strings

January 28, 2020

1 Cadena de Caracteres

Una cadena de caracteres tambien se le conoce como *cadena* o *string* (por su nombre en ingles).

1.1 Que es un string?

En el modulo anterior, creamos un string / cadena `Hola Mundo` y lo imprimimos en la ventana interactiva utilizando la función `print()`.

En esta sección vamos a aprender lo que es un string / cadena y las distintas maneras de crearlos en Python.

1.1.1 El String como Tipo de Dato

El string o cadena de caracteres es un tipo de dato fundamental en Python. La frase *tipo de dato* hace referencia al tipo de dato que representa un valor determinado. En ese sentido, los strings son utilizados para representar texto. Hay que destacar que existen otro tipo de datos, como los que representan numeros por ejemplo, lo cual veremos mas adelante.

Decimos que es un tipo de dato **fundamental** porque no puede ser descompuesto en valores mas pequeños de otro tipo. No obstante, no todos los tipos de datos son fundamentales. Tambien existen los tipos de datos compuestos, conocidos como estructuras de datos, que veremos mas adelante.

El string tiene una abreviacion especial en Python: `str`.

Utlizemos la función incorporada `type()` para determinar el tipo de dato de un valor.

[1]:
```
print(type('Hola Mundo'))
```

```
<class 'str'>
```

[2]:
```
frase = 'Hola Mundo'
print(type(frase))
```

```
<class 'str'>
```

[3]:
```
type(frase)
```

[3]: str

Los strings contienen tres propiedades que exploraremos en las próximas secciones: 1. Contienen caracteres 2. Tienen un tamaño / longitud, determinado por el numero de caracteres 3. Los caracteres aparecen en una secuencia (cada numero tiene una posición númerica determinada)

1.1.2 String Literals

Como ya han podido observar, podemos crear strings al rodearlos por citas sencillas o comillas dobles:

```
string1 = 'Hola'
string2 = "Mundo"
```

Cada vez que creamos un string de esta forma, el string se le llama **string literal** o **literal de cadena de caracteres**. Esto quiere decir que el string esta literalmente escrito en el codigo.

Las citas o comillas alrededor de un string son llamados **delimiters** o delimitadores, toda vez que marcan los limites de un string, informandole a Python donde empieza y donde termina.

Cuando un tipo de citas es utilizado como delimitador, el otro tipo de citas podrá ser utilizado dentro del string:

```
string3 = "'Hola', dijo Adriaan"
string4 = 'Yo respondi "Hola" también'
```

Python lee el primer delimitador, y todos los caracteres despues del mismo son considerados parte del string hasta encontrar el segundo delimitador. Por tanto, no podemos utilizar el mismo delimitador varias veces dentro de un string.

[4]:
```
texto = "Ella preguntó, "Qué hora es?""  # Python no sabe como interpretarlo
```

```
  File "<ipython-input-4-fc177b6de3c0>", line 1
    texto = "Ella preguntó, "Qué hora es?""  # Python no sabe como␣
↪interpretarlo
                                ^
SyntaxError: invalid syntax
```

Consejo: Apeguense a una forma de escribir un string en un proyecto. Es considerada mala practica escribir strings de las dos formas en el mismo proyecto.

1.1.3 La longitud de un string

La longitud de un string esta compuesto por el numero de caracteres, incluyendo espacios, contenidos en el mismo. Por ejemplo, el string `abc` tiene una longitud de `3`. Para determinar la longitud de un string, podemos utilizar la funcion incorporada `len()`.

[10]:
```
len('abc')
```

[10]: 3

[11]:
```
letras = 'abc'
len(letras)
```

[11]: 3

[12]:
```
num_letras = len(letras)
num_letras
```

[12]: 3

1.1.4 Strings Multilineales

Conforme a la guía oficial de estilo de Python, cada linea de Python puede contener hasta 79 caracteres incluyendo espacios. Sin embargo, muchos programadores piensan que este limite es muy corto y, por tanto, incrementan el limite.

A veces tenemos que crear strings muy largos, que se exceden del limite. Para manejar esta limitante, podemos romper el string en varias lineas.

Una manera de crear un string multilinear, se hace insertando un \ al final de cada linea.

[13]:
```
parrafo =  "Alcanzamos por fin la victoria \
en el campo feliz de la unión; \
con ardientes fulgores de gloria \
se ilumina la nueva nación."

print(parrafo)
```

```
Alcanzamos por fin la victoria en el campo feliz de la unión; con ardientes
fulgores de gloria se ilumina la nueva nación.
```

Observemos: * no tuvimos que insertar una cita al final de cada linea * la impresion del parrafo se refleja en una sola linea

Strings multilineales tambien pueden ser escritos utilizando triple citas, a fin de conservar los espacios.

[14]:
```
parrafo =  """Alcanzamos por fin la victoria
en el campo feliz de la unión;
con ardientes fulgores de gloria
se ilumina la nueva nación."""
print(parrafo)
```

```
Alcanzamos por fin la victoria
en el campo feliz de la unión;
con ardientes fulgores de gloria
se ilumina la nueva nación.
```

[15]:
```
parrafo =  '''Alcanzamos por fin la victoria
en el campo feliz de la unión;
```

```
con ardientes fulgores de gloria
se ilumina la nueva nación.'''
print(parrafo)
```

```
Alcanzamos por fin la victoria
en el campo feliz de la unión;
con ardientes fulgores de gloria
se ilumina la nueva nación.
```

1.1.5 Ejercicios

1. Imprime un string que utilize comillas adentro de un string.
2. Imprime un string que utiliza una apostrofe adentro de un string.
3. Imprime un string multilinear, que conserve el espacio.
4. Imprime un string multilinear pero que no sonserve el espacio.

1.2 Concatenar, Indexar y Cortar / Segmentar

Ahora que sabemos lo que es un string y como declararlo, vamos a explorar algunas cosas que podemos hacer con ellos. En este seccion, aprenderemos tres operaciones básicas: * Concatenación, es decir, la unión de dos strings. * Indexación, es decir, la obtención de un caracter de un string. * Segmentación, es decir, la obtención de varios caracteres de un string.

1.2.1 Concatenar

[16]:
```
# Concatenacion de strings
string1 = 'abra'
string2 = 'cadabra'
magia = string1 + string2
print(magia)  # notemos que no hay espacio entre medio
```

```
abracadabra
```

[17]:
```
nombre = 'Juan'
apellido = 'Perez'
nombre_completo = nombre + ' ' + apellido
print(nombre_completo)
```

```
Juan Perez
```

[18]:
```
# Tambien se puede concatenar directamente en el print
print('abra' + 'cadabra')
```

```
abracadabra
```

[19]:
```
# Podemos imprimir varias palabras a la vez
print('abra', 'cadabra')
```

```
abra cadabra
```

1.2.2 Indexar

Cada caracter en un string tiene un posición numerada llamada un índice. Podemos acceder al caracter en una determinada posición insertando el numero entre corchetes.

[20]:
```
# Un string es una secuencia de caracteres; por tanto, podemos acceder a cada␣
↪caracter de manera individual
fruta = 'manzana'
print(fruta[2])
```

```
n
```

[21]:
```
# En Python, se empieza a contar desde cero
print(fruta[0])
```

```
m
```

[22]:
```
# El numero que usamos para acceder a un caracter conforme a su posicion se␣
↪llama indice o index
for indice, letra in enumerate(fruta):
    print(indice, letra)
```

```
0 m
1 a
2 n
3 z
4 a
5 n
6 a
```

[23]:
```
# Intentemos obtener el indice 7
fruta[7]  # fuera de rango
```

```
      ␣
↪---------------------------------------------------------------------------

       IndexError                                Traceback (most recent call␣
↪last)

       <ipython-input-23-06f021574b60> in <module>
         1 # Intentemos obtener el indice 7
   ----> 2 fruta[7]  # fuera de rango


       IndexError: string index out of range
```

[24]:
```
# El indice mas alto en un string siempre es uno menos que la longitud del mismo
len(fruta)
```

[24]:
```
7
```

[25]:
```
fruta[6]
```

[25]:
```
'a'
```

[26]:
```
# Podemos acceder al ultimo índice con -1
fruta[-1]
```

[26]:
```
'a'
```

[27]:
```
# Podemos seguir accediendo a todos los numeros desde fin a principio con -1,␣
↪-2, -3, etc...
print(fruta[-1], fruta[-2], fruta[-3], fruta[-4])
```

```
a n a z
```

[28]:
```
posicion = -1
for letra in fruta[::-1]:
    print(f'{posicion} => {letra}')
    posicion -= 1
```

```
-1 => a
-2 => n
-3 => a
-4 => z
-5 => n
-6 => a
-7 => m
```

1.2.3 Segmentar

Supongamos que necesitas solamente las tres primeras letras de un string. Podemos acceder cada caracter por indice y concatenarlos.

[29]:
```
tres_primeras = fruta[0] + fruta[1] + fruta [2]
tres_primeras
```

[29]:
```
'man'
```

Si necesitamos mas caracteres, estamos claro que hacer esto por cada uno de ellos es algo manual, repetitivo y tedioso. Existe una mejor manera de hacerlo.

Para extraer una porcion o segmento de un string, a lo cual se le conoce como un **substring**, debemos insertar un colon entre los dos indices adentro de una corcheta. Este metodo de extraer un segmento de un string se le llama **slicing** (acción de cortar en pedazos).

[30]:
```
# Podemos extraer una seccion de la secuencia, de un indice al otro (esto se␣
↪llama slicing o slice)
fruta[0:3]  # comenzamos en 0 y paramos justo antes de 3
```

[30]: 'man'

[31]:
```
for indice, letra in enumerate(fruta):
    print(indice, letra)
```

```
0 m
1 a
2 n
3 z
4 a
5 n
6 a
```

[32]:
```
# Si omitimos el primer o segundo numero, Python asume que es uno de los dos␣
↪extremos
print(fruta[:5], fruta[5:], fruta[:])
```

```
manza na manzana
```

[33]:
```
# si intentas cortar un segmento que no existe
fruta[7:9]
```

[33]: ''

[34]:
```
# tambien podemos usar los numeros negativos
fruta[-6:-1]
```

[34]: 'anzan'

[35]:
```
# no podemos cortar de derecha a izquierda
fruta[-6:0]
```

[35]: ''

[36]:
```
# Strings son inmutables, por lo que no se puede cambiar un pedazo del mismo
fruta[0] = 'l'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)
```

```
        <ipython-input-36-6ad05f9d00ce> in <module>
          1 # Strings son inmutables, por lo que no se puede cambiar un pedazo␣
↪del mismo
       ----> 2 fruta[0] = 'l'


        TypeError: 'str' object does not support item assignment
```

[37]:
```
# Para poder hacerlo, debemos crear un nuevo string
nuevo = 'l' + fruta[1:5]
print(nuevo)
```

```
lanza
```

Ejercicios 1. Crea un nuevo string e imprime el numero de caracteres 2. Crea dos strings, concatenalos e imprime la combinacion de los dos 3. Crea dos variables con strings e imprime uno depues de otro con un espacio en el medio 4. Imprime la palabra "riqui" utilizando slicing sobre la palabra "Chiriqui"

1.3 Metodos para la Manipulación de Strings

Las cadenas / strings vienen con funciones especiales conocidas en ingles como **string methods** o metodos de cadenas. Estos metodos se utilizan para manipular las cadenas y hay alrededor de 45 metodos disponibles. No obstante, nos enfocaremos en los mas comunes.

En esta sección aprenderemos a: * Convertir un string en mayuscula o minuscula * Eliminar espacios en un string * Determinar si un string empieza o termina con un ciertos caracteres

[38]:
```
metodos = [metodo for metodo in dir(str) if not metodo.startswith('_')]
```

[39]:
```
# cantidad de metodos de cadenas
len(metodos)
```

[39]:
```
45
```

[40]:
```
metodos
```

[40]:
```
['capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
```

```
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

1.3.1 Mayusculas o minusculas

Para convertir todos los caracteres de un string en minusculas, podemos utilizar el metodo `.lower()`. Se utiliza agregando `.lower()` al final del string.

[41]:
```
'Juan Perez'.lower()
```

[41]:
```
'juan perez'
```

El punto `.` le dice a Python que lo que sigue es un metodo. Los metodos de strings funcionan con variables cuyo valor es un string.

[42]:
```
nombre = 'Juan Perez'
nombre.lower()
```

[42]:
```
'juan perez'
```

[43]:
```
# podemos convertirlo a mayusculas tambien
nombre.upper()
```

[43]:
```
'JUAN PEREZ'
```

Comparemos los metodos `.upper()` y `.lower()` con la función incorporada `len()` que vimos anteriormente. Aparte de que producen resultados distintos, la distincion primordial radica en como son utilizados. La funcion `len()` es una función independiente. Si queremos obtener la longitud de un string, podemos ejecutar la funcion directamente, sin necesidad de agregarla despues de un objeto, e.g. `cadena.metodo()`

[44]:
```
len(nombre)
```

[44]:
```
10
```

[45]:
```
# sin embargo, .lower() o upper() se ejecutan agregandolo a un string
nombre.lower()
```

[45]:
```
'juan perez'
```

1.3.2 Eliminando los espacios de una cadena

El espacio es un caracter que se imprime como un espacio en blanco / transparente entre dos caracteres. A veces, necesitamos eliminar el espacio ubicado al principio o al final de una cadena / string. Esto es util cuando trabajamos con datos de entrada del usuario, donde existe el riesgo que espaciones adicionales se hayan introducido por accidente.

Existen tres metodos de string para eliminar espacios en los mismos: * `.rstrip()` => Right Strip. Elimina espacio ubicado a la derecha del string. * `.lstrip()` => Left Strip. Elimina espacio ubicado a la izquierda del string. * `.strip()` => Strip. Elimina espacio en ambos lados del string

[46]:
```
oracion = 'Tengo espacios a mi derecha.       '
```

[47]:
```
negacion = 'Ya no'
negacion + ' ' + oracion.rstrip()
```

[47]:
```
'Ya no Tengo espacios a mi derecha.'
```

[48]:
```
oracion = '    Tengo espacios a mi izquierda.'
```

[49]:
```
negacion + ' ' + oracion.lstrip()
```

[49]:
```
'Ya no Tengo espacios a mi izquierda.'
```

[50]:
```
oracion = '   Tengo espacios en ambos lados   '
negacion + ' ' + oracion.strip()
```

[50]:
```
'Ya no Tengo espacios en ambos lados'
```

Es importante destacar que ninguno de los metodos elimina los espacios en el medio.

1.3.3 Determina si un string empieza o termina con un string particular

Cuando trabajamos con texto, a veces necesitamos determinar si un string empieza o termina con ciertos caracteres. Podemos utilizar dos metodos para resolver este problema: `.startswith()` y `.endswith()`.

[51]:
```
pais = 'Panamá'
pais.startswith('pa')  # False
```

[51]:
```
False
```

[52]:
```
pais.startswith('Pa')
```

[52]:
```
True
```

[53]:
```
pais.endswith('ma')
```

[53]:
```
False
```

[54]:
```
pais.endswith('má')
```

[54]:
```
True
```

1.3.4 Metodos de Strings e Inmutabilidad

Recordemos que strings son inmutables, toda vez que no pueden ser modificados una vez hayan sido creados. La mayoria de los metodos que modifican un string, como `.upper()` o `.lower()`, en realidad retornan copias del string original con las modificaciones apropiadas.

[55]:
```
nombre = 'Juan'
```

[56]:
```
nombre.upper()
```

[56]:
```
'JUAN'
```

[57]:
```
nombre  # nada cambió
```

[57]:
```
'Juan'
```

[58]:
```

```

