

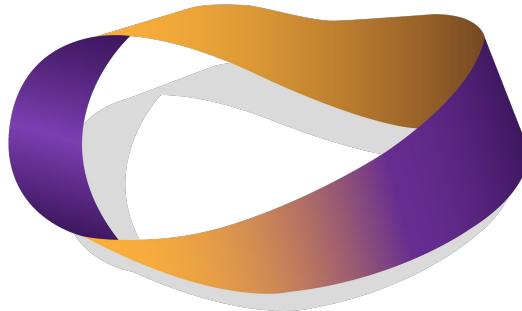
Estructuras de datos en Python



ACTUMLOGOS

DESARROLLANDO HABILIDADES TECNOLÓGICAS

Diccionarios y conjuntos



ACTUMLOGOS

DESARROLLANDO HABILIDADES TECNOLÓGICAS

Diccionarios

Un diccionario asocia *claves* con valores, de tal forma que a cada clave se asigna a un valor específico. Otro nombre con que se les conoce a los diccionarios es el de *matrices asociativas*, ya que son colecciones que relacionan una clave y un valor.

El uso de un nombre facilita el acceso a valores particulares con algo que no sea un índice numérico. Para crear un diccionario, encierra pares de nombre y valor entre llaves.

Escribe la siguiente celda

```
[1] # Ejemplo de diccionario
peliculas = {'Hellboy': 'Guillermo del Toro', 'Roma': 'Alfonso Cuarón'}
codigos = {'México': 52, 'Brasil': 55, 'Francia': 33}
print(peliculas)
print(codigos)
for pais,cod in codigos.items():
    print("%s -> %d" % (pais,cod))
```

La forma básica para acceder a los valores de un diccionario, es mediante la clave del elemento, por ejemplo:

```
[2] print('El director de Hellboy es: ' + peliculas['Hellboy'])
```

```
↳ El director de Hellboy es: Guillermo del Toro
```

La gran ventaja de utilizar diccionarios como estructuras de datos, es que facilita el acceso a conjuntos de datos complejos utilizando términos que todos pueden entender (las claves).

En un diccionario se pueden también modificar el contenido asociado a una clave, añadir un nuevo par *clave:valor* o quitar uno ya existente.

Escribe lo siguiente:

```
[1] alumnos = {'Jesús': 10, 'Manuel': 7, 'Raúl': 8, 'Tania': 9}
    print(alumnos)
    alumnos['Jesús'] = 9
    print(alumnos)
    alumnos['Lili'] = 5
    print(alumnos)
    del alumnos['Manuel']
    print(alumnos)
```

Reto 1: En 10 minutos completa la celda 1, donde se crea una función de suma de la población de dos estados de la república que reciba como parámetros (en forma de lista). Los datos de los estados y sus poblaciones deben estar almacenados en un diccionario. En una segunda celda, solicita al usuario los dos estados y manda a llamar la función, de la primer celda.

Tips:

- Utilice un ciclo for para recorrer el diccionario.
- Utilice el método items() y el operador or
- Consulte en “San Google”

5.12 millones (2015) Nuevo León	8.113 millones (2015) Veracruz
5.218 millones (2015) Chiapas	8.256 millones (2018) Jalisco
5.854 millones (2015) Guanajuato	16.19 millones (2015) Estado de México
6.169 millones (2015) Puebla	

Resultado esperado:

Dame el primer estado: Jalisco

Dame el segundo estado: Puebla

La sumas es: 14.425 millones

Solución 1:

```
[1] def suma_poblacion(estados):
    suma = 0
    poblacion = {'Nuevo León': 5.12, 'Chiapas': 5.218,
                  'Guanajuato': 5.854, 'Puebla': 6.169,
                  'Veracruz': 8.113, 'Jalisco': 8.256,
                  'EdoMex': 16.19}
    for est,pob in poblacion.items():
        if estados[0] == est or estados[1] == est:
            suma += pob
    return suma
```

```
[2] estados = []
    estados.append(input('Dame el primer estado: '))
    estados.append(input('Dame el segundo estado: '))
    print('La sumas es: ' + str(suma_poblacion(estados)) + ' millones')
```

Solución 2:

```
[1] def suma_poblacion(estados):
    suma = 0
    poblacion = {'Nuevo León': 5.12, 'Chiapas': 5.218,
                  'Guanajuato': 5.854, 'Puebla': 6.169,
                  'Veracruz': 8.113, 'Jalisco': 8.256,
                  'EdoMex': 16.19}
    suma = poblacion[estados[0]] + poblacion[estados[1]]
    return suma
```

```
[2] estados = []
estados.append(input('Dame el primer estado: '))
estados.append(input('Dame el segundo estado: '))
print('La sumas es: ' + str(suma_poblacion(estados)) + ' millones')
```

Así como se utilizó el método *items()* para iterar a través de los pares *clave:valor* de un diccionario, existen un par de métodos llamados *keys()* y *values()* que se utilizan para recorrer solo las claves o los valores de un diccionario

Escribe lo siguiente:

```
[1] poblacion = {'Nuevo León': 5.12, 'Chiapas': 5.218,  
                 'Guanajuato': 5.854, 'Puebla': 6.169,  
                 'Veracruz': 8.113, 'Jalisco': 8.256,  
                 'EdoMex': 16.19}
```

```
[2] for estado in poblacion.keys():  
    print(estado, end = ' ')
```

```
[3] for poblacion in poblacion.values():  
    print(poblacion, end = ' ')
```

En ocasiones es necesario almacenar los datos de un diccionario en listas, lo cual se puede hacer con la función `list()`, auxiliándose de los métodos `items()`, `keys()` y `values()`

Escribe lo siguiente:

```
[1] poblacion = {'Nuevo León': 5.12, 'Chiapas': 5.218,  
                 'Guanajuato': 5.854, 'Puebla': 6.169,  
                 'Veracruz': 8.113, 'Jalisco': 8.256,  
                 'EdoMex': 16.19}
```

```
[2] list(poblacion.items())
```

```
[3] list(poblacion.keys())
```

```
[4] list(poblacion.values())
```

La comparación entre diccionarios se puede llevar a cabo con los operadores == y != . Esta operación nos permitir saber si dos diccionarios tienen información idéntica, sin importar el orden en que fueron añadidos al diccionario.

```
[1] capitales1 = {'Brasil': 'Brasilia', 'Francia': 'París', 'México': 'CdMx'}
capitales2 = {'Brasil': 'Brasilia', 'EUA': 'Washington', 'México': 'CdMx'}
capitales3 = {'México': 'CdMx', 'Brasil': 'Brasilia', 'Francia': 'París'}
print(capitales1 == capitales2)
print(capitales1 == capitales3)
print(capitales2 != capitales3)
```

Reto 2: En 10 minutos, realiza un programa que en un diccionario almacene las calificaciones de un grupo de alumnos y que con esa información calcule el promedio de cada uno.

Tips:

- Utilice un ciclo for para recorrer el diccionario.
- Importe el modulo statistics
- Utilice el método append()
- Investigue lo que son las *cadenas f* para imprimir los resultados
- Consulte en “San Google”

	Cálculo	Física	Historia	Química
Martín	10	8	9	7
Armando	6	6	7	9
Ernesto	10	5	8	7
Ignacio	7	7	8	9

Resultado esperado:

El alumno Martín obtuvo 8.5 de promedio
El alumno Armando obtuvo 7 de promedio
El alumno Ernesto obtuvo 7.5 de promedio
El alumno Ignacio obtuvo 7.75 de promedio
El promedio del grupo es 7.6875

Solución:

```
[1] import statistics as st
    lista_promedios = []
    alumnos = {'Martín': [10, 8, 9, 7], 'Armando': [6, 6, 7, 9],
               'Ernesto': [10, 5, 8, 7], 'Ignacio': [7, 7, 8, 9]}
    for nombre, calif in alumnos.items():
        prom = st.mean(calif)
        print(f'El alumno {nombre} obtuvo {prom} de promedio')
        lista_promedios.append(prom)
    print(f'El promedio del grupo es {st.mean(lista_promedios)}')
```

Caso práctico

Las técnicas como el conteo de frecuencia de palabras, se usan a menudo para analizar trabajos publicados. Por ejemplo, algunas personas creen que las obras de William Shakespeare en realidad podrían haber sido escritas por Sir Francis Bacon, Christopher Marlowe u otros. La comparación de las frecuencias de palabras de sus obras con las de Shakespeare, puede revelar similitudes de estilo de escritura.



