Master de Programación en Python - Unidad 7

Manejo de Texto

Cadenas de texto

Como ya sabemos, las cadenas de Python son secuencias de caracteres Unicode. Y siendo más preciso son secuencias de valores numéricos que se corresponden con puntos de código Unicode. La función nativa `ord()` convierte un caracter en su versión numérica (punto de código) y `chr()` convierte un valor numérico en un caracter:

In [ ]:

```
print(hex(ord('c')))
print(chr(0x63))
```

```
0x63
c
```

Las cadenas de Pyhton (`str`) implementan todas las operaciones comunes de secuencias que ya conocemos (p. ej. len, min, max, index, count, etc) junto con métodos especificos de su clase que iremos aprendiendo en esta unidad. Además de los métodos nativos también veremos las funciones extendidas para manejo de texto que forman parte de la librería estándar.

El siguiente ejemplo muestra los métodos más utilizados para manipulación básica de cadenas:

In [ ]:

```
cadena = '     hola mundo!     '
print(cadena.upper())          # Retorna una copia de la cadena convertida a mayusculas
print(cadena.strip())          # Retorna una copia eliminando espacios del ppio y final.
                               # Admite un argumento para especificar qué borrar (espacio
s por defecto)
print(cadena.center(28, '*')) # Rellena con asteriscos hasta longitud 28
print(cadena.strip().center(28, '*'))
print()

print(cadena.isalnum())        # Retorna True si todos los cars son alfanum. y contiene al
menos un caracter
print('12.3'.isnumeric())      # Retorna True si todos los cars son numericos y contiene a
l menos un caracter
print(cadena.upper().isupper()) # Retorna True si todos los cars son mayusculas y contien
e al menos un caracter
print(cadena.islower())        # Retorna True si todos los cars son minusculas y al menos
contiene un caracter
print(cadena.isalpha())        # Retorna True si todos los cars son alfabeticos y al menos
contiene un caracter
print()

print(cadena.title())          # Retorna una copia con la primera letra de cada palabra co
nvertida a mayuscula (title cased)
print(cadena.title().lower()) # retorna una copia todo minusculas
```

```
     HOLA MUNDO!
hola mundo!
***     hola mundo!     ****
********hola mundo!*********

False
False
```

```
True
True
False
```

```
    Hola Mundo!
    hola mundo!
```

Si estás interesado en conocer la lista completa de métodos de `str` consulta la [guía de referencia]

Formato de texto

En esta sección veremos las alternativas que nos ofrece Python para presentar nuestros resultados de una forma más atractiva. Hasta ahora hemos venido utilizando métodos muy rudimentarios para formatear nuestros resultados:

In [ ]:

```
q = 459
p = 0.098
print(q, p, p * q)             # salida separada por espacios
print(q, p, p * q, sep=",")    # salida separada por comas
print(str(q) + " " + str(p) + " " + str(p * q))    # Formateo mediante concatenación de c
adenas
```

```
459 0.098 44.982
459,0.098,44.982
459 0.098 44.982
```

Python 3.6 introdujo los literales de formato de cadenas, o más conocidos como *f-strings*, una nueva forma de formatear cadenas que aprenderemos a utilizar enseguida. Pero antes de nada hagamos un breve repaso por todas las opciones que proporciona Python para que nuestro texto se imprima a nuestro gusto

¡Adios a las tediosas concatenaciones de cadenas!

Antes de Python 3.6 se utilizaba el método `format`. Dicho método es extensible via `__format__` de la clase que se convierte a cadena. El método utiliza `{}` para delimitar los campos de sustitución (i.e. se sustitiyen por variables).:

In [ ]:

```
nombre, edad = 'Enrique', 34
print('Hola {}, tienes {} años'.format(nombre, edad))      # Usa el orden proporcionado
print('Hola {}, tienes {} años'.format(edad*2, nombre))    # Usa el orden proporcionado
print('Hola {1}, tienes {0} años'.format(edad, nombre))    # Usa indices para referencia
r variables en orden arbitrario
```

```
Hola Enrique, tienes 34 años
Hola 68, tienes Enrique años
Hola Enrique, tienes 34 años
```

In [ ]:

```
persona = dict(nombre='Enrique', edad='34')
print('Hola {n}, tienes {e} años'.format(n=persona['nombre'], e=persona['edad']))  # Usa
ndo nombres para referenciar atributos de objetos
print('Hola {nombre}, tienes {edad} años'.format(**persona))    # Usando desempaquetado
de diccionarios
```

```
Hola Enrique, tienes 34 años
Hola Enrique, tienes 34 años
```

Los f-strings son literales de cadena especiales que se identifican mediante el prefijo `f`. Contienen `{}` para delimitar las expresiones que serán reemplazadas con sus valores. Las expresiones se evalúan en tiempo de ejecución y luego se formatean utilizando el protocolo `str.__format__`. Aceptan cualquier expresión válida de Python y son evaluadas en el contexto donde aparecen, es decir, tienen acceso completo a las variables locales y globales que la misma expresión tendría fuera del f-string.

La sintaxis de los f-strings es similar a la del método `format` pero más ligera y fácil de leer:

In [ ]:

```
print(f'Hola {nombre}, tienes {edad}')
print(f'{2**3}')      # Puedes usar cualquier expresion váida de Python
print(f'HOLA {nombre.upper()}!!!')  # Incluso llamar funciones o metodos
```

```
Hola Enrique, tienes 34
8
HOLA ENRIQUE!!!
```

Por defecto, los f-strings invocan `__str__` del objeto que se va a convertir en texto. Añadiendo el sufijo de conversión `!r` podemos forzar que `__repr__` sea invocado en lugar de `__str__`:

In [ ]:

```
class Persona:
  def __init__(self, nombre, apellido, edad):
    self.nombre = nombre
    self.apellido = apellido
    self.edad = edad

  def __repr__(self):
    return f'Persona("{self.nombre}", "{self.apellido}", "{self.edad}")'

  def __str__(self):
    return f'{self.nombre} {self.apellido} tiene {self.edad} años'

p = Persona('Enrique', 'Lazaro', 67)
print(f'{p}')
print(f'{p!r}')    # Usando indicador de conversion !r para __repr__
```

```
Enrique Lazaro tiene 67 años
Persona("Enrique", "Lazaro", "67")
```

Además de ser mucho más legibles y potentes que el metodo `format`, las f-strings también son más rápidas.

Los especificadores de formato se incluyen después de `:` dentro de los campos de sustitución. El mini-lenguaje de especificación de formato es exactamente el mismo que el del método `format`. Aquí encontrareis la especificación completa, a continuación veremos unos ejemplos con los usos más comunes:

In [ ]:

```
print(f'dec: {42:d};  hex: {42:x};  oct: {42:o};  bin: {42:b}')     # Salida sin prefijo
print(f'int: {42:};  hex: {42:#x};  oct: {42:#o};  bin: {42:#b}') # Con prefijo
print(f'{12.348:.2f}')      # Precisión de dos decimales
print(f'{0.23:.4%}')        # Porcentaje

test = 'prueba'
print(f'{test:10}fin')      # Texto por defecto se justifica a la izda
print(f'{test:>10}')        # Justificacion a la derecha con relleno
print(f'{12:4}')            # Numeros se justifican a la dcha por defecto
print(f'inicio{12:<4}fin') # Justificacion a la izda
print(f'inicio{12:^4}fin') # Centrado
```

```
dec: 42;  hex: 2a;  oct: 52;  bin: 101010
int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010
12.35
23.0000%
prueba    fin
    prueba
  12
inicio12  fin
inicio 12 fin
```

Existe una tercera via para formatear texto: las plantillas. `Template` es un mecanismo mucho más simple, y

también más limitado, pero tiene sus casos de uso como veremos luego. Ejemplo de uso:

In [ ]:

```
from string import Template

edad = 23
nombre = 'Maria'
t = Template('Hola, $nombre! Tienes $edad')
t.substitute(nombre=nombre, edad=edad)
```

Out[ ]:

```
'Hola, Maria! Tienes 23'
```

Una diferencia clave entre las plantillas y la interpolación o formato de cadenas es que el tipo de los argumentos no se tiene en cuenta. Los valores se convierten en cadenas y las cadenas se insertan en el resultado. No hay opciones de formato disponibles. Por ejemplo, no hay forma de controlar el número de dígitos utilizados para representar un valor de coma flotante.