```
# puedes descubrir metodos adicionales
nombre.  # espera unos momentos o presiona TAB y saldrá una lista de metodos␣
 ↪disponibles
```

```
  File "<ipython-input-58-a824f0a6344b>", line 2
    nombre.  # espera unos momentos o presiona TAB y saldrá una lista de␣
↪metodos disponibles
                ^
SyntaxError: invalid syntax
```

[ ]:
```
nombre.u  # presiona TAB para que el metodo se auto complete
```

[60]:
```
for metodo in dir(nombre):
    if not metodo.startswith('_'):
        print(metodo)
```

```
capitalize
casefold
center
count
encode
endswith
expandtabs
find
format
format_map
index
isalnum
isalpha
isascii
isdecimal
isdigit
isidentifier
islower
isnumeric
isprintable
isspace
istitle
isupper
join
ljust
lower
lstrip
maketrans
```

```
partition
replace
rfind
rindex
rjust
rpartition
rsplit
rstrip
split
splitlines
startswith
strip
swapcase
title
translate
upper
zfill
```

[61]:
```
# tambien puedes pedir ayuda
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by format_spec.
```

```
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(…)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
```

```
 |  __sizeof__(self, /)
 |      Return the size of the string in memory, in bytes.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  capitalize(self, /)
 |      Return a capitalized version of the string.
 |
 |      More specifically, make the first character have upper case and the rest
lower
 |      case.
 |
 |  casefold(self, /)
 |      Return a version of the string suitable for caseless comparisons.
 |
 |  center(self, width, fillchar=' ', /)
 |      Return a centered string of length width.
 |
 |      Padding is done using the specified fill character (default is a space).
 |
 |  count(…)
 |      S.count(sub[, start[, end]]) -> int
 |
 |      Return the number of non-overlapping occurrences of substring sub in
 |      string S[start:end].  Optional arguments start and end are
 |      interpreted as in slice notation.
 |
 |  encode(self, /, encoding='utf-8', errors='strict')
 |      Encode the string using the codec registered for encoding.
 |
 |      encoding
 |        The encoding in which to encode the string.
 |      errors
 |        The error handling scheme to use for encoding errors.
 |        The default is 'strict' meaning that encoding errors raise a
 |        UnicodeEncodeError.  Other possible values are 'ignore', 'replace' and
 |        'xmlcharrefreplace' as well as any other name registered with
 |        codecs.register_error that can handle UnicodeEncodeErrors.
 |
 |  endswith(…)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      suffix can also be a tuple of strings to try.
 |
```

```
 |  expandtabs(self, /, tabsize=8)
 |      Return a copy where all tab characters are expanded using spaces.
 |
 |      If tabsize is not given, a tab size of 8 characters is assumed.
 |
 |  find(…)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  format(…)
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions from args and
kwargs.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  format_map(…)
 |      S.format_map(mapping) -> str
 |
 |      Return a formatted version of S, using substitutions from mapping.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  index(…)
 |      S.index(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  isalnum(self, /)
 |      Return True if the string is an alpha-numeric string, False otherwise.
 |
 |      A string is alpha-numeric if all characters in the string are alpha-
numeric and
 |      there is at least one character in the string.
 |
 |  isalpha(self, /)
 |      Return True if the string is an alphabetic string, False otherwise.
 |
 |      A string is alphabetic if all characters in the string are alphabetic
and there
```

```
 |      is at least one character in the string.
 |
 |  isascii(self, /)
 |      Return True if all characters in the string are ASCII, False otherwise.
 |
 |      ASCII characters have code points in the range U+0000-U+007F.
 |      Empty string is ASCII too.
 |
 |  isdecimal(self, /)
 |      Return True if the string is a decimal string, False otherwise.
 |
 |      A string is a decimal string if all characters in the string are decimal
and
 |      there is at least one character in the string.
 |
 |  isdigit(self, /)
 |      Return True if the string is a digit string, False otherwise.
 |
 |      A string is a digit string if all characters in the string are digits
and there
 |      is at least one character in the string.
 |
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier, False otherwise.
 |
 |      Call keyword.iskeyword(s) to test whether string s is a reserved
identifier,
 |      such as "def" or "class".
 |
 |  islower(self, /)
 |      Return True if the string is a lowercase string, False otherwise.
 |
 |      A string is lowercase if all cased characters in the string are
lowercase and
 |      there is at least one cased character in the string.
 |
 |  isnumeric(self, /)
 |      Return True if the string is a numeric string, False otherwise.
 |
 |      A string is numeric if all characters in the string are numeric and
there is at
 |      least one character in the string.
 |
 |  isprintable(self, /)
 |      Return True if the string is printable, False otherwise.
 |
 |      A string is printable if all of its characters are considered printable
in
```

