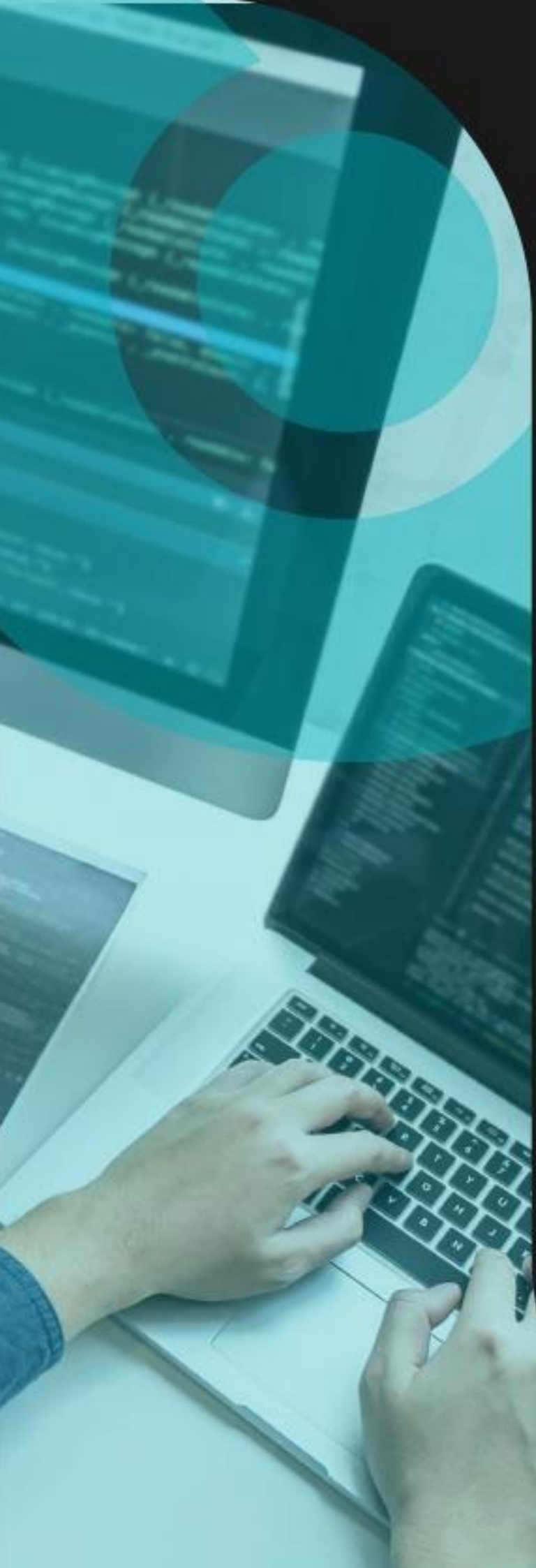

AWAKELAB

BASECAMP

Ciencia de Datos

Módulo : Fundamentos de Programación en Python

Aprendizaje Esperado

- Construye un algoritmo utilizando estructuras de dato del lenguaje Python para resolver un problema

Estructura de Datos

Exploramos variables que contienen colecciones de datos como cadenas, listas, diccionarios y tuplas.

Métodos de String

string.replace(string_viejo, string_nuevo)

El método replace() devuelve una copia de la cadena con la subcadena vieja reemplazada por una nueva

Ejemplo:

```
cadena = "Hola Mundo"
cadena.replace("Mundo", "Internet")

# 'Hola Internet'
```

string.partition(char)

El método partition() divide la cadena en la primera aparición de char y devuelve una tupla que contiene la parte anterior a char, el mismo char, y la parte posterior de char.

Ejemplo:

```
cadena = "Hola Mundo"
cadena.partition("la")

# ('Ho', 'la', ' Mundo')
```

string.title()

El método title() devuelve una copia de la cadena donde las palabras comienzan con una letra mayúscula.

Ejemplo:

```
cadena = "mi diario python"
cadena.title()

# 'Mi Diario Python'
```

string.swapcase()

El método swapcase() devuelve una copia de la cadena con los caracteres en mayúsculas convertidