PYTHON
EXPRESIONES REGULARES
CON RE

Las expresiones regulares, también llamadas **regex**, son un conjunto de caracteres que forman un patrón de búsqueda, y que están normalizadas por medio de una sintaxis específica. Los patrones se interpretan como un conjunto de instrucciones, que 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 reglas de reconocimiento de texto.

Componentes de su sintaxis:

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.

Secuencia de escape	Significado
\n	Nueva línea . El cursor pasa a la primera posición de la línea siguiente.
\t	Tabulador. El cursor pasa a la siguiente posición de tabulación.
\\	Barra diagonal inversa
\v	Tabulación vertical.
\ooo	Carácter ASCII en notación octal.
\xhh	Carácter ASCII en notación hexadecimal.
\xhhhh	Carácter Unicode en notación hexadecimal.

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.

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.

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.

Metacaracteres

```
# ejemplo  1
import re

texto = 'hi hla hola hoola hooola hooooola huuuuulaaaa'

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

# Metacaracter * : ninguna o mas veces ese carácter a su izquierda
patrones = ['ho*la']
buscar (patrones, texto)

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

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

```
>>> patrones = ['ho*la']
>>> buscar (patrones, texto)
['hla', 'hola', 'hoola', 'hooola', 'hooooola']
>>> patrones = ['ho+']
>>> buscar(patrones, texto)
['ho', 'hoo', 'hooo', 'hooooo']
>>> patrones = ['ho?','ho?la']
>>> buscar(patrones, texto)
['h', 'h', 'ho', 'ho', 'ho', 'ho', 'h']
['hla', 'hola']
```

Repeticiones y rangos

```
# ejemplo  2
import re

texto = 'hi hla hola hoola hooola hooooola huuuuulaaaa'

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

 # Número de repeticiones explicito de la letra a su izquierda: {n}
patrones = ['ho{0}','ho{1}la','ho{3}la']
buscar(patrones, texto)

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

```
>>> patrones = ['ho{0}','ho{1}la','ho{3}la']
>>> buscar(patrones, texto)
['h', 'h', 'h', 'h', 'h', 'h', 'h']
['hola']
['hooola']
>>> patrones = ['ho{0,1}la', 'ho{1,2}la', 'ho{2,10}la']
>>> buscar(patrones, texto)
['hla', 'hola']
['hola', 'hoola']
['hoola', 'hooola', 'hooooola']
>>>
```

Conjuntos de caracteres, repeticiones y exclusión

```
# ejemplo  3
 import re

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

# 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)

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

# 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)
```

```
>>> 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']
>>> texto = 'heeele haala heeela hiiiila hooooola hiiilooo'
>>> patrones = ['h[ae]la', 'h[ae]*la', 'h[io]{3,9}la']
>>> buscar(patrones, texto)
[]
['haala', 'heeela']
['hiiiila', 'hooooola']
>>> 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

```
# ejemplo  4
import re
def buscar(patrones, texto):
    for patron in patrones:
        print(re.findall(patron, texto))
# [A-Z]: Cualquier carácter alfabético en mayúscula (no especial ni número)
# [a-z]: Cualquier carácter alfabético en minúscula (no especial ni número)
# [A-Za- z]: Cualquier carácter alfabético en minúscula o mayúscula (no especial ni número)
# [A-z]: Cualquier carácter alfabético en minúscula o mayúscula (no especial ni número)
# [0-9]: Cualquier carácter numérico (no especial ni alfabético)
# [a-zA-Z0-9]: Cualquier carácter alfanumérico (no especial)
texto = 'hola h0la Hola mola m0la M0la'
patrones = ['h[a-z]la', 'h[0-9]la', '[A-z]{4}', '[A-Z][A-z0-9]{3}']
buscar(patrones, texto)
# 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 - 2020'
patrones = [r'\d+', r'\D+', r'\s', r'\S+', r'\w+',
                                          r'\W+']
buscar(patrones, texto)
```

```
>>> texto = 'hola h0la Hola mola m0la M0la'
>>> patrones = ['h[a-z]la', 'h[0-9]la', '[A-z]{4}', '[A-Z][A-z0-9]{3}']
>>> buscar(patrones, texto)
['hola']
['h0la']
['hola', 'Hola', 'mola']
['Hola', 'M0la']
>>> texto = 'Matemática III - 1er. cuatrimestre - 2020'
>>> patrones = [r'\d+', r'\D+', r'\s', r'\S+', r'\w+', r'\W+']
>>> buscar(patrones, texto)
['1', '2020']
['Matemática III - ', 'er. cuatrimestre - ']
[' ', ' ', ' ', ' ', ' ', ' ']
['Matemática', 'III', '-', '1er.', 'cuatrimestre', '-', '2020']
['Matemática', 'III', '1er', 'cuatrimestre', '2020']
[' ', ' - ', '. ', ' - ']
```

Método search()
escanea todo el texto buscando cualquier ubicación donde haya una coincidencia.