```
 |      repr() or if it is empty.
 |
 |  isspace(self, /)
 |      Return True if the string is a whitespace string, False otherwise.
 |
 |      A string is whitespace if all characters in the string are whitespace
and there
 |      is at least one character in the string.
 |
 |  istitle(self, /)
 |      Return True if the string is a title-cased string, False otherwise.
 |
 |      In a title-cased string, upper- and title-case characters may only
 |      follow uncased characters and lowercase characters only cased ones.
 |
 |  isupper(self, /)
 |      Return True if the string is an uppercase string, False otherwise.
 |
 |      A string is uppercase if all cased characters in the string are
uppercase and
 |      there is at least one cased character in the string.
 |
 |  join(self, iterable, /)
 |      Concatenate any number of strings.
 |
 |      The string whose method is called is inserted in between each given
string.
 |      The result is returned as a new string.
 |
 |      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
 |
 |  ljust(self, width, fillchar=' ', /)
 |      Return a left-justified string of length width.
 |
 |      Padding is done using the specified fill character (default is a space).
 |
 |  lower(self, /)
 |      Return a copy of the string converted to lowercase.
 |
 |  lstrip(self, chars=None, /)
 |      Return a copy of the string with leading whitespace removed.
 |
 |      If chars is given and not None, remove characters in chars instead.
 |
 |  partition(self, sep, /)
 |      Partition the string into three parts using the given separator.
 |
 |      This will search for the separator in the string.  If the separator is
```

```
found,
 |      returns a 3-tuple containing the part before the separator, the
separator
 |      itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containing the original
string
 |      and two empty strings.
 |
 |  replace(self, old, new, count=-1, /)
 |      Return a copy with all occurrences of substring old replaced by new.
 |
 |        count
 |          Maximum number of occurrences to replace.
 |          -1 (the default value) means replace all occurrences.
 |
 |      If the optional argument count is given, only the first count
occurrences are
 |      replaced.
 |
 |  rfind(…)
 |      S.rfind(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  rindex(…)
 |      S.rindex(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  rjust(self, width, fillchar=' ', /)
 |      Return a right-justified string of length width.
 |
 |      Padding is done using the specified fill character (default is a space).
 |
 |  rpartition(self, sep, /)
 |      Partition the string into three parts using the given separator.
 |
 |      This will search for the separator in the string, starting at the end.
If
```

```
 |      the separator is found, returns a 3-tuple containing the part before the
 |      separator, the separator itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containing two empty
strings
 |      and the original string.
 |
 |  rsplit(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the delimiter
string.
 |
 |        sep
 |          The delimiter according which to split the string.
 |          None (the default value) means split according to any whitespace,
 |          and discard empty strings from the result.
 |        maxsplit
 |          Maximum number of splits to do.
 |          -1 (the default value) means no limit.
 |
 |      Splits are done starting at the end of the string and working to the
front.
 |
 |  rstrip(self, chars=None, /)
 |      Return a copy of the string with trailing whitespace removed.
 |
 |      If chars is given and not None, remove characters in chars instead.
 |
 |  split(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the delimiter
string.
 |
 |      sep
 |        The delimiter according which to split the string.
 |        None (the default value) means split according to any whitespace,
 |        and discard empty strings from the result.
 |      maxsplit
 |        Maximum number of splits to do.
 |        -1 (the default value) means no limit.
 |
 |  splitlines(self, /, keepends=False)
 |      Return a list of the lines in the string, breaking at line boundaries.
 |
 |      Line breaks are not included in the resulting list unless keepends is
given and
 |      true.
 |
 |  startswith(…)
 |      S.startswith(prefix[, start[, end]]) -> bool
```

```
 |
 |      Return True if S starts with the specified prefix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      prefix can also be a tuple of strings to try.
 |
 |  strip(self, chars=None, /)
 |      Return a copy of the string with leading and trailing whitespace
removed.
 |
 |      If chars is given and not None, remove characters in chars instead.
 |
 |  swapcase(self, /)
 |      Convert uppercase characters to lowercase and lowercase characters to
uppercase.
 |
 |  title(self, /)
 |      Return a version of the string where each word is titlecased.
 |
 |      More specifically, words start with uppercased characters and all
remaining
 |      cased characters have lower case.
 |
 |  translate(self, table, /)
 |      Replace each character in the string using the given translation table.
 |
 |        table
 |          Translation table, which must be a mapping of Unicode ordinals to
 |          Unicode ordinals, strings, or None.
 |
 |      The table must implement lookup/indexing via __getitem__, for instance a
 |      dictionary or list.  If this operation raises LookupError, the character
is
 |      left untouched.  Characters mapped to None are deleted.
 |
 |  upper(self, /)
 |      Return a copy of the string converted to uppercase.
 |
 |  zfill(self, width, /)
 |      Pad a numeric string with zeros on the left, to fill a field of the
given width.
 |
 |      The string is never truncated.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
```