La recomendación es usar modelos siempre que tengamos que formatear texto introducido por el usuario. Debido a su simplicidad, es la opción de formatear texto más segura y robusta en Python. El uso de `safe_substitute` evita la generación de errores en tiempo de ejecución:

In [ ]:

```
import string

valores = {'var': 'algo'}

t = string.Template('$var está pero $falta no existe')

print('substitute()     :', t.substitute(valores))  # Error!! No existe una de las claves del diccionario
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-33-0a14a13012d3> in <module>()
      5 t = string.Template('$var está pero $falta no existe')
      6
----> 7 print('substitute()     :', t.substitute(valores))  # Error!! No existe una de las claves del diccionario

/usr/lib/python3.6/string.py in substitute(*args, **kws)
    128             raise ValueError('Unrecognized named group in pattern',
    129                              self.pattern)
--> 130         return self.pattern.sub(convert, self.template)
    131
    132     def safe_substitute(*args, **kws):

/usr/lib/python3.6/string.py in convert(mo)
    121             named = mo.group('named') or mo.group('braced')
    122             if named is not None:
--> 123                 return str(mapping[named])
    124             if mo.group('escaped') is not None:
    125                 return self.delimiter

KeyError: 'falta'
```

In [ ]:

```
print('safe_substitute():', t.safe_substitute(valores))
```

```
safe_substitute(): algo está pero $falta no existe
```

El módulo `string` proporciona un conjunto de constantes relacionadas con el juego de caracteres ASCII y los caracteres numéricos

In [ ]:

```
import inspect
import string


def is_str(valor):
    return isinstance(valor, str)


for nombre, valor in inspect.getmembers(string, is_str):
    if nombre.startswith('_'):
        continue
    print(f'{nombre}={valor!r}\n')
```

```
ascii_letters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

ascii_lowercase='abcdefghijklmnopqrstuvwxyz'

ascii_uppercase='ABCDEFGHIJKLMNOPQRSTUVWXYZ'

digits='0123456789'

hexdigits='0123456789abcdefABCDEF'

octdigits='01234567'

printable='0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./
:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'

punctuation='!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

whitespace=' \t\n\r\x0b\x0c'
```

Entrada de texto por teclado

La función nativa `input` permite obtener el texto introducido por teclado. Al llegar a la función, el programa se detiene esperando que se escriba algo y se pulse la tecla Intro, como muestra el siguiente ejemplo:

In [ ]:

```
print('Hola! ¿Cómo te llamas?')
nombre = input()
print(f'Encantado de conocerte {nombre}!!')
```

```
Hola! ¿Cómo te llamas?
Jorge
Encantado de conocerte Jorge!!
```

En el ejemplo anterior, el usuario escribe su respuesta en una línea distinta a la pregunta porque Python añade un salto de línea al final de cada print().

Opcionalmente podriamos utilizar el parámetro `end` de la función `print` para configurar la formar de terminar nuestra salida de texto. Por defecto, Python termina con un cambio de línea como acabamos de comprobar. En el siguiente ejemplo, hemos añadido un espacio al final de la pregunta para separarla de la respuesta:

In [ ]:

```
print('Hola! ¿Cómo te llamas? ', end='')
nombre = input()
print(f'Encantado de conocerte {nombre}!!')
```

```
Hola! ¿Cómo te llamas? Jorge
Encantado de conocerte Jorge!!
```

Otra solución, más compacta, es aprovechar que a la función `input()` admite un argumento que se imprime por pantalla (sin añadir un salto de línea). Fijaos que tenemos que añadir un espacio al final de la pregunta para separarla de la respuesta del usuario (igual que antes):

In [ ]:

```
nombre = input('Hola! ¿Cómo te llamas? ')
print(f'Encantado de conocerte {nombre}!!')
```

```
Hola! ¿Cómo te llamas? Jorge
Encantado de conocerte Jorge!!
```

La función `input` retorna siempre una cadena de texto, incluso si la respuesta del usuario son números. Por ello, si deseamos realizar operaciones aritméticas sobre la entrada proporcionada por el usuario debemos convertir explicitamente al tipo numerico deseado:

In [ ]:

```
cantidad = int(input('Introduce una cantidad en pesetas para convertirla a euros: '))
print(f'{cantidad} pesetas son {round(cantidad / 166.386, 2)} euros')
```

```
Introduce una cantidad en pesetas para convertirla a euros: 123
123 pesetas son 0.74 euros
```