[Sir Francis Bacon IS Shakespeare!](#)

Reto 3: En 5 minutos complete la celda del reto 3. Lo que se busca es contar cuantas veces aparece una palabra en un par de cadenas de texto.

Tips:

- Utilice los métodos `split()` e `item()`
- Consulte en “San Google”

Resultado esperado:

Palabra	Conteo
Este	2
algunas	1
con	1
diferentes	1
es	1
otro	1
palabras	2
texto	2
tiene	1
unas	1

Número de palabras únicas: 10

Solución:

```
[1] '''Programa para el conteo de frecuencias de palabras
utilizando diccionarios'''

texto = ('Este texto tiene algunas palabras '
         'Este es otro texto con unas palabras diferentes')

cont_pal = {}

# cuenta las veces que aparecen las palabras en los textos
for palabra in texto.split():
    if palabra in cont_pal:
        cont_pal[palabra] += 1 # Actualización de un par clave:valor
    else:
        cont_pal[palabra] = 1 # Nuevo par clave:valor

print(f'{"Palabra":<12}Conteo')

for palabra, contador in sorted(cont_pal.items()):
    print(f'{palabra:<12}{contador}')
print('\nNúmero de palabras únicas:', len(cont_pal))
```

Biblioteca estándar de Python: Módulo *collections*

Dado que la operación de conteo de ocurrencias es una operación bastante empleada, la biblioteca estándar de Python ya contiene una función de conteo que funciona de forma similar a la que acabamos de implementar. Para ello se ocupa una subclase llamada *Counter*, que recibe un iterable y genera un diccionario donde los valores son el número de ocurrencias.

Escribe el siguiente código:

```
[1] from collections import Counter
      texto = ('Este texto tiene algunas palabras '
              'Este es otro texto con unas palabras diferentes')

      contador = Counter(texto.split())
      for palabra, cont in sorted(contador.items()):
          print(f'{palabra}:{cont}')

      print('\nNúmero de palabras únicas:', len(contador.keys()))
```

Metodo *Update*

Otra forma para poder añadir o actualizar los pares clave:valor de un diccionario es por medio del método *Update()*

Escribe el siguiente código

```
[1] # diccionario vacio
    dir_peliculas = {}
    print(dir_peliculas)
    # se añaden elementos al diccionario
    dir_peliculas.update({'Cuarón':'Roma'})
    print(dir_peliculas)
    dir_peliculas.update(Spielberg = 'E.T.')
    print(dir_peliculas)
```

Reto 4: En 10 minutos realice un programa que pida los datos nombre y edad de n compañeros del grupo y los guarde en un diccionario. La cantidad n la solicita al usuario y en caso de ser una cantidad no válida (alumnos ≤ 0), manda un mensaje que lo indique.

Tips:

- Utilice los métodos *update()*
- Consulte en “San Google”

Resultado esperado:

```
¿Cuantos alumnos se inscriben? 3
Dame el nombre: Manuel
Dame la edad: 24
Dame el nombre: Ana
Dame la edad: 21
Dame el nombre: Hugo
Dame la edad: 29
```

Solución:

```
[1] # Programa de inscripción de alumnos
dic_alumnos = {}
num_alumnos = input('¿Cuantos alumnos se inscriben? ')
# El if determina una cantidad invalida de alumnos
if(int(num_alumnos) > 0):
    for i in range(int(num_alumnos)):
        nombre = input('Dame el nombre: ')
        edad = int(input('Dame la edad: '))
        dic_alumnos.update({nombre:edad})
else:
    print('El número de alumnos no es válido')
print(dic_alumnos) # Se muestra el contenido de dict
```

Conjuntos

Un conjunto es una colección desordenada de valores únicos. Los conjuntos pueden contener solo objetos inmutables, como cadenas, ints, flotantes y tuplas que contienen sólo elementos inmutables. Aunque los conjuntos son iterables, no son secuencias y no admiten la indexación. Los principales usos de los conjuntos son la verificación de pertenencia y eliminación de entradas duplicadas.

Escribe lo siguiente:

```
[1] paises = {'México', 'EUA', 'Japón', 'México', 'Rusia'}  
      print(paises)
```

Iteraciones sobre conjuntos

Los conjuntos son iterables, por lo que puede procesar cada elemento del conjunto con un bucle `for`

Escriba la siguiente celda:

```
[2] for pais in paises:  
    print(pais.upper(), end=' ')
```

Los conjuntos no están ordenados, por lo que no tiene importancia el orden de iteración.

Función Set

Puede crear un conjunto a partir de otra colección de valores utilizando la función set() incorporada en la biblioteca estándar.

Escribe la siguiente celda

```
[1] numeros = list(range(10)) + list(range(5))
    print(numeros)
    mi_conjunto = set(numeros)
    print(mi_conjunto)
```

Como ya lo vimos, en la creación de un conjunto, no se consideran los valores repetidos.

Frozense: un tipo de conjunto inmutable

Los conjuntos son mutables, es decir se pueden agregar y eliminar elementos, pero los elementos del conjunto deben ser inmutables. Por lo tanto, un conjunto no puede tener otros conjuntos como elementos.

Un *frozense* (conjunto congelado) es un conjunto inmutable; no se puede modificar después de crearlo, por lo que un conjunto puede contener *frozense's* como elementos. La función incorporada *frozense()* crea un conjunto congelado a partir de cualquier iterable.

Reto 5: En 5 minutos haga un programa que cree un conjunto y un conjunto congelado. Inicialmente ambos conjuntos guardan cinco nombres de compañeros del grupo. Compruebe que, efectivamente, el conjunto y el conjunto congelado son mutable e inmutable respectivamente.

Tips:

- Utilice el métodos `add()`
- Consulte en “San Google”

Resultado esperado:

```
El contenido del conjunto es
{'Rodrigo', 'Joel', 'Karla', 'Jesús', 'Jimena', 'Manuel'}
```

```
AttributeError                                     Traceback (most recent call last)
<ipython-input-12-d5d13f561dba> in <module>()
      4 print('El contenido del conjunto es ')
      5 print(nombres1)
----> 6 nombres2.add('Rodrigo')
      7 print('El contenido del conjunto es ')
      8 print(nombres2)

AttributeError: 'frozenset' object has no attribute 'add'
```

Solución:

```
[1] nombres1 = set(('Jesús', 'Karla', 'Manuel', 'Joel', 'Jimena'))
    nombres2 = frozenset(('Jesús', 'Karla', 'Manuel', 'Joel', 'Jimena'))
    nombres1.add('Rodrigo')
    print('El contenido del conjunto es ')
    print(nombres1)
    nombres2.add('Rodrigo')
    print('El contenido del conjunto es ')
    print(nombres2)
```

Comparando conjuntos

Al igual que con los diccionarios, se pueden usar los operadores `==` y `!=`. Esta operación nos permite saber si dos conjuntos tienen información idéntica, sin importar el orden en que se añadan.

Escribe la siguiente celda

```
[1] print('¿Los conjuntos son iguales?')
    print({1, 2, 5} == {2, 1, 5})
    print('¿Los conjuntos son diferentes?')
    print({1, 2, 5} != {2, 1, 5})
```

El operador < comprueba si el conjunto a su izquierda es un subconjunto propio del que está a su derecha, es decir, todos los elementos en el operando izquierdo están en el operando derecho y los conjuntos no son iguales.

El operador <= prueba si el conjunto a su izquierda es un subconjunto impropio del que está a su derecha, es decir, todos los elementos en el operando izquierdo están en el operando derecho, y los conjuntos pueden ser iguales.

```
[1] # ¿Es {3, 5, 1} subconjunto propio de {1, 3, 5}?  
print({1, 3, 5} < {3, 5, 1})  
# ¿Es {1, 3, 5} subconjunto propio de {7, 3, 5, 1}  
print({1, 3, 5} < {7, 3, 5, 1})  
# ¿Es {1, 3, 5} subconjunto impropio de {3, 5, 1}  
print({1, 3, 5} <= {3, 5, 1})  
# ¿Es {1, 3} subconjunto impropio de {3, 5, 1}  
print({1, 3} <= {3, 5, 1})
```

El operador `>` prueba si el conjunto a su izquierda es un superconjunto propio del que está a su derecha, es decir, todos los elementos en el operando derecho están en el operando izquierdo, y el operando izquierdo tiene más elementos.