```
#ejemplo 5
import re
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)

print(x.span())
print(x.group())
print(x.start())
print(x.end())
```

```
>>> a_buscar = "casi"
>>> x = re.search(a_buscar, texto)
>>> print(x.span()) #Escribe la posicion inicial y final de la ocurrencia
(26, 30)
>>> print(x.group())  # devuelve el texto que coincide con la expresion regular.
casi
>>> print(x.start()) # devuelve la posición inicial de la coincidencia.
26
>>> print(x.end())  # devuelve la posición final de la coincidencia.
30
```

Para buscar un patrón en un string podemos utilizar el método **search**() que busca el patrón dentro del texto y escribe la posición inicial y final de la ocurrencia

Método match():
Determina si la regex tiene coincidencias en el comienzo del texto.

```
# ejemplo 6
import re

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"
a_buscar2 = "expresiones"

x = re.match(a_buscar1, texto)
print(x.span())

y = re.match(a_buscar2, texto)
print(y.span()) #Error!
```

```
>>> a_buscar1= "Las"
>>> a_buscar2 = "expresiones"
>>> x = re.match(a_buscar1, texto)
>>> print(x.span())
(0, 3)
>>> y = re.match(a_buscar2, texto)
>>> print(y.span())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
```

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

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

```
#ejemplo  7
import re

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 = "est"

x = re.findall(a_buscar, texto)

print(x)
```

```
>>> a_buscar = "est"
>>> x = re.findall(a_buscar, texto)
>>> print(x)
['est', 'est', 'est']
```

Busca todas las coincidencias en una cadena, si agregamos len(**re.findall**(a_buscar, texto)), podríamos saber cuantas veces se repite un patrón dentro de una cadena

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

```
# ejemplo 8
import re

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 = "est"

x = re.finditer(a_buscar, texto)

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

```
>>> a_buscar = "est"
>>> x = re.finditer(a_buscar, texto)
>>>
>>> for i in x:
...     print(i.span())
...
(47, 50)
(207, 210)
(239, 242)
```

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()

Modificando el texto de entrada

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

```
# ejemplo  9
import re
texto = """Las expresiones regulares casi son un lenguaje de
   programación para buscar y analizar cadenas."""“
# patron para dividir donde no encuentre un caracter alfanumerico
patron = re.compile(r'\W+')
palabras = patron.split(texto)
palabras[:8]  # 8 primeras palabras
# Utilizando la version no compilada de split.
re.split(r'\n', texto)  # Dividiendo por linea.
# Utilizando el tope de divisiones
patron.split(texto, 3)
```

```
>>> patron = re.compile(r'\W+')
>>> palabras = patron.split(texto)
>>>
>>> palabras[:8]  # 8 primeras palabras
['Las', 'expresiones', 'regulares', 'casi', 'son', 'un', 'lenguaje', 'de']
>>> # Utilizando la version no compilada de split.
>>>
>>> re.split(r'\n', texto)  # Dividiendo por linea.
['Las expresiones regulares casi son un lenguaje de ', '    programación para buscar y analizar cadenas.']
>>> # Utilizando el tope de divisiones
>>>
>>> patron.split(texto, 3)
['Las', 'expresiones', 'regulares', 'casi son un lenguaje de \n    programación para buscar y analizar cadenas.']
>>>
```

```
# ejemplo 10
import re

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)egulares\b')
regex = reg.sub("REGEX", texto)
print(regex)

regex = reg.sub("REGEX", texto, 1)
print(regex)

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

```
>>> reg = re.compile(r'\b(R|r)egulares\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.
```

```
>>> regex = reg.sub("REGEX", texto, 1)
>>> 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 regulares nos permitirá buscar coincidencias y extraer datos usando unas
pocas líneas de código.
```

```
>>> re.subn(r'\b(R|r)egulares\b', "REGEX", texto)
('El poder de las expresiones REGEX se manifiesta cuando agregamos caracteres\nespeciales a la cadena de búsqueda que nos
permite controlar de manera más precisa\nqué líneas coinciden con la cadena. Agregar estos caracteres especiales a nuestra
s\nexpresiones 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.

IGNORECASE, I: búsquedas sin tener en cuenta las minúsculas o mayúsculas**.** Múltiples banderas pueden ser especificadas utilizando el operador "|" OR.

VERBOSE, X: Los comentarios y espacios son ignorados (en la expresión). Es importante indicar que los espacios SON ignorados por lo que si necesitamos indicar un espacio hay que escaparlo

ASCII, A: Que hace que las secuencias de escape \w, \b, \s and \d funciones para coincidencias con los caracteres ASCII.

DOTALL, S: permite que el punto (.) coincida con la nueva línea

LOCALE, L: Esta opción hace que \w, \W, \b, \B, \s, y \S dependientes de la localización actual.