Igualmente si queremos que Python interprete la entrada del usuario como un número decimal debemos convertirlo explicitamente:

In [ ]:

```
cantidad = float(input('Introduce una cantidad en euros (hasta con 2 decimales): '))
print(f'{cantidad} euros son {round(cantidad * 166.386)} pesetas')
```

```
Introduce una cantidad en euros (hasta con 2 decimales): 12.25
12.25 euros son 2038 pesetas
```

En caso de que el usuario no introduzca un número, cualquiera de las conversiones explicitas generará un error:

In [ ]:

```
cantidad = float(input('Introduce una cantidad en euros (hasta con 2 decimales): '))
print(f'{cantidad} euros son {round(cantidad * 166.386)} pesetas')
```

```
Introduce una cantidad en euros (hasta con 2 decimales): 34,23

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-42-ff7bb3702628> in <module>()
----> 1 cantidad = float(input('Introduce una cantidad en euros (hasta con 2 decimales):
'))
      2 print(f'{cantidad} euros son {round(cantidad * 166.386)} pesetas')

ValueError: could not convert string to float: '34,23'
```

De igual modo, si el usuario introduce un número decimal cuando el programa espera un entero, se generará un error. Sin embargo, cuando se espera un decimal da igual introducir un entero o un flotante:

In [ ]:

```
cantidad = float(input('Introduce una cantidad en euros (hasta con 2 decimales): '))
print(f'{cantidad} euros son {round(cantidad * 166.386)} pesetas')
```

```
Introduce una cantidad en euros (hasta con 2 decimales): 12
12.0 euros son 1997 pesetas
```

La función `input` sólo toma un argumento, nuestro texto de salida. Gracias a las f-strings podemos incorporar variables a la función `print` de una manera legible:

In [ ]:

```
nombre = input('Introduzca su nombre: ')
```

```
apellido = input(f'Apellido, {nombre}: ')
print(f'Encantado de conocerte, {nombre} {apellido}.')
numero1 = int(input('Diga un número: '))
numero2 = int(input(f'Diga un número mayor que {numero1}: '))
print(f'La diferencia entre ellos es {numero2 - numero1}.')
```

```
Introduzca su nombre: Jorge
Apellido, Jorge: Lopez Arienza
Encantado de conocerte, Jorge Lopez Arienza.
Diga un número: 24
Diga un número mayor que 24: 124
La diferencia entre ellos es 100.
```

Separar cadenas

El método `split(sep=None, maxsplit=-1)` retorna una lista de palabras, utilizando el argumento `sep` como delimitador. `maxsplits` establece un límite superior al número de palabras retornadas. En este caso, se devuelve una lista de `maxsplit+1` elementos. El último elemento contiene el resto de la cadena sin procesar. Por defecto, no hay límite y se usa espacio como separador .

Si se especifica un separador, la existencia de delimitadores contiguos generan palabras vacias (i.e. ''):

In [4]:

```
print('hola mundo'.split())                    # El separador por defecto es el espacio
print('1,,, ,2,a'.split(sep=','))                 # Los separadores contiguos no se agrupan
y retornan palabras vacias
print('texto con  varios espacios'.split()) # Salvo cuando no se especifica separador. Lo
s espacios contiguos se agrupan en un único separador
print(''.split())                              # split de una cadena vacia devuelve una lis
ta vacia
print('1<>2<>4<>34'.split('<>'))              # Admite un numero arbitrario de caracteres
como separador
```

```
['hola', 'mundo']
['1', '', '', ' ', '2', 'a']
['texto', 'con', 'varios', 'espacios']
[]
['1', '2', '4', '34']
```

In [5]:

```
# Desempaquetando el valor devuelto
colores = 'azul, blanco, rojo'
c1, c2, c3 = colores.split(',')  # Notad que el espacio no se elimina
print(c1)
print(c2)
print(c3)
```

```
azul
 blanco
 rojo
```

In [6]:

```
# Poniendo límites a la salida: devuelve sólo los 3 primeros y el resto sin separar
ciudades = 'Madrid,Coruña,Pontevedra,Sevilla,Valencia,Barcelona'.split(',', 3)
for c in ciudades:
  print(c)
```

```
Madrid
Coruña
Pontevedra
Sevilla,Valencia,Barcelona
```

In [7]:

```
ciudades = 'Madrid,Coruña,Pontevedra'.split(',', 3)
print(len(ciudades))
```

```
for c in ciudades:
  print(c)
```

```
3
Madrid
Coruña
Pontevedra
```

In [8]:

```
ciudades = 'Madrid,Coruña,Pontevedra,Sevilla,Valencia,Barcelona'.split(',')
for c in ciudades:
  print(c)
```

```
Madrid
Coruña
Pontevedra
Sevilla
Valencia
Barcelona
```

El método `rsplit` realiza la misma operación empezando por la derecha.