El operador `>=` prueba si el conjunto a su izquierda es un superconjunto impropio del que está a su derecha, es decir, todos los elementos en el operando derecho están en el operando izquierdo, y los conjuntos pueden ser iguales:

```
[1] # ¿Es {1, 3, 5} superconjunto propio de {3, 5, 1}?
    print({1, 3, 5} > {3, 5, 1})
# ¿Es {1, 3, 5} superconjunto propio de {3, 5, 1}?
print({1, 3, 5, 7} > {3, 5, 1})
# ¿Es {1, 3, 5} superconjunto impropio de {3, 5, 1}?
print({1, 3, 5} >= {3, 5, 1})
# ¿Es {1, 3, 5} superconjunto impropio de {3, 1}?
print({1, 3, 5} >= {3, 1})
# ¿Es {1, 3} superconjunto impropio de {3, 1, 7}?
print({1, 3} >= {3, 1, 7})
```

Operaciones matemáticas con conjuntos

Algunas de las propiedades más interesantes de los conjuntos radican en sus operaciones principales: *unión, intersección y diferencia*.

La **unión** genera un conjunto que consta de todos los elementos únicos de ambos conjuntos. En Python, la operación de unión se lleva a cabo con el operador `|` o con el método `union()`.

```
[1] print({1, 3, 5} | {2, 3, 4})  
      print({1, 3, 5}.union([2, 3, 4]))
```

La **intersección** de dos conjuntos genera un conjunto que consta de todos los elementos únicos que los dos conjuntos tienen en común. En Python, la operación de intersección se lleva a cabo con el operador **&** o con el método *intersection()*.

```
[1] print({1, 3, 5} & {2, 3, 4})  
     print({1, 3, 5}.intersection([2, 3, 4]))
```

La **diferencia** entre dos conjuntos genera un conjunto que consta de los elementos en el operando izquierdo que no están en el operando derecho. Puede calcular la diferencia con el operador **-** o con el método *difference()*.

```
[1] print({1, 3, 5} - {2, 3, 4})  
     print({1, 3, 5}.difference([2, 3, 4]))
```

Métodos para añadir y remover elementos en un conjunto

Al igual que con las listas, existen métodos que van a permitir adicionar o quitar elementos de los conjuntos. Esos son los métodos *add()*, *remove()*, *discard()* y *clear()*.

El método *add()* permite insertar un elemento si aún no está en el conjunto; de lo contrario, el conjunto permanece sin cambios.

```
[1] numeros = {1, 2, 3}
     print(numeros)
     numeros.add(17)
     print(numeros)
```

Por otra parte el método *remove()* elimina un elemento del conjunto, si el elemento no está en el conjunto, se produce un error.

```
[1] numeros = {1, 2, 3}
    print(numeros)
    numeros.remove(1)
    print(numeros)
```

El método *discard()* también elimina su argumento del conjunto, pero no causa una excepción si el valor no está en el conjunto.

```
[1] numeros = {1, 2, 3}
    print(numeros)
    numeros.discard(2)
    print(numeros)
```

Finalmente, el método *clear()* vacía el conjunto en el que se llama:

```
[1] numeros = {1, 2, 3}
    print(numeros)
    numeros.clear()
    print(numeros)
```

Tabla comparativa entre estructuras de datos

Estructura de datos	Símbolos para construirlo	¿es mutable?	¿están ordenados?	¿están indexados?	¿son iterables?
list	[]	sí	sí	sí	sí
tupla	()	no	sí	sí	sí
dict	{ 'key':value}	sí	no	sí, por key	sí
set	{ }	sí	no	no	sí

Programación orientada a arreglos de números con Numpy



ACTUMLOGOS

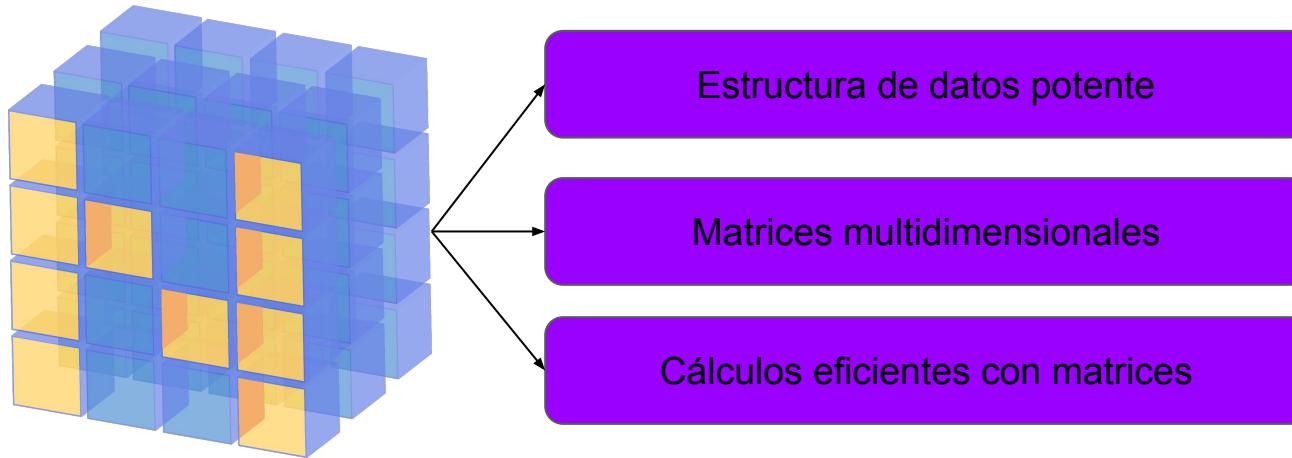
DESARROLLANDO HABILIDADES TECNOLÓGICAS

Programación orientada a arreglos de números con Numpy

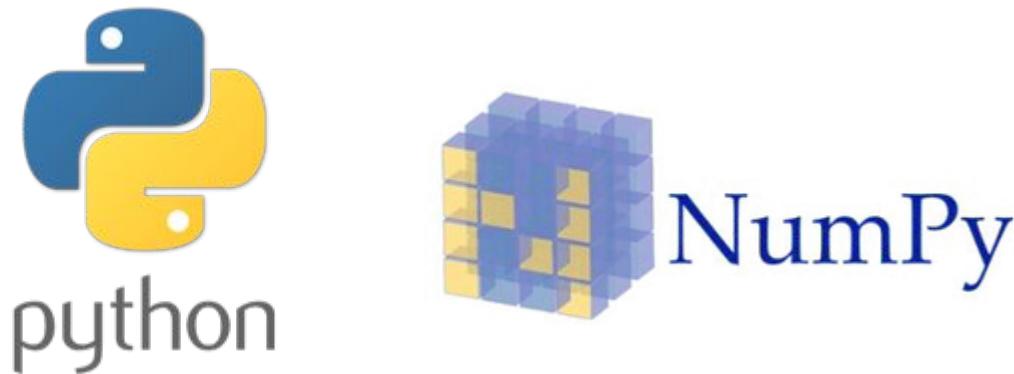
La biblioteca NumPy (Numerical Python) apareció por primera vez en 2006. Ofrece un tipo de matriz n -dimensional funcional de alto rendimiento llamado **ndarray**, que comúnmente se le conoce por su sinónimo, **array**. NumPy es un paquete básico pero poderoso para la computación científica y la manipulación de datos en Python.



Las operaciones en **arrays** son hasta dos veces más rápidas que con listas. En un mundo del *Big Data* en el que las aplicaciones pueden realizar grandes cantidades de procesamiento en grandes cantidades de datos basados en **arrays**, esta ventaja de rendimiento puede ser crítica. Según [bibliotecas.io](#), más de 450 bibliotecas de Python dependen de NumPy. Muchas bibliotecas populares de ciencia de datos, como Pandas, SciPy (Scientific Python) y Keras (para el aprendizaje profundo) se basan o dependen de NumPy.



Generalmente, para procesar listas multidimensionales es necesario utilizar bucles anidados. Un punto fuerte de NumPy es la "programación orientada a arreglos", que utiliza programación de estilo funcional con iteración interna para hacer que las manipulaciones de arreglos sean concisas y directas, eliminando los tipos de errores que pueden ocurrir con la iteración externa de bucles programados explícitamente.