MULTILINE, M: dentro de una cadena compuesta de muchas líneas, permite que ^ y $ coincidan con el inicio y el final de cada línea. Normalmente ^ / $ solo coincidiría con el inicio y el final de toda la cadena**.**

```
# ejemplo  11 - IGNORECASE
import re
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'regulares\b', re.I)
regex = reg.sub("REGEX", texto)
print(regex)
```

```
>>> reg = re.compile(r'regulares\b', re.I)
>>> 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.
```

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

```
# ejemplo  12 - VERBOSE
import re
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)
```

```
>>> mail.findall(mails)
['aaa.bbbbbb@gmail.com', 'ccc.dddddd@yahoo.com.ar', 'eeeee@github.io', 'river@riverplate.com.ar']
```

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

```
# ejemplo  13
import re
url = re.compile(r"^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$")
print(url.search("https://www.python.org/"))
print(url.search("https://www.google.com/!.html"))
```

```
>>> import re
>>> 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/'>
>>> print(url.search("https://www.google.com/!.html"))
None
```

Para 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]?)$

```
# ejemplo 14
import re
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")
print(ip.search("256.60.124.136"))
```

```
>>> import re
>>> 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")
<re.Match object; span=(0, 13), match='98.61.125.138'>
>>> print(ip.search("256.60.124.136"))
None
```

Para 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)$

```
# ejemplo 15
import re

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"))
print(fecha.search("2-10-1990"))
print(fecha.search("32/12/2021"))
print(fecha.search("30/13/2020"))
```

```
>>> import re
>>> 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'>
>>> print(fecha.search("2-10-1990"))
None
>>> print(fecha.search("32/12/2021"))
None
>>> print(fecha.search("30/13/2020"))
None
```

Análisis the HTML mediante expresiones regulares

La regex del ejemplo busca cadenas que comiencen con “href="http://” o “href=https://, seguido de uno o más caracteres (.+?), seguidos por otra comilla doble . El signo de interrogación después de [s]? , indica que la coincidencia debe ser hecha en modo “ no codicioso”, en vez de en modo “codicioso”. Una búsqueda no-codiciosa intenta encontrar la cadena coincidente más pequeña posible, mientras que una búsqueda codiciosa intentaría localizar la cadena coincidente más grande. Agregamos paréntesis a la regex para indicar qué parte de la cadena localizada queremos extraer.

```
# ejemplo  16

import re
import urllib.request as ur

url= 'https://docs.python.org‘
#probamos con https://docs.python.org ó http://python.org

with ur.urlopen(url) as f:
    html = f.read().decode('utf-8')

regex = re.compile('href="(http*s+?://.*?)"', re.IGNORECASE)

links = re.findall(regex, html)

for link in links:
    print(link)
```

```
https://docs.python.org/3/index.html
https://www.python.org/
https://devguide.python.org/docquality/#helping-with-documentation
https://docs.python.org/3.10/
https://docs.python.org/3.9/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.6/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://devguide.python.org/
https://www.python.org/
https://www.python.org/psf/donations/
https://docs.python.org/3/bugs.html
https://www.sphinx-doc.org/
```

Expresiones Regulares y archivos

```
# ejemplo  17
import re

with open("<ruta del archivo>/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("<ruta del archivo>/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')
```

```
Berlin {'10967', '13629', '10829', ...}
Hamburg {'22527', '22297', '22043', ...}
München {'80538', '81373', '81739', ...}
Köln {'51107', '50939', '50969', ...}
Frankfurt am Main {'60594', '65929', '60549', ...}
Essen {'45326', '45359', '45134', ...}
Dortmund {'44267', '44319', '44357', ...}
Stuttgart {'70174', '70178', '70376', ..}
Düsseldorf {'40589', '40219', '40468', ...}
Bremen {'28777', '28755', '28325', ...}
Hannover {'30559', '30159', '30455', ...}
Duisburg {'47229', '47137', '47059', ...}
Leipzig {'4275', '4157', '4178', ...}
Nürnberg {'90482', '90419', '90439', ...}
Dresden {'1139', '1099', '1465', '...}
Bochum {'44879', '44793', '44803',...}
Wuppertal {'42113', '42107', '42281',...}
Bielefeld {'33729', '33615', '33604', ...}
Mannheim {'68163', '68239', '68169', ...}
```

Comando dir() para localizar los métodos del módulo

```
>>> import re
>>> dir(re)
['A', 'ASCII', 'DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'Match', 'Pattern', 'RegexFlag', 'S', 'Scanner', 'T', 'TEMPLATE', 'U',
 'UNICODE', 'VERBOSE', 'X', '_MAXCACHE', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__'
, '__version__', '_cache', '_compile', '_compile_repl', '_expand', '_locale', '_pickle', '_special_chars_map', '_subx', 'compile', 'copyreg', 'enum', 'error
', 'escape', 'findall', 'finditer', 'fullmatch', 'functools', 'match', 'purge', 'search', 'split', 'sre_compile', 'sre_parse', 'sub', 'subn', 'template']
```

Ayuda sobre los objetos , clases y métodos con help

```
>>> help(re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a Match object, or None if no match was found.
```

https://docs.python.org/3/library/re.html

https://pypi.org/project/regex/

https://www.debuggex.com/

https://regex101.com/

FIN