```
|      Create and return a new object.  See help(type) for accurate signature.
|
|  maketrans(…)
|      Return a translation table usable for str.translate().
|
|      If there is only one argument, it must be a dictionary mapping Unicode
|      ordinals (integers) or characters to Unicode ordinals, strings or None.
|      Character keys will be then converted to ordinals.
|      If there are two arguments, they must be strings of equal length, and
|      in the resulting dictionary, each character in x will be mapped to the
|      character at the same position in y. If there is a third argument, it
|      must be a string, whose characters will be mapped to None in the result.
```

Ejercicios: 1. Escribe un script the convierte algunas palabras a letra minuscula 2. Repite el ejercicio anterior pero a letras mayusculas. 3. Escribe un script que elimina el espacio a la izquierda de una palabra 4. Escribe un script que elimina el espacio a la derecha de una palabra 5. Escribe un script que elimina el espacio en ambos lados de la palabra 6. Escribe un script que diga si una palabra empieza con un determinado string

1.4 Interactuando con Dato de Entrada del Usuario

En esta seccion vamos a aprender como obtener datos de entrada del usuario con la funcion incorporada `input()`. Vamos a escribir un programa que pregunta por dato de entrada de usuario y luego imprime el texto en mayuscula.

[62]:
```
texto = 'Hola, como estas?\n'
dato_de_usuario = input(texto)
print('Dijiste:', dato_de_usuario)
```

```
Hola, como estas?
Bien
Dijiste: Bien
```

[63]:
```
# Combinemos el metodo input() con el metodo de string upper()
respuesta = input('Que deberia gritar?\n')
respuesta = respuesta.upper()
print('Bueno, si insistes...', respuesta)
```

```
Que deberia gritar?
Nada
Bueno, si insistes… NADA
```

1.5 Ejercicios

1. Toma datos de entrada del usuario e imprimelo
2. Toma datos de entrada del usuario y retorna una version del string con todas las letras mayusculas.

3. Toma datos de entrada del usuario y retorna una version del string con la primera letra mayuscula.

1.6 Reto: Manipulacion del Dato de Entrada del Usuario

Escribe un script llamado `primera_letra.py` que le pide al usuario su contraseña. El programa deberia: * convertir la primera letra del dato de entrada en mayuscula * imprimir "La primera letra que ingresaste fue:" seguido por la letra

1.7 Trabajando con Strings y Numeros

Cuando obtenemos datos de entrada del usuario utilizando la funcion `input()`, el resultado siempre es un string. Sin embargo, existen otras veces cuando el dato de entrada contiene numeros en los cuales se le debe hacer algunas operaciones / calculaciones.

En esta seccion aprenderemos a trabajar con strings de numeros. Vamos a ver como trabajan los operadores aritmeticos en strings y como arribamos a unos sorprendentes resultados. Tambien aprenderemos como convertir entre strings / cadenas y numeros.

1.7.1 Strings y Operadores Aritmeticos

Los strings pueden tener una gran variedad de caracteres, incluyendo numeros.

Sin embargo, no confundamos numeros en un string con numeros actuales.

[64]:
```
num = '2'
```

[65]:
```
num + num  # cual será el resultado de esta operación?
```

[65]:
```
'22'
```

El operador aritmetico hace una concatenación de ambos strings. Asimismo, strings pueden ser *multiplicados* por un numero siempre y cuando ese numero sea un numero entero.

[66]:
```
num * 3  # cual será el resultado?
```

[66]:
```
'222'
```

[67]:
```
'2' * '3'  # cual será el resultado?
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-67-ae501cc910b0> in <module>
----> 1 '2' * '3'  # cual será el resultado?
```

```
TypeError: can't multiply sequence by non-int of type 'str'
```

Python eleva un error de `TypeError` toda vez que no se puede multiplicar una secuencia por algo que no sea un numero entero de tipo integer o `int`.

Una secuencia es un objeto que permite acceder sus elementos por indice. Por tanto, strings son secuencias. Mas adelante veremos otro tipo de secuencias.

[68]:
```
type(3)
```

[68]:
```
int
```

[69]:
```
# que sucede si agregamos un string con un numero?
'3' + 3
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-69-b91289f03597> in <module>
      1 # que sucede si agregamos un string con un numero?
----> 2 '3' + 3


TypeError: can only concatenate str (not "int") to str
```

Python eleva `TypeError` porque el operador `+` exige que ambos lados de la operación sean de tipo string.

1.7.2 Convirtiendo Strings a Numeros

[70]:
```
num = input('Ingrese un numero para que sea doblado: ')
```

```
Ingrese un numero para que sea doblado: 2
```

[71]:
```
numero_doblado = num * 2
print(numero_doblado)  # cual será el resultado?
```

```
22
```

Para poder realizar operaciones aritmeticas en numeros que son strings, primero debemos convertirlos al tipo de dato indicado: numero. Existen dos maneras de lograrlo: `int()` y `float()`.

- `int()` representa **integer** o numero entero

- `float()` representa **floating-point-number** o numero de **punto flotante** (con puntos decimales), tambien conocido como numero **real**.

[72]:
```
int('31')
```

[72]:
```
31
```

[73]:
```
float('31')  # Python le agrega un punto cero
```

[73]:
```
31.0
```

Los numeros de punto flotante siempre tienen por lo menos un punto decimal de precision. Por esta razón, no podemos convertir un string que se ve como un numero de punto flotante a un numero entero de tipo `int` porque perderíamos todo despues del punto decimal.

[74]:
```
int('31.0')
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-74-93693ac29684> in <module>
----> 1 int('31.0')