In [9]:

```
ciudades = 'Madrid,Coruña,Pontevedra,Sevilla,Valencia,Barcelona'.rsplit(',', 3)
for c in ciudades:
  print(c)
```

```
Madrid,Coruña,Pontevedra
Sevilla
Valencia
Barcelona
```

In [10]:

```
print(ciudades)
```

```
['Madrid,Coruña,Pontevedra', 'Sevilla', 'Valencia', 'Barcelona']
```

Convertir secuencias en texto

Podemos tomar una secuencia (cuyos elementos sean cadenas) y transformarla en una cadena, usando la función `join()`. Esta función coloca un separador entre los elementos consecutivos de la secuencia para formar la cadena de salida. Si guardamos nuestro separador en una variable s, debemos escribir:

```
s.join(secuencia)
```

In [11]:

```
sep = ','
print(sep.join(('hola', 'mundo')))     # Tupla a texto
print(''.join({'h', 'o', 'l', 'a'}))   # Conjunto a texto: fijaos el desorden
print(sep.join(dict(uno='hola', dos='mundo')))   # Por defecto convierte las claves
print(sep.join(dict(uno='hola', dos='mundo').values()))   # Pero puedo convertir en texto los valores del diccionario tambien
print(sep.join(['hola', 'mundo']))     # Lista a texto
```

```
hola,mundo
ahlo
uno,dos
hola,mundo
hola,mundo
```

Buscar texto al principio o al final

El método `startswith(prefix)` devuelve `True` si la cadena comienza con el texto especificado por el

argumento `prefix` y `False` en caso contrario. `prefix` puede contener una tupla con varias cadenas para comprobar. Los parámetros opcionales `start` y `stop` delimitan la zona de búsqueda dentro de la cadena original.

In [12]:

```
url = 'https://ejemplo.com'
print(url.startswith('https:'))
print(url.startswith('http:'))
print()

esquemas = ('mailto:', 'http:', 'https', 'file:', 'ftp:')
url1 = 'ftp://ftp.ejemplo.com'
print(url1.startswith(esquemas))    # Ejemplo con multiples cadenas de búsqueda
url2 = 'icap://icap.example.net:2000/services/icap-service-1'
print(url2.startswith(esquemas))
```

```
True
False

True
False
```

De manera similar, el método `endswith(suffix)` devuelve `True` se el texto acaba con la cadena especificada en el argumento `suffix` y `False en caso contrario.

In [13]:

```
nombres_archivo = ['test1.pptx', 'test2.txt', 'test3.docx', 'test4.doc', 'test1.xls', 't
est6.txt']
print([n for n in nombres_archivo if n.endswith('.txt')])
print(any([n for n in nombres_archivo if n.endswith(('.doc', '.docx'))]))  # Ejemplo con
multiples cadenas de búsqueda
```

```
['test2.txt', 'test6.txt']
True
```

Búsqueda de subcadenas

El método `find(sub)` devuelve el menor índice dentro de la cadena original donde se encuentra `sub` y devuelve `-1` en caso de no encontrar la subcadena. Opcionalmente se puede limitar la búsqueda a un tramo de la cadena mediante los argumentos opcionales `start` y `stop`. En este caso, `stop` no se incluye en la zona de búsqueda.

Existe un método equivalente `index` que lanza un error cuando no encuentra la subcadena.

Tanto `find` como `index` realizan la búsqueda empezando por la izquierda. Existen las versiones correspondientes `rfind` y `rindex` que realizan la búsqueda empezando por la derecha (busqueda inversa).