Creando arreglos con datos existentes

La documentación de NumPy recomienda importar el módulo *numpy* como *np* para que pueda acceder a sus miembros con "*np.*":

```
[1] import numpy as np
    numeros = np.array([0, 1, 1, 2, 3, 5, 8])
    print(type(numeros))
    numeros
```

El módulo *numpy* proporciona varias funciones para crear arreglos. Aquí usamos la función de *array*, que recibe como argumento un arreglo u otra colección de elementos y devuelve una nuevo arreglo que contiene los elementos del argumento.

La función `array()` copia las dimensiones de su argumento. Por ejemplo, una matriz a partir de una lista de dos filas por tres columnas:

```
[2] np.array([[1, 2, 3], [4, 5, 6]])
```

Las matrices de autoformatos NumPy, en función de su número de dimensiones, alinean las columnas dentro de cada fila.

```
↳ array([[1, 2, 3],  
         [4, 5, 6]])
```

Atributos de los arreglos

Un objeto `array` tiene atributos que le permiten descubrir información sobre su estructura y contenido. Algunos de los ejemplos de atributos son:

- `dtype`: permite checar el tipo de elementos contenidos en el arreglo.
- `ndim`: indica el número de dimensiones del arreglo.
- `shape`: contiene una tupla indicando las dimensiones del arreglo.
- `size`: indica el número total de elementos.
- `itemsize`: indica el número de bytes requeridos para almacenar cada elemento.

Reto 6: En 5 minutos haga un programa que guarde en un arreglo los primero 5 números de Fermat. Compruebe el funcionamiento de los atributos revisados (dtype, ndim, shape, size e itemsize).

Resultado esperado:

Tips:

- Investigue que es un número de Fermat.
- Investigue cómo acceder a los atributos de los arreglos.
- Consulte en “San Google”

```
[ 3 5 17 257 65537 ]  
int64  
1  
(5, )  
5  
8
```

Solución 1:

```
[1] import numpy as np
    arreglo = np.array([2**(2**0)+1, 2**(2**1)+1, 2**(2**2)+1, 2**(2**3)+1,
                       2**(2**4)+1])
    print(arreglo)
    print(arreglo.dtype)
    print(arreglo.ndim)
    print(arreglo.shape)
    print(arreglo.size)
    print(arreglo.itemsize)
```

Solución 2:

```
[1] import numpy as np
    arreglo = np.array([2**2**i)+1 for i in range(5)])
    print(arreglo)
    print(arreglo.dtype)
    print(arreglo.ndim)
    print(arreglo.shape)
    print(arreglo.size)
    print(arreglo.itemsize)
```

Llenar arreglos con valores específicos.

NumPy proporciona métodos para poder crear arreglos que contengan 0's, 1's o un valor específico, estos métodos son `zeros()`, `ones()` y `full()`. Por defecto, los ceros y unos crean matrices que contienen valores float64. Para un entero, cada función devuelve una matriz unidimensional con el número especificado de elementos.

```
[1] import numpy as np
    ceros = np.zeros(5)
    unos = np.ones(8)
    pers = np.full(5,2)
    print(ceros)
    print(unos)
    print(pers)
```

[Numpy Ones](#)

[Numpy Zeros](#)

[Numpy Full](#)

Para una tupla de enteros, estas funciones devuelven un arreglo multidimensional con las dimensiones especificadas. En el caso de los métodos `zeros()` y `ones()`, se puede especificar el tipo de elementos del arreglo utilizando `dtype` como argumento.

```
[3] mi_arreglo = np.ones((3, 4), dtype=int)
     print(mi_arreglo)
```

Crear arreglos a partir de rangos

NumPy también proporciona métodos para crear arreglos a partir de rangos:

- *arange()*: crea un rango de enteros, similar al método *range()* de la biblioteca estándar. Este método primero determina el número de elementos, reserva memoria y al final guarda los datos.
- *linspace()*: crea un rango con intervalos de punto decimal, recibe tres argumentos que son el valor inicial, el valor final y la cantidad de intervalos que se desean.

Reto 7: En 10 minutos haga un programa que grafique la función coseno. Para ello se le debe pedir al usuario los valores inicial y final así como el intervalo para la gráfica.

Tips:

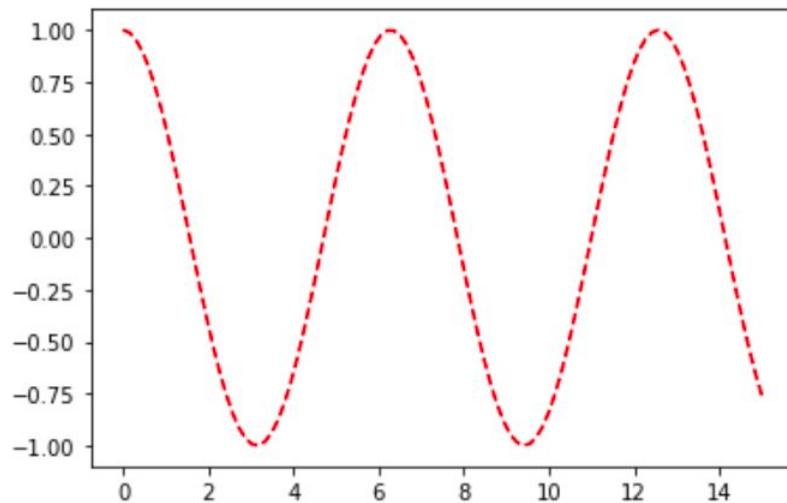
- Utilice la biblioteca matplotlib
- Consulte en “San Google”

Resultado esperado:

Dame el valor inicial: 0

Dame el valor final: 15

Dame el número de intervalos: 100



Solución:

```
[1] # Se importan bibliotecas
    import numpy as np
    import matplotlib.pyplot as plt

    val_ini = input('Dame el valor inicial: ')
    val_fin = input('Dame el valor final: ')
    inter = input('Dame el número de intervalos: ')

    # Se crea el arreglo con numpy
    t = np.linspace(float(val_ini), float(val_fin), int(inter))

    # Se grafica el resultado
    plt.plot(t, np.cos(t), 'r--')
    plt.show()
```

Redimensionar un arreglo

En muchas aplicaciones de ciencia de datos, llega a ser necesario cambiar las dimensiones de un arreglo. NumPy nos ofrece un método para lograr esto, el cual es *reshape()*. Con este método podemos convertir un arreglo de una dimensión a uno multidimensional. Se puede redimensionar cualquier arreglo, siempre y cuando las nuevas dimensiones sean acordes con el número de elementos.

¿Funcionara esta celda?

```
[1] import numpy as np  
mi_arreglo = np.arange(8)  
mi_matriz = mi_arreglo.reshape(3,3)  
print(mi_arreglo)  
print(mi_matriz)
```

Visualización de arreglos de gran dimensión

En ciencia de datos, se suele trabajar con una gran cantidad de información y en ocasiones es necesario visualizar dicha información. NumPy utiliza la notación ... para separar las filas y/o columnas y facilitar la visualización de la información.



```
# Se crea un arreglo y se redimensiona  
print(np.arange(1, 100001).reshape(100, 1000))
```

```
[[    1      2      3 ...     998     999    1000]  
 [ 1001    1002    1003 ...    1998    1999    2000]  
 [ 2001    2002    2003 ...    2998    2999    3000]  
 ...  
 [ 97001   97002   97003 ...   97998   97999   98000]  
 [ 98001   98002   98003 ...   98998   98999   99000]  
 [ 99001   99002   99003 ...   99998   99999 100000]]
```

Listas vs Arreglos

Como ya se mencionó antes, las operaciones con arreglos de NumPy son más rápidas que sus equivalentes operaciones con listas. Para demostrar eso, haremos uso del comando mágico `%timeit`, el cual nos permite conocer el tiempo en que Python tarda en ejecutar cierta expresión.

```
[1] import random
     import numpy as np

     %timeit dados_lista = [random.randrange(1, 7) for i in range(0, 6_000_000)]

     %timeit dados_numpy = np.random.randint(1, 7, 6_000_000)
```

⇨ 1 loop, best of 3: 5.94 s per loop
10 loops, best of 3: 65.8 ms per loop

¡Realmente es mucho más rápido!