ValueError: invalid literal for int() with base 10: '31.0'
```

[75]:
```
# volvamos a nuestro programa inicial
num = input('Ingrese un numero para que sea doblado: ')
numero_doblado = num * 2
print(numero_doblado)
```

```
Ingrese un numero para que sea doblado: 2
22
```

[76]:
```
# como podemos arreglar el problema?
num = input('Ingrese un numero para que sea doblado: ')
numero_doblado = num * 2  # esta linea debe ser modificada
print(numero_doblado)
```

```
Ingrese un numero para que sea doblado: 4
44
```

1.7.3 Transformando numeros a strings

A veces debemos convertir un numero a un string.

[77]:
```
num_tortillas = 5
'Me voy a comer ' + num_tortillas + 'tortillas!'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-77-1d6616b11f80> in <module>
  1 num_tortillas = 5
----> 2 'Me voy a comer ' + num_tortillas + 'tortillas!'


TypeError: can only concatenate str (not "int") to str
```

[78]:
```
# tenemos que convertir el numero a un string utilizando str()
num_tortillas = 5
'Me voy a comer ' + str(num_tortillas) + ' tortillas!'
```

[78]:
```
'Me voy a comer 5 tortillas!'
```

[79]:
```
# tambien podemos realizar operaciones aritmeticas adentro de str()
num_tortillas = 5
tortillas_comidas = 3
'Nada mas quedan ' + str(num_tortillas - tortillas_comidas) + ' tortillas!'
```

[79]:
```
'Nada mas quedan 2 tortillas!'
```

[80]:
```
# podemos pasarle todo tipo de objetos a str()
str(print)
```

[80]:
```
'<built-in function print>'
```

[81]:
```
str(int)
```

[81]:
```
"<class 'int'>"
```

[82]:
```
str(float)
```

[82]:
```
"<class 'float'>"
```

Ejercicios: 1. Crea un string que contiene un numero, luego convierte ese numero a un `int()`. Comprueba que el numero es realmente un numero multiplicandolo por otro numero. 2. Repite el ejercicio anterior utilizando un numero de punto flotante `float()` 3. Crear un string y un integer, luego imprimeles con `print()` utilizando le funcion `str()` 4. Escribe un script que obtiene 2 numeros del usuario utilizando `input()` 2 veces, multiplica los numeros e imprime el resultado.

1.8 Agilicemos las declaraciones `print()`

Supongamos que tenemos un string con `nombre = 'Monstro'` y dos integers `cabezas = 2`, `brazos = 3`. Queremos que se impriman en la siguiente linea `Monstro tiene 2 cabezas y 3 brazos`. A esto lo llamamos **string interpolation** o **interpolacion de cadena**

Ya hemos visto dos formas de hacerlo: 1. Utilizando comas para separar cada string en un `print()` 2. Utilizando el operador `+` para concatenar strings

[83]:
```
nombre = 'Monstro'
cabezas = '2'
brazos = '3'
```

[84]:
```
print(nombre, 'tiene', str(cabezas), 'cabezas y', str(brazos), 'brazos')
```

```
Monstro tiene 2 cabezas y 3 brazos
```

[85]:
```
print(nombre + ' tiene ' + str(cabezas) + ' cabezas y ' + str(brazos) + '␣
 ↪brazos')
```

```
Monstro tiene 2 cabezas y 3 brazos
```

Ambas tecnicas producen codigo que es dificil de leer, toda vez que el lector debe acordarse que va adentro o afuera de comillas. Afortunadamente, existe una tercer forma de combinar strings: **formatted string literals** o **literales de cadena con formato**, conocidos como **f-strings**.

[86]:
```
f'{nombre} tiene {cabezas} cabezas y {brazos} brazos'
```

[86]:
```
'Monstro tiene 2 cabezas y 3 brazos'
```

[87]:
```
print(f'{nombre} tiene {cabezas} cabezas y {brazos} brazos')
```