In [14]:

```
texto = 'Pablito clavo un clavito, ¿que clavito clavo Pablito'
print(texto.find('ito'))  # Busca terminación -ito: devuelve primera ocurrencia empezando
por la izda
print(texto.find('ito', 5, 30))  # Busca terminación -ito en otra parte del texto
print(texto.find('ato'))  # Devuelve -1 cuando no encuentra la subcadena
print(texto.index('ito'))  # index funciona exactamente igual que find
print(texto.index('ato'))  # Salvo que lanza error si no encuentra la subcadena
```

```
4
21
-1
4

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-fd8ed1e65fda> in <module>()
```

```
      4 print(texto.find('ato'))   # Devuelve -1 cuando no encuentra la subcadena
      5 print(texto.index('ito'))   # index funciona exactamente igual que find
----> 6 print(texto.index('ato'))   # Salvo que lanza error si no encuentra la subcadena

ValueError: substring not found
```

Los mismos ejemplos con sus versiones inversas:

In [15]:

```
print(texto.rfind('ito'))       # Busca la última ocurrencia
print(texto.rfind('ito', 3, 10))
print(texto.rindex('ito'))
```

```
49
4
49
```

Acordaos, si solo necesitamos saber si una cadena contiene una subcadena utilizamos el operador de pertencia:

In [16]:

```
'ito' in 'Pablito'
```

Out[16]:

```
True
```

Búsqueda de patrones de texto

El módulo `re` de la librería estándar proporciona herramientas de expresiones regulares para un procesamiento avanzado de cadenas. Para manipulación y coincidencias complejas, las expresiones regulares ofrecen soluciones concisas y optimizadas.

Una expresión regular, también conocida como regex, es una secuencia de caracteres que especifican un patrón de búsqueda y permite filtrar textos para encontrar coincidencias, comprobar la validez de fechas, documentos de identidad o contraseñas. También se utiliza para reemplazar texto con unas características concretas por otro, y para dividir cadenas de texto. En realidad, regex es un mini-lenguaje de programación especializado en especificación de reglas para definir patrones de busquéda.

Las expresiones regulares utilizan el carácter de barra invertida para permitir el uso de caracteres especiales sin invocar su significado especial. Esto choca con el uso de Python de `\` como carácter de escape en literales de texto; por ejemplo, para buscar una barra invertida en un literal, habría que escribir '\\' como patrón de búsqueda, porque la expresión regular debe ser \, y cada barra invertida debe expresarse como \ dentro de un literal de cadena.

La solución es usar la notación de literal de cadena crudo (*raw*) de Python para especificar patrones regex. Las barras invertidas no tienen ningún significado especial en un literal de texto con prefijo 'r'. Por lo general, usaremos literales crudos para definir expresiones regulares en el código. Veamos un ejemplo:

In [20]:

```
import re

# Notad el prefijo r para especificar el patrón de busqueda regex
print(re.findall(r'\bt[a-z]*', 'tres tristes tigres comen trigo en un trigal'))
print(re.sub(r'(\b[a-z]+) \1', r'\1', 'gato en el el sombrero'))
```

```
['tres', 'tristes', 'tigres', 'trigo', 'trigal']
gato en el sombrero
```

En Python, una búsqueda regex utiliza normalmente la siguiente sintaxis:

```
coincidencia = re.search(patron, cadena)
```

El método `re.search` toma un patrón regex y una cadena y busca dicho patrón dentro de la cadena. Si la búsqueda es exitosa devuelve un objeto coincidente. En caso contrario devuelve `None`.

Patrones básicos

El poder de las expresiones regulares radica en que se pueden especificar patrones, no solo caracteres fijos. A continuación se listan los patrones más básicos que coinciden con los caracteres individuales:

- **a, A, 9: los caracteres normales coinciden exactamente consigo mismos**
- **. (un punto): especifica cualquier carácter excepto cambio de línea `\n`**
- **\w: especifica carácter alfanumérico y guión bajo**
- **\W: especifica cualquier carácter no alfanumérico**
- **\b: límite entre palabra y no palabra**
- **\s: especifica "carácter de espacio en blanco" como espacio, nueva línea, retorno, tabulador**
- **\d: dígito decimal [0-9]**
- **^: indica inicio de una cadena**
- **$: indica final de una cadena**

Las reglas básicas de la búsqueda de patrones a tener en cuenta son las siguientes:

1. **La búsqueda se realiza desde el inicio hasta el final y se detiene en la primera coincidencia**
2. **Todo el patrón debe coincidir, pero no toda la cadena**
3. **Si la búsqueda tiene éxito el método `search` devuelve un objeto `match`. `match.group()` retorna el texto coincidente**