Operadores para arreglos

NumPy también nos ofrece operadores que permiten escribir expresiones sencillas que realizan operaciones sobre los elementos del arreglo.

Operaciones aritméticas con escalares

Estas operaciones se llevan a cabo utilizando los operadores aritméticos, vistos con anterioridad. La operación se realiza sobre cada elemento del arreglo con el escalar.

Operador	Descripción
+	Suma
-	Resta
-	Negación
*	Multiplicación
**	Exponente
/	División
//	División entera
%	Módulo

Reto 8: En 5 minutos haga un programa que implemente todos los operadores aritméticos entre un arreglo formado por los primeros 100 números naturales y un valor escalar dado por el usuario.

Resultado esperado:

Tips:

- Utilice la biblioteca NumPy
- Consulte en “San Google”

```
Dame el valor: 2
[ 3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52]
[-1  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48]
[ -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18
-19 -20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30 -31 -32 -33 -34 -35 -36
-37 -38 -39 -40 -41 -42 -43 -44 -45 -46 -47 -48 -49 -50]
[  2   4   6   8   10  12  14  16  18   20  22  24  26  28   30  32  34  36
38  40  42  44  46  48  50  52  54   56  58  60  62  64  66  68  70  72
74  76  78  80  82  84  86  88  90   92  94  96  98 100]
[   1    4    9   16   25   36   49   64   81   100   121   144   169   196
225  256  289  324  361  400  441  484  529  576  625  676  729  784
841  900  961 1024 1089 1156 1225 1296 1369 1444 1521 1600 1681 1764
1849 1936 2025 2116 2209 2304 2401 2500]
[ 0.5  1.  1.5  2.  2.5  3.  3.5  4.  4.5  5.  5.5  6.  6.5  7.
7.5  8.  8.5  9.  9.5 10. 10.5 11. 11.5 12. 12.5 13. 13.5 14.
14.5 15. 15.5 16. 16.5 17. 17.5 18. 18.5 19. 19.5 20. 20.5 21.
21.5 22. 22.5 23. 23.5 24. 24.5 25. ]
[ 0  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 11 12
12 13 13 14 14 15 15 16 16 17 17 18 18 19 19 20 20 21 21 22 22 23 23 24
24 25]
[1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1 0 1 0 1 0 1 0]
```

Solución:

```
[1] # Se importa la biblioteca numpy
     import numpy as np

# Se define el arreglo con 100 numeros enteros
mi_arreglo = np.arange(1, 51)

# El usuario da el valor
val = int(input('Dame el valor: '))

# Se realizan las operaciones y se imprimen en pantalla
print(mi_arreglo + val)
print(mi_arreglo - val)
print(- mi_arreglo)
print(mi_arreglo * val)
print(mi_arreglo ** val)
print(mi_arreglo / val)
print(mi_arreglo // val)
print(mi_arreglo % val)
```

Operaciones aritméticas entre arreglos

Las operaciones aritméticas entre arreglos se pueden llevar a cabo cuando estos tienen las mismas dimensiones. Las operaciones se realizan elemento a elemento (element-wise).

```
[1] import numpy as np  
  
numeros1 = np.array([1, 2, 3, 4])  
numeros2 = np.array([3, 4, 5, 6])  
print(numeros1 * numeros2)
```

¿Cuales son los valores que se imprime en pantalla?

→ [3 8 15 24]

Comparaciones entre arreglos

Las comparaciones entre arreglos se pueden llevar a cabo con valores escalares y otros arreglos (que deben ser del mismo tamaño). El resultado de esta operación es un arreglo de valores booleanos (verdadero o falso).

Operador	Descripción
==	¿iguales?
!=	¿distintos?
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que

```
[1] import numpy as np  
  
valor = 1  
numeros1 = np.array([1, 2, 3, 4])  
numeros2 = np.array([3, 4, 3, 6])  
print(numeros1 == numeros2)  
print(numeros1 > valor)
```

¿Cuáles son los valores que se imprime en pantalla?

⇨ [False False True False]
[False True True True]

Reto 9: En 10 minutos haga un programa que implemente todos los operadores de comparación entre un par de arreglos. Los arreglos deben tener las estaturas de 4 alumnos cada uno, las cuales deben ser introducidas por el usuario.

Tips:

- Investigue el método `append()` de NumPy
- Considere ocupar un ciclo
- Consulte en “San Google”

Resultado esperado:

```
↳ Dame la estatura: 1.75
Dame la estatura: 1.70
Dame la estatura: 1.65
Dame la estatura: 1.81
Dame la estatura: 1.75
Dame la estatura: 1.83
Dame la estatura: 1.62
Dame la estatura: 1.67
[ True False False False]
[False  True  True  True]
[False  True False False]
[False False  True  True]
[ True  True False False]
[ True False  True  True]
```

Solución:

```
[1] import numpy as np

arreglo1 = np.array([])
arreglo2 = np.array([])

for i in range(2):
    for j in range(4):
        val = float(input('Dame la estatura: '))
        if i==1:
            arreglo1 = np.append(arreglo1, val)
        else:
            arreglo2 = np.append(arreglo2, val)
print(arreglo1 == arreglo2)
print(arreglo1 != arreglo2)
print(arreglo1 > arreglo2)
print(arreglo1 < arreglo2)
print(arreglo1 >= arreglo2)
print(arreglo1 <= arreglo2)
```

Métodos para hacer cálculos con arreglos NumPy

Los arreglos NumPy poseen métodos para realizar cálculos con su contenido. Por defecto, estos métodos ignoran las dimensiones del arreglo y trabajan con todos los elementos, aunque también se pueden realizar en cada una de las dimensiones. Algunos de estos métodos son:

- `sum()`: la suma de los elementos del arreglo.
- `min()`: el valor mínimo dentro del arreglo.
- `max()`: el valor máximo dentro del arreglo.
- `mean()`: el promedio de todos los elementos del arreglo.
- `std()`: la desviación estándar.
- `var()`: la varianza.

Reto 10: En 10 minutos haga un programa que implemente los métodos para realizar para realizar cálculos con NumPy previamente revisados. Estas operaciones se harán con los *Consumer Reach Points* (CRPs), de las siguientes marcas de la tabla.

Ranking	Marca	Fabricante	CRP	Penetración
1		COCA-COLA COMPANY	5.722	43,3%
2		COLGATE-PALMOLIVE COMPANY	3.992	64.6%
3		NESTLÉ	2.755	32,7%
4		UNILEVER	2.338	26,5%
5		NESTLÉ	2.137	22,8%
6		PEPSICO	1.984	24,6%
7		PEPSICO	1.747	27,2%
8		UNILEVER	1.647	29,4%
9		UNILEVER	1.457	34,7%
10		P&G	1.438	29,8%

Tips:

- Utilice los métodos revisados
- Consulte en “San Google”

Resultado esperado:

- ⇨ La suma es: 25.2170
El mínimo es: 1.4380
El máximo es: 5.7220
La media es: 2.5217
La desviación estándar es: 1.2885
La varianza es: 1.6602

Solución:

```
[1] # Se importan bibliotecas
import numpy as np

# Se crea el arreglo
crp = np.array([5.722, 3.992, 2.755, 2.338, 2.137, 1.984, 1.747,
                1.647, 1.457, 1.438])

# Métodos para hacer cálculos
print('La suma es: %0.4f' % crp.sum())
print('El mínimo es: %0.4f' % crp.min())
print('El máximo es: %0.4f' % crp.max())
print('La media es: %0.4f' % crp.mean())
print('La desviación estándar es: %0.4f' % crp.std())
print('La varianza es: %0.4f' % crp.var())
```

Funciones universales

La biblioteca NumPy ofrece decenas de *funciones universales* (ufuncs) que implementan varias operaciones de elemento a elemento. Estas funciones son llamadas usando como argumento un arreglo, dos arreglos o listas.

```
[1] import numpy as np

arreglo = np.array([1, 4, 9, 16, 25])
lista = [1.1, 2.1, 3.2, 4.7, 5.1]

print(np.sqrt(arreglo))
print(np.add(arreglo, lista))
print(np.floor(lista))
print(np.greater(arreglo, lista))
```

¿Cuáles son los valores
que se imprime en pantalla?

```
↳ [1. 2. 3. 4. 5.]
   [ 2.1 6.1 12.2 20.7 30.1]
   [1. 2. 3. 4. 5.]
   [False True True True True]
```

Indexar y dividir arreglos

Los arreglos de una dimensión se pueden indexar y dividir, tal como se hace en las listas y tupla. Sin embargo, existen otras formas utilizando características específicas de los arreglos NumPy para arreglos de dos o más dimensiones.