```
Monstro tiene 2 cabezas y 3 brazos
```

Importante destacar dos cosas: * El literal de cadena de caracteres empieza con la letra `f` antes de la apertura de citas * Los nombres de variables estan rodeados por llaves `{}` y son remplazados por su valor correspondiente sin la necesidad de utilizar `str()`

[88]:
```
# tambien podemos insertar expresiones entre las llaves
x = 3
y = 4
f'{x} por {y} es igual a {x * y}'
```

[88]:
```
'3 por 4 es igual a 12'
```

Los **f-strings** pueden ser utilizados desde la version 3.6 de Python. En versiones anteriores, se utilizaba el metodo `.format()` para obtener los mismos resultados.

[89]:
```
'{} tiene {} cabezas y {} brazos'.format(nombre, cabezas, brazos)
```

[89]:
```
'Monstro tiene 2 cabezas y 3 brazos'
```

1.8.1 Ejercicios

1. Crea un objeto de punto flotante `float` llamado `peso` con un valor de `3.9` y luego crea un objeto de tipo `str` llamado `animal` con un valor `gato`. Utiliza estos objetos para imprimir el siguiente string, utilizando concatenacion: `3.9kg es el peso de un gato dómestico`
2. Imprime el mismo string utilizando el metodo `.format()`
3. Imprime el mismo string utilizando **f-strings**

1.9 Encuentra un string dentro de un string

Uno de los metodos mas utiles es el metodo `.find()`. Su nombre se traduce a *encontrar*, y se utiliza para encontrar la ubicacion de un string dentro de otro string, lo que comunmente se conoce como **substring** o **subcadena**.

Para utilizar `.find()`, agregalos al final de una variable o literal de cadena y pasale el string que quieres encontrar.

[90]:
```
frase = 'la sorpresa esta aqui en alguna parte'
```

[91]:
```
frase.find('sorpresa')
```

[91]:
```
3
```

El valor de `.find()` retorna el indice de la primera ocurrencia del string que ingresamos entre paréntesis. En este caso, `'sorpresa'` empieza en el caracter numero 5 de la frase, que tiene un indice de 4 porque empezamos a contar en `0`.

[92]:
```
# Si .find() no encuentra el substring, retorna -1
frase.find('comida')
```

[92]:
```
-1
```

[93]:
```
# tambien podemos utilizar find() directamente en el string
'la sorpresa esta por aqui en alguna parte'.find('sorpresa')
```

[93]:
```
3
```

[94]:
```
# find() es sensible a mayuscula / minuscula
frase.find('SORPRESA')
```

[94]:
```
-1
```

[95]:
```
# si el substring aparece 2 veces, find() retorna el indice de la primera␣
↪ocurrencia
'Tres tristes tigres siguen siendo tigres'.find('tigres')
```

[95]:
```
13
```

[96]:
```
# find() solo acepta strings como argumento / parametro
'Mi numero es el 123-456'.find(1)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-96-b7d9e55bfaaa> in <module>
  1 # find() solo acepta strings como argumento / parametro
----> 2 'Mi numero es el 123-456'.find(1)


TypeError: must be str, not int
```

[97]:
```
'Mi numero es el 123-456'.find('1')
```

[97]:
```
16
```

A veces necesitamos encontrar todas las ocurrencias de una subcadena en particular y reemplazarlo por otra cadena. Toda vez que `.find()` solo retorna el indice de la primera ocurrencia de un substring, no podemos *facilmente* utilizarlo para esta operación. Afortunadamente, existe un metodo llamado `.replace()` que reemplaza cada ocurrencia de un substring. La traduccion de `.replace()` es *reemplaza*.

Al igual de `.find()`, agregamos `.replace()` al final de una variable o literal de cadena. En este caso, `.replace()` acepta 2 argumentos / parametros entre parentesis: 1. El substring a reemplazar 2. El substring con que se efectua el reemplazo

[98]:
```
frase = 'Hola mundo'
frase.replace('mundo', 'Chiriquí')
```

[98]:
```
'Hola Chiriquí'
```

[99]:
```
# acordemonos que strings son inmutables
frase
```

[99]:
```
'Hola mundo'
```

[100]:
```
# si deseamos que se altere el contenido, habra que guardarlo nuevamente
frase = frase.replace('mundo', 'Chiriquí')
frase
```

[100]:
```
'Hola Chiriquí'
```

1.9.1 Ejercicios

1. En una linea de codigo, imprime el resultado de `.find()` el substring `'a'` en `'AAA'`.
2. Reempleza todas las ocurrencias de `'t'` con `'d'`
3. Escribe un programa que acepta dato de usuario con `input()` e imprime el resultado de `.find()` sobre una letra en particular.

1.10 Reto: Multiples Reemplazos

Escribe un programa llamado `translate.py` que: * obtiene dato de entrada del usuario utilizando el siguiente mensaje: `'Escribe algo de texto:` * utiliza el metodo `.replace()` para convertir el texto ingresado por el usuario de la siguiente manera: * La letra `a` se convierte en `4` * La letra `e` se convierte en `3` * La letra `i` se convierte en `1` * La letra `o` se convierte en `0` * La letra `r` se convierte en `2` * La letra `s` se convierte en `5` * el programa debe imprimir el resultado del string como dato de salida.

Ejemplo:

```
Escribe algo: Tres tristes tigres
T235 t215t35 t1g235
```

1.11 Resumen

En esta seccion aprendimos sobre Python strings. Aprendimos como acceder distintos caracteres en un string utilizando indexacion y segmentacion, asi como determinar la longitud de un string con `len()`.

Los strings tiene una gran variedad de metodos. Los metodos `.upper()` y `.loer()` convierten todos los caracteres de un string a mayuscula o minuscula. Los metodos `.rstrip()`, `.lstrip()` y `.strip()` eliminan espacion en un string, y los metodos `.startswith()` y `.endswith()` nos informan si un string empieza o termina con determinado substring.

Ademas, vimos como capturar dato de entrada del usuario como un string utilizando la funcion `input()`, y como concertimos ese dato de entrada a un numero utilizando `int()` y `float()`. Para convertir numeros y otros objetos a strings, vimos como utilizar `str()`.

Finalmente, vimos como `.find()` y `.replace()` son utiles para encontrar la ubicacion de un substring y remplazar un substring con un nuevo string.