Veamos algunos ejemplos:

In [21]:

```
import re

coincidencia = re.search(r'ooo', 'hooola')    # Busca triple 'o' en la cadena
print(coincidencia.group())
coincidencia = re.search(r'lao', 'hooola')    # Si no encuentra coincidencias devuelve Non
e
print(coincidencia)
print()

coincidencia = re.search(r'..l', 'hooola')    # Patron buscado: dos caracteres cualquiera
y una l
print(coincidencia.group())
coincidencia = re.search(r'\d\d\d', 'p123g') # Busca 3 digitos seguidos
print(coincidencia.group())
coincidencia = re.search(r'\w\w\w', '@abcd34!!') # Busca 3 alfanumericos seguidos
print(coincidencia.group())
```

```
ooo
None

ool
123
abc
```

Repeticiones

- **+: indica 1 o más repeticiones del patrón especificado a su izquierda**
- **`*`: indica 0 o más repeticiones del patrón especificado a su izquierda**
- **?: indica 0 o 1 repetición del patrón especificado a su izda**

Veamos unos ejemplos:

In [22]:

```
match = re.search(r'ho+', 'hooooooola')   # o+ = una o mas "o"
print(match.group())
```

```
match = re.search(r'o+', 'pooloooo') # Se detiene en la primera coincidencia
print(match.group())
print()

# \s* indica cero o más caracteres de espacio
# Busca 3 digitos, posiblemente separados por espacios
match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx')
print(match.group())
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx')
print(match.group())
match = re.search(r'\d\s*\d\s*\d', 'xx123xx')
print(match.group())
print()

match = re.search(r'^b\w+', 'cabra') # Busca b seguido de uno o más caracteres alfanumer
icos al comienzo de la cadena
print(match)
match = re.search(r'b\w+', 'foobar') # Mismo patrón pero puede estar en cualquier sitio
de la cadena
print(match.group())
```

```
hooooooo
oo

1 2   3
12  3
123

None
bar
```

In [23]:

```
type(match.group())
```

Out[23]:

```
str
```

In [24]:

```
type(match)
```

Out[24]:

```
_sre.SRE_Match
```

Conjuntos de caracteres y agrupación de resultados

Los corchetes se usan para especificar un conjunto de caracteres. Dentro de los corchetes también se pueden especificar rangos usando un guión para indicar comienzo y fin. Además, dentro de los corchetes se pueden incluir los patrones básicos excepto el punto. Cuando usamos el punto dentro de corchetes se interprete como el caracter '.'

Dentro de un regex, los paréntesis se usan para agrupar los resultados de manera que podemos selectivamente usar partes del texto encontrado. Se accede mediante un índice (empieza a contar desde la izda en 1).

Veamos varios ejemplos:

In [25]:

```
# Extrae dirección de correo electronico de un texto
texto = 'cadena nombre-usuario@ejemplo.com colores y numeros'
match = re.search(r'[\w.-]+@[\w.-]+', texto)
if match:
  print(match.group())
```

```
nombre-usuario@ejemplo.com
```

In [26]:

```
# Extrae nombre de usuario y dominio de la direccion de email
match = re.search(r'([\w.-]+)@([\w.-]+)', texto)    # Usamos parentesis para agrupar la s
alida
if match:
  print(match.group())    # Todo el resultado
  print(f'Nombre de usuario: {match.group(1)}')  # Notad el uso del indice para acceder a
los grupos: empieza por 1!!!
  print(f'Dominio: {match.group(2)}')
```

```
nombre-usuario@ejemplo.com
Nombre de usuario: nombre-usuario
Dominio: ejemplo.com
```

A diferencia de `search` que sólo devuelve la primera ocurrencia, `findall` devuelve todas las ocurrencias encontradas en una lista de cadenas donde cada elemento es una ocurrencia.