Primero, para acceder a elementos concretos dentro del arreglo (indexar), se utiliza una tupla con la fila y columna a la cual se desea acceder.

```
[1] import numpy as np

calif = np.array([[87, 96, 70],
                  [100, 87, 90],
                  [94, 77, 90],
                  [100, 81, 82]])
calif[0, 1]
```

Indexar y dividir arreglos

Otra curiosidad en Python es que el acceso de los elementos también se puede hacer en reversa, considerando un índice negativo.

```
[1] import numpy as np  
  
a = np.arange(15)  
print(a)  
print(a[-1])  
print(a[0])  
print(a[1])
```

¿Cuáles son los valores que se imprime en pantalla?

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]  
14  
0  
1
```

Otra operación interesante es extraer desde el principio (o final) del arreglo una cierta cantidad de filas, lo cual se logra con la sintaxis `arreglo[:fila_final+1]` o `arreglo[fila_inicial:]` respectivamente, lo cual también se logra con las columnas con la sintaxis `arreglo[:,columna_final+1]` , `arreglo[:,columna_inicial:]`

¿Cuáles son los valores que se imprime en pantalla?

```
[2] print('Las primeras dos filas son: \n %s' % (calif[:2]))  
print('Las últimas tres filas son: \n %s' % (calif[1:]))  
print('Las primeras dos columnas son: \n %s' % (calif[:, :2]))  
print('Las últimas dos columnas son: \n %s' % (calif[:, 1:]))
```

Las primeras dos filas son:
[[87 96 70]
 [100 87 90]]
Las últimas tres filas son:
[[100 87 90]
 [94 77 90]
 [100 81 82]]
Las primeras dos columnas son:
[[87 96]
 [100 87]
 [94 77]
 [100 81]]
Las últimas dos columnas son:
[[96 70]
 [87 90]
 [77 90]
 [81 82]]

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

[The NumPy array object](#)

Copias de arreglos

Un método interesante para crear copias de arreglos es el método `view()`, el cual crea una copia del arreglo original pero con las características de que esta copia se modificará si se modifican los elementos del arreglo original. Mientras que el método `copy()` crea una copia independiente del arreglo original.

```
[1] import numpy as np

arreglo = np.arange(5)

copia1 = arreglo.view()
print(copia1)
arreglo *= 2
print(copia1)

copia2 = arreglo.copy()
print(copia2)
arreglo *= 2
print(copia2)
```

¿Cuales son los valores
que se imprime en pantalla?

→ [0 1 2 3 4]
[0 2 4 6 8]
[0 2 4 6 8]
[0 2 4 6 8]

Cuando se utilizan los métodos `copy()` y `view()` se crea un nuevo objeto que tiene una ubicación en memoria diferente al arreglo original, si solamente asignamos un arreglo a una nueva variable, no se está creando una copia, se está haciendo referencia al mismo elemento.

```
[2] arreglo1 = np.arange(10)
    arreglo2 = arreglo1
    print(id(arreglo1))
    print(id(arreglo2))
    arreglo3 = arreglo2.copy()
    print(id(arreglo3))
    arreglo4 = arreglo2.view()
    print(id(arreglo4))

    arreglo2[1] = 100
    print(arreglo1)
```

Cadenas de caracteres (strings)

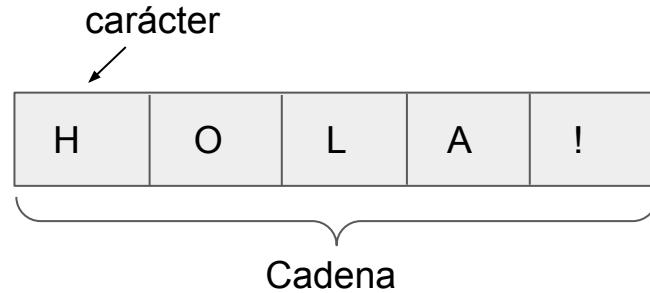


ACTUMLOGOS

DESARROLLANDO HABILIDADES TECNOLÓGICAS

Introducción

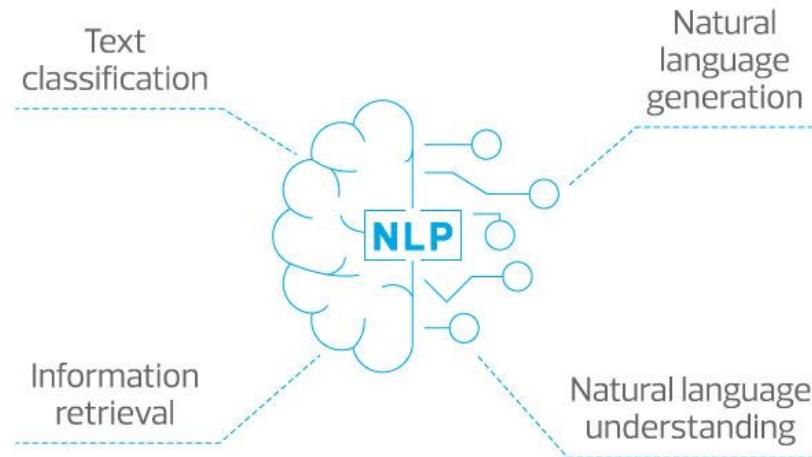
Las cadenas de caracteres, o simplemente cadenas, son secuencias inmutables que contienen caracteres.



En Python se utilizan comillas simples o dobles para delimitar las cadenas.

```
[1] cadena1 = 'Esta es una cadena'  
cadena2 = "Esta tambien es una cadena"  
print(cadena1)  
print(cadena2)
```

El manejo correcto de cadenas es fundamental para aplicaciones de inteligencia artificial relacionadas con el Procesamiento del Lenguaje Natural (NLP por sus siglas en inglés).



[Procesamiento del lenguaje
natural](#)

5

Applications of
Natural Language Processing
for Businesses



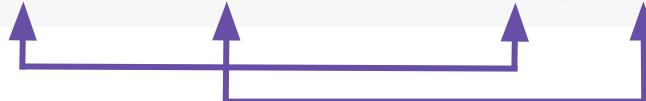
Formateando cadenas

En muchas aplicaciones de inteligencia artificial, es necesario dar un formato adecuado a las cadenas de caracteres, con el objetivo de leerlas y entenderlas. Vamos a ver primero las formas tradicionales de dar formato a cadenas.

- Usando el operador %

Las cadenas tienen una operación integrada que utiliza el operador % para dar formato.

```
[1] nombre = "Juan"  
edad = 25  
print("Hola, %s. Tienes %d años." % (nombre, edad))
```



%d	Entero con signo
%f	Punto flotante
%s	Cadena
%x	Hexadecimal

- Usando *str.format()*

El método `str.format()` es una mejora al formateo realizado con el operador `%`. Con `str.format()`, los campos de reemplazo están marcados con llaves.

```
[1] nombre = "Juan"  
     edad = 25  
     print("Hola, {}. Tienes {} años.".format(nombre, edad))
```

¿Cuáles son las ventajas de este formateo?

f-strings

La *f-strings* permiten realizar el formateo de una manera sencilla y estuvieron disponibles a partir de la versión 3.6 de Python. Este tipo de cadenas también se conocen como "cadenas con formato literal". Una de sus características es que llevan una *f* al principio y utilizan llaves que contienen expresiones con los valores que se desean colocar. Las expresiones se evalúan en tiempo de ejecución y luego se formatean.

