)), podríamos saber cuántas veces se repite un patrón dentro de una cadena.

In [17]:
```
texto = """El poder de las expresiones regulares se manifiesta cuando agregamos caracteres
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestra
expresión regular nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código."""

a_buscar = "la"
x = re.findall(a_buscar, texto)
print(x), len(x)
```

```
['la', 'la', 'la', 'la', 'la', 'la']
```

Out[17]:
```
(None, 6)
```

finditer() similar a findall pero en lugar de devolver una lista devuelve un iterador.

In [18]:
```
len(re.findall(a_buscar, texto))
```

Out[18]:
```
6
```

Tanto match() como search() sólo se quedan con la primera ocurrencia encontrada.

Se puede utilizar la función finditer() para buscarlas todas, y se devuelve las posiciones de las ocurrencias. También se puede recorrer una a una con next()

In [19]:
```
a_buscar = "os"

x = re.finditer(a_buscar, texto)

for i in x:
    print(i.span())
```

```
(66, 68)
(120, 122)
(210, 212)
(264, 266)
(311, 313)
```

Modificar el texto de entrada

Además de buscar coincidencias podemos utilizar ese mismo patrón para realizar modificaciones al texto de entrada.

In [20]:
```
texto = """Las expresiones regulares casi son un lenguaje de
programación para buscar
y analizar cadenas."""

'''
patron para dividir dónde no encuentre un caracter alfanumérico
'''
patron = re.compile(r'\W+')
palabras = patron.split(texto)
'''
5 primeras palabras
'''
palabras[:5]
```

Out[20]:
```
['Las', 'expresiones', 'regulares', 'casi', 'son']
```

In [21]:
```
'''
Utilizando la versión no compilada de split. Dividiendo por línea.
'''
re.split(r'\n', texto)
```

Out[21]:
```
['Las expresiones regulares casi son un lenguaje de',
 'programación para buscar ',
 'y analizar cadenas.']
```

In [22]:
```
texto = """El poder de las expresiones regulares se manifiesta cuando agregamos caracteres
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones regulares nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código."""

reg = re.compile(r'\b(R|r)[a-z]+\b')

regex = reg.sub("REGEX", texto)
print(regex)
```

```
El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres
especiales a la cadena de búsqueda que nos permite controlar de manera más precisa
qué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras
expresiones REGEX nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código.
```

In [23]:
```
regex = reg.sub("REGEX", texto, 1)

re.subn(r'\b(R|r)egulares\b', "REGEX", texto)
```

Out[23]:
```
('El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres\nespeciales a la cadena de búsqueda que nos per
mite controlar de manera más precisa\nqué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestras\nexp
resiones REGEX nos permitirá buscar coincidencias y extraer datos usando unas\npocas líneas de código.',
 2)
```

Banderas de compilación

Las banderas de compilación permiten modificar algunos aspectos de cómo funcionan las expresiones regulares. Todas ellas están disponibles en el módulo re con un nombre largo y una letra que lo identifica.

Ejemplos (https://pynative.com/python-regex-flags/)

Validar mails

Una regex aproximada puede ser: \b[\w.%+-]+@[\w.-]+.[a-zA-Z]{2,6}\b

In [25]:
```
'''
VERBOSE
'''

mails = """aaa.bbbbbb@gmail.com, Pepe Pepitito,
ccc.dddddd@yahoo.com.ar, qué lindo día , eeeee@github.io,
https://pypi.org/project/regex/, https://ffffff.github.io,
python@python, river@riverplate.com.ar, pythonAR@python.pythonAR
"""

mail = re.compile(r"""
\b # comienzo de delimitador de palabra
[\w.%+-] # 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)

mail.findall(mails)
```

Out[25]:
```
['aaa.bbbbbb@gmail.com',
 'ccc.dddddd@yahoo.com.ar',
 'eeeee@github.io',
 'river@riverplate.com.ar']
```

Validar una URL

Una regex aproximada puede ser: ^(https?://)?([\da-z.-]+).([a-z.]{2,6})([/\w .-]*/?$

In [26]:
```
url = re.compile(r"^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$")
print(url.search("https://www.python.org/"))
```

```
<re.Match object; span=(0, 23), match='https://www.python.org/'>
```

In [27]:
```
print(url.search("https://www.google.com/!.html"))
```

```
None
```

Validar una dirección IP

Una regex aproximada puede ser: ^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?).){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0- 9][0-9]?)$

In [28]:
```
patron = (r'^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$')

ip = re.compile(patron)
ip.search("98.61.125.138")
```

Out[28]:
```
<re.Match object; span=(0, 13), match='98.61.125.138'>
```

In [29]:
```
print(ip.search("256.60.124.136"))
```

```
None
```

? combinados con otros caracteres realiza otras operaciones.

?: significa iniciar un grupo sin captura. Los grupos regulares que no capturan permiten que el motor vuelva a ingresar al grupo e intente hacer coincidir algo diferente (como una alternativa diferente, o hacer coincidir menos caracteres cuando se usa un cuantificador). Un grupo puede ser excluido de la enumeración al agregar ?: en el inicio. Eso se usa cuando necesitamos aplicar un cuantificador a todo el grupo, pero no lo queremos como un elemento separado en el array de resultados.

Según la documentación en: https://docs.python.org/es/3/howto/regex.html
[A veces querrá usar un grupo para denotar una parte de una expresión regular, pero no está interesado en recuperar el contenido del grupo. Puede hacer que este hecho sea explícito utilizando un grupo de no captura:
(?:...), donde puede reemplazar el ... con cualquier otra expresión regular.]...[es particularmente útil cuando se modifica un patrón existente, ya que puede agregar nuevos grupos sin cambiar cómo se numeran todos los demás grupos.]

Validar una fecha

Una regex aproximada puede ser: ^(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)$

In [30]:
```
fecha = re.compile(r'^(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/((19|20)\d\d)$')
print(fecha.search("2/10/1990"))
```

```
<re.Match object; span=(0, 9), match='2/10/1990'>
```

In [31]:
```
print(fecha.search("2-10-1990"))
```

```
None
```

In [32]:
```
print(fecha.search("32/12/2021"))
```

```
None
```

In [33]:
```
print(fecha.search("30/13/2020"))
```

```
None
```

El siguiente ejemplo utiliza expresiones regulares, busca códigos postales y los agrupa según la ciudad a la que pertenezcan. Ejecutar el código.

In [ ]:
```
with open("archs/codigos_postales.txt", encoding="utf-8") as f_codigos_postales:
    codigos = {}
    for linea in f_codigos_postales:
        res1 = re.search(r"[\d ]+([^\d]+[a-z])\s(\d+)", linea)
        if res1:
            ciudad, cp = res1.groups()
            if ciudad in codigos:
                codigos[ciudad].add(cp)
            else:
                codigos[ciudad] = {cp}

with open("archs/ciudades.txt", encoding="utf-8") as f_ciudades:
    for linea in f_ciudades:
        res2 = re.search(r"^[0-9]{1,2}\.\s+([\w\s-]+\w)\s+[0-9]", linea)
        ciudad = res2.group(1)
        print(ciudad, codigos[ciudad])
        print('\n')
```

Más información: (https://www.regular-expressions.info/)