In [27]:

```
# Extrae las direcciones de email de un texto
texto = 'azul user1@ejemplo.com, verde jleon@otro-ejemplo.net 123 colores, bla, bla, bla'
emails = re.findall(r'[\w\.-]+@[\w\.-]+', texto)
for email in emails:
  print(email)
```

```
user1@ejemplo.com
jleon@otro-ejemplo.net
```

Si estamos utilizando paréntesis para agrupar resultados, entonces `findall` devuelve una lista de tuplas (en lugar de una lista de cadenas). Cada tupla representa una coincidencia y cada elemento de la tupla se corresponde con un grupo:

In [28]:

```
# Extre nombre de usuario y dominio para cada email
res = re.findall(r'([\w\.-]+)@([\w\.-]+)', texto)
print(res)
for r in res:
  print(f'Nombre de usuario: {r[0]}')
  print(f'Dominio: {r[1]}')
```

```
[('user1', 'ejemplo.com'), ('jleon', 'otro-ejemplo.net')]
Nombre de usuario: user1
Dominio: ejemplo.com
Nombre de usuario: jleon
Dominio: otro-ejemplo.net
```

La función `re.sub(patron, sustituto, cadena)` busca todas las coincidencias de `patrón` en la cadena dada, y las reemplaza por `sustituto`. La cadena de reemplazo puede incluir '\ 1', '\ 2', que se refiere al texto de grupo (1), grupo (2), etc. del texto original que coincide. Se puede limitar el número de reemplazos a un máximo especificado por el argumento opcional `count`:

In [29]:

```
# Cambia el domino de todas las direcciones de email
print(re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@yo-yo.com', texto))
```

```
azul user1@yo-yo.com, verde jleon@yo-yo.com 123 colores, bla, bla, bla
```

A continuación os dejo una par de enlaces que os pueden resultar de utiles cuando trabajeis con expresiones regulres:

Tutorial regex

Prueba tus regex online

Sustitución de texto

Sustitución de texto

El método `replace(antiguo, nuevo)` devuelve una copia de la cadena con todas las apariciones de la subcadena especificada en `antiguo` reemplazadas por `nuevo`. El parámetro opcional `count` permite especificar el número máximo de sustituciones. El método distingue entre mayúsculas y minúsculas.

In [30]:

```
cadena = 'Este ejemplo parece bastante facil'
print(cadena.replace('Este', 'Aquel'))
print(cadena.replace('este', 'Aquel'))  # Distingue may-min: no encuentra coincidencias
y no reemplaza nada
```

```
Aquel ejemplo parece bastante facil
Este ejemplo parece bastante facil
```

In [ ]:

```
cadena = 'Este es facil y esos no son ni faciles ni dificiles'
print(cadena.replace('es', 'era', 3))  # notad que el cuarto es no ha sido sustituido
```

El método translate

El método `translate(table)` devuelve una copia del texto de entrada en la que los caracteres se han traducido usando una tabla. Para generar la tabla se usa el método `maketrans(x[, y[, z]])`. Si solo hay un argumento, éste debe ser un diccionario que asigne ordinales de Unicode (enteros) o caracteres (cadenas de longitud 1) a ordinales de Unicode, cadenas (de longitudes arbitrarias) o `None`. Las claves de caracteres se convertirán a ordinales.

Si hay dos argumentos, deben ser cadenas de igual longitud, y en el diccionario resultante, cada carácter en `x` se asignará al carácter en la misma posición en `y`. Si hay un tercer argumento, debe ser una cadena, cuyos caracteres se asignarán a `None` en el resultado, es decir, se borran.

A continuación vemos un ejemplo de uso:

In [31]:

```
# Sustituye las vocales por digitos
texto = 'esto es un texto de ejemplo... toma ya!!'
x = "aeiou"
y = "43102"
tabla = texto.maketrans(x, y)

print(texto.translate(tabla))
```

```
3st0 3s 2n t3xt0 d3 3j3mpl0... t0m4 y4!!
```

In [32]:

```
# Igual pero borrando las exclamaciones
texto = 'esto es un texto de ejemplo... toma ya!!'
x = "aeiou"
y = "43102"
tabla = texto.maketrans(x, y, '!')

print(texto.translate(tabla))
```

```
3st0 3s 2n t3xt0 d3 3j3mpl0... t0m4 y4
```

In [33]:

```
# Igual que el último ejemplo pero usando dict
d = {
    'a': '4',
    'e': '3',
    'i': '1',
    'o': '0',
```

```
    'u': '2',
    '!': None,
}
tabla = texto.maketrans(d)
print(texto.translate(tabla))
```

```
3st0 3s 2n t3xt0 d3 3j3mpl0... t0m4 y4
```