```
[1] nombre = "Juan"  
edad = 25  
print(f"Hola, {nombre}. Tienes {edad} años.")
```

¿Cuáles son las ventajas de este formateo?

Reto 11: En 10 minutos haga una función que imprima en pantalla el CRP de las siguientes empresas utilizando f-strings, para ello almacene los CRP en un arreglo NumPy.

Ranking	Marca	Fabricante	CRP	Penetración
1		COCA-COLA COMPANY	5.722	43,3%
2		COLGATE-PALMOLIVE COMPANY	3.992	64.6%
3		NESTLÉ	2.755	32,7%
4		UNILEVER	2.338	26,5%
5		NESTLÉ	2.137	22,8%
6		PEPSICO	1.984	24,6%
7		PEPSICO	1.747	27,2%
8		UNILEVER	1.647	29,4%
9		UNILEVER	1.457	34,7%
10		P&G	1.438	29,8%

Tips:

- Consulte la sección de arreglos NumPy
- Considere utilizar la función enumerate en el ciclo for
- Consulte en “San Google”

Resultado esperado:

- ⇒ La empresa 1 tiene un crp de 5.722
La empresa 2 tiene un crp de 3.992
La empresa 3 tiene un crp de 2.755
La empresa 4 tiene un crp de 2.338
La empresa 5 tiene un crp de 2.137
La empresa 6 tiene un crp de 1.984
La empresa 7 tiene un crp de 1.747
La empresa 8 tiene un crp de 1.647
La empresa 9 tiene un crp de 1.457
La empresa 10 tiene un crp de 1.438

Solución:

```
[1] # Se importa la biblioteca numpy  
import numpy as np
```

```
[2] # Se define la función
```

```
def CRP():  
    crp_tot = np.array([5.722, 3.992, 2.755, 2.338, 2.137, 1.984, 1.747,  
                       1.647, 1.457, 1.438])  
    for num, crp in enumerate(crp_tot, start=1):  
        print(f'La empresa {num} tiene un crp de {crp}')
```

```
[3] # Se manda a llamar la función  
CRP()
```

Concatenando y repitiendo cadenas

Para concatenar (unir) cadenas se utiliza el operador + , mientras que para repetir cadenas se utiliza el operador *.

```
[1] cadena1 = f'Python '
    cadena2 = f'es un gran lenguaje'
    cadena3 = cadena1 + cadena2
    cadena4 = cadena1 * 5
    print(cadena3)
    print(cadena4)
```

→ Python es un gran lenguaje
Python Python Python Python Python

Quitar espacios en blanco de cadenas

En muchas ocasiones es necesario retirar espacios en blanco de las cadenas. Para quitar los espacios al principio y al final de una cadena, se utiliza el método *strip()*.

```
[1] mi_cadena = f'      Python es un gran lenguaje      '
    print(mi_cadena)
    mi_cadena = mi_cadena.strip()
    print(mi_cadena)
```



Python es un gran lenguaje
Python es un gran lenguaje

Cambiando mayúsculas y minúsculas

Para cambiar a mayúsculas o minúsculas toda una cadena, se tienen los métodos *lower()* y *upper()*. Por otra parte si queremos cambiar solo la primera letra de la cadena a mayúscula tenemos el método *capitalize()* mientras que si queremos cambiar la primera letra de cada palabra, tenemos el método *title()*.

```
[1] mi_cadena = f'python Es un gran lenguaje'  
     print(mi_cadena.upper())  
     print(mi_cadena.lower())  
     print(mi_cadena.capitalize())  
     print(mi_cadena.title())
```

→ PYTHON ES UN GRAN LENGUAJE
python es un gran lenguaje
Python es un gran lenguaje
Python Es Un Gran Lenguaje

Búsqueda y reemplazo de subcadenas

Puede buscar en una cadena uno o más caracteres adyacentes, conocidos como subcadena, para contar el número de ocurrencias, determinar si una cadena contiene una subcadena o determinar el índice en el que reside una subcadena en una cadena. Cada método compara los caracteres lexicográficamente utilizando sus valores numéricos subyacentes.

- **count()**: cuenta el número de ocurrencias de la subcadena en la cadena.
- **index()**: busca una subcadenas y regresa el índice donde comienza la subcadena.
- **startswith()**: determina si la subcadena se encuentra al principio de la cadena.
- **endswith()**: determina si la subcadena se encuentra al final de la cadena.
- **replace()**: reemplaza una subcadena por otra dentro de una cadena

```
[1] oracion = 'El curso de Python para IA me encanta'  
      print(oracion.count('Python para'))  
      print(oracion.index('Python'))
```

Reto 12: En 5 minutos complete la celda del reto en donde se implementan las funciones de búsqueda y reemplazo de subcadenas. Para ello se utiliza el siguiente texto tomado de [El futuro de la IA: hacia inteligencias artificiales realmente inteligentes](#).

El objetivo último de la IA, lograr que una máquina tenga una inteligencia de tipo general similar a la humana, es uno de los objetivos más ambiciosos que se ha planteado la ciencia. Por su dificultad, es comparable a explicar el origen de la vida, el origen del universo o conocer la estructura de la materia

Tips:

- Utilice los métodos revisados
- Consulte en “San Google”

Resultado esperado:

⇒ Ocurrencias de la palabra "objetivo": 2
Índice de la palabra "IA": 25
"El" es la palabra inicial: True
"universo" es la palabra final: False
El texto sin espacios es:
ElobjetivoúltimodelaIA,lograrqueunamáquina
tengaunaainteligenciadetipogeneralsimilaralahumana,esuno
delosobjetivosmásambiciososquesehaplanteadolaciencia.
Porsudificultad,escomparableaexplicarelorigendelavida,
elorigendeluniversooconocerlaestructuradelamateria

Solución:

```
[1] texto_ia = '''El objetivo último de la IA, lograr que una máquina
    tenga una inteligencia de tipo general similar a la humana, es uno
    de los objetivos más ambiciosos que se ha planteado la ciencia.
    Por su dificultad, es comparable a explicar el origen de la vida,
    el origen del universo o conocer la estructura de la materia'''

    ocurrencias = texto_ia.count('objetivo')
    indice = texto_ia.index('IA')
    inicia = texto_ia.startswith('El')
    termina = texto_ia.endswith('universo')
    nuevo_texto = texto_ia.replace(' ', '')

    print(f'Occurencias de la palabra "objetivo": {ocurrencias}')
    print(f'Indice de la palabra "IA": {indice}')
    print(f'"El" es la palabra inicial: {inicia}')
    print(f'"universo" es la palabra final: {termina}')
    print('El texto sin espacios es: ')
    print(nuevo_texto)
```

Separando y uniendo cadenas

Cuando lees una oración, tu cerebro la divide en palabras individuales o fichas, cada una de las cuales transmite significado. Los intérpretes de Python *tokenizan* las declaraciones, dividiéndolas en componentes individuales como palabras clave, identificadores, operadores y otros elementos de un lenguaje de programación. Los *tokens* normalmente están separados por caracteres de espacio en blanco, tabulación y nueva línea, aunque se pueden usar otros caracteres: los separadores se conocen como delimitadores.

```
[1] mensaje = 'Hola, el curso de Python para IA me encanta'
     print(mensaje)
     print(mensaje.split())

→ Hola, el curso de Python para IA me encanta
   ['Hola,', 'el', 'curso', 'de', 'Python', 'para', 'IA', 'me', 'encanta']
```

Tokenización

Para separar y unir cadenas, podemos utilizar los métodos *split()* y *join()*, dónde se puede indicar qué elemento se utiliza como delimitador.

```
[1] paises = ['Rusia', 'Brasil', 'Sudáfrica', 'Alemania', 'Corea-Japón']
    paises_mundial = ':' .join(pais for pais in paises)
    print(paises_mundial)
```

⇨ Rusia:Brasil:Sudáfrica:Alemania:Corea-Japón

Métodos para caracteres

Muchos lenguajes de programación consideran como tipos diferentes las cadenas de los caracteres. En Python, un carácter es simplemente una cadena de un carácter. Python proporciona métodos para probar si una cadena coincide con ciertas características.

- `isdigit()`: determina si la cadena tiene solo dígitos (0-9)
- `isalnum()`: regresa *True* si la cadena contiene sólo caracteres alfanuméricos
- `isalpha()`: regresa *True* si la cadena contiene sólo caracteres alfabéticos
- `isdecimal()`: regresa *True* si la cadena contiene sólo enteros sin signos + o -
- `isnumeric()`: regresa *True* si los caracteres en la cadena representan un valor numérico sin signos + , - y punto decimal.
- `islower()`: regresa *True* si todos los caracteres en la cadena son minúsculas
- `isupper()`: regresa *True* si todos los caracteres en la cadena son mayúsculas.

Reto 13: En 10 minutos complete la celda del reto que se utiliza para probar el funcionamiento de los métodos para caracteres en el siguiente texto: “*En el año 2019 , NETFLIX tiene 167.1 millones de suscriptores*”

Tips:

- Utilice los métodos revisados
- Consulte en “San Google”

Resultado esperado:

```
[ 'En', 'en', 'año', '2019', ',', 'NETFLIX', 'tiene', '167.1', 'millones', 'de', 'suscriptores' ]  
Las palabras con digitos son: [False, False, False, True, False, False, False, False, False, False]  
Las palabras con alfanumericos son: [True, True, True, True, False, True, True, False, True, True]  
Las palabras con alfabeticos son: [True, True, True, False, False, True, True, False, True, True]  
Las palabras con decimales son: [False, False, False, True, False, False, False, False, False, False]  
Las palabras con numericos son: [False, False, False, True, False, False, False, False, False, False]  
Las palabras con minusculas son: [False, True, True, False, False, True, False, True, True, True]  
Las palabras con mayusculas son: [False, False, False, False, True, False, False, False, False, False]
```



Solución:

```
[1] mi_cadena = 'En el año 2019 , NETFLIX tiene 167.1 millones de suscriptores'
separados = mi_cadena.split()
digitos = [token.isdigit() for token in separados]
alnum = [token.isalnum() for token in separados]
alpha = [token.isalpha() for token in separados]
decimal = [token.isdecimal() for token in separados]
numeric = [token.isnumeric() for token in separados]
minus = [token.islower() for token in separados]
mayus = [token.isupper() for token in separados]

print(separados)
print(f'Las palabras con digitos son: {digitos}')
print(f'Las palabras con alfanumericos son: {alnum}')
print(f'Las palabras con alfabeticos son: {alpha}')
print(f'Las palabras con decimales son: {decimal}')
print(f'Las palabras con numericos son: {numeric}')
print(f'Las palabras con minusculas son: {minus}')
print(f'Las palabras con mayusculas son: {mayus}')
```

Expresiones regulares

En ocasiones, deberá reconocer patrones en el texto, como números de teléfono, direcciones de correo electrónico, códigos postales, direcciones de páginas web, números de Seguro Social y más. Una expresión regular describe un patrón de búsqueda para caracteres coincidentes en otras cadenas. Las expresiones regulares permiten extraer datos de texto no estructurado, como publicaciones en redes sociales.



Módulo *Re*

Para usar expresiones regulares en Python, se cuenta con el módulo *re* de la Biblioteca estándar de Python. Una de las funciones de expresión regular más simples es *fullmatch*, que verifica si la cadena completa en su segundo argumento coincide con el patrón en su primer argumento.

```
[1] import re

# Buscando coindicencias
cp = '09715'

print('Coincide') if re.fullmatch(cp, '09715') else print('No coincide')
print('Coincide') if re.fullmatch(cp, '09730') else print('No coincide')
```

Metacaracteres, clases de caracteres y cuantificadores

Las expresiones regulares generalmente contienen varios símbolos especiales llamados metacaracteres, que se muestran a continuación

[] { } () \ * + ^ \$? . |

El metacaracter "\\" comienza cada una de las clases de caracteres predefinidas, cada una de las cuales coincide con un conjunto específico de caracteres.

```
[2] print('Valido') if re.fullmatch('\d{5}', '002215') else print('Invalido')
    print('Valido') if re.fullmatch('\d{5}', '9876') else print('Invalido')
```

La siguiente tabla muestra algunas clases de caracteres comunes y los grupos de caracteres que buscan la coincidencia

Clase de carácter	Coincidencia
\d	Cualquier dígito entre 0-9
\D	Cualquier carácter que no es un dígito
\s	Cualquier espacio en blanco, como espacios, tabuladores, nuevas líneas
\S	Cualquier carácter que no es un espacio en blanco
\w	Cualquier carácter alfanumérico, esto es cualquier letra mayúscula o minúscula, cualquier dígito o un guión bajo
\W	Cualquier carácter que no es alfanumérico

Reto 14: En 5 minutos escribe un programa que determine cuántos números de celular y fijos se encuentran en la siguiente frase: *A nuestros estimados clientes, les proporcionamos nuestros números celular: 5545221035 y 4675104590 ; fijo: 56460923*

Tips:

- Utilice la clase de carácter más apropiada
- Consulte en “San Google”

Resultado esperado:

→ A nuestros estimados clientes, les proporcionamos nuestros números celular: 5545221035 y 4675104590 ; fijo: 56460923
La cantidad de teléfonos celulares es 2
La cantidad de teléfonos fijos es 1

Solución:

```
[1] import re

celulares = 0
fijo = 0

frase = '''A nuestros estimados clientes, les proporcionamos nuestros números
celular: 5545221035 y 4675104590 ; fijo: 56460923'''
print(frase)
palabras = frase.split()
for palabra in palabras:
    if(re.fullmatch('\d{10}', palabra)):
        celulares += 1
    elif(re.fullmatch('\d{8}', palabra)):
        fijo += 1
print(f'La cantidad de telefonos celulares es {celulares}')
print(f'La cantidad de telefonos fijos es {fijo}')
```

Sustitución de subcadenas y división de cadenas

El módulo *re* proporciona una función *sub* para reemplazar patrones en una cadena y la función *split* para dividir una cadena en pedazos, según los patrones.

```
[1] import re

frase = 'Python es un gran lenguaje'

reemplazo = re.sub(' ', ',', frase)
print(reemplazo)
separacion = re.split('\s', frase)
print(separacion)
```

¿Cuales son los valores que se imprime en pantalla?

- Python,es,un,gran,lenguaje
['Python', 'es', 'un', 'gran', 'lenguaje']

Documentación de NumPy

<https://numpy.org/doc/>

Documentación de Strings

<https://docs.python.org/3/library/string.html>

Módulo *re*

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



NumPy Documentation

The most up-to-date NumPy documentation can be found at [Latest \(development\) version](#). It includes a user guide, full reference documentation, a developer guide, and meta information.

Other links:

- [NumPy Enhancement Proposals](#) (which include the NumPy Roadmap and detailed plans for major new features).
- [Numpy 1.17 Manual](#)
[HTML+zip] [Reference Guide PDF] [User Guide PDF]
- [Numpy 1.16 Manual](#)
[HTML+zip] [Reference Guide PDF] [User Guide PDF]
- [Numpy 1.15 Manual](#)
[HTML+zip] [Reference Guide PDF] [User Guide PDF]
- [Numpy 1.14 Manual](#)
[HTML+zip] [Reference Guide PDF] [User Guide PDF]
- [Numpy 1.13 Manual](#)
[HTML+zip] [Reference Guide PDF] [User Guide PDF]
- [Older versions \(on scipy.org\)](#)

Python » English 3.8.1 Documentation » The Python Standard Library » Text Processing Services » Quick search Go | previous | next | modules | index

Table of Contents

- [string — Common string operations](#)
 - String constants
 - Custom String Formatting
 - Format String Syntax
 - Format Specification Mini-Language
 - Format examples

string — Common string operations ¶

Source code: [Lib/string.py](#)

See also: [Text Sequence Type — str](#)

String Methods

Conclusión Final

- Una de las estructuras de datos importante dentro de Python son los diccionarios que permiten crear colecciones de datos que están relacionadas por una clave y un valor.
- El uso de una clave facilita el acceso a los elementos de la estructura, en lugar de ocupar un índice.
- Los conjuntos son una estructura de datos que permite guardar una colección desordenada de datos, por lo cual no permite indexación.
- Los conjuntos permiten corroborar la pertenencia de un elemento y/o evitar la duplicación de datos.
- La biblioteca NumPy es una de las más importantes dentro del mundo de Python, la cual permite una manejo eficiente de arreglos y matrices de datos.
- Las cadenas de caracteres es un tipo de dato comúnmente utilizado en aplicaciones de Python y en inteligencia artificial, como lo es el procesamiento del lenguaje natural.
- Las expresiones regulares permiten reconocer patrones de texto.

Glosario

Diccionario: tipo de dato nativo de Python que permite almacenar valores ordenados mediante claves.

Conjuntos: colección desordenada de valores no repetidos, que son mutables.

NumPy: paquete utilizado para el manejo eficiente de arreglos y matrices, constituyendo una biblioteca de funciones matemáticas de alto nivel.

Strings: secuencias inmutables que contienen caracteres.

Indexar: acción de acceder a los elementos de un objeto que lo permite.

Concatenar: acción de unir los elementos de una cadena con otra.

Expresión regular: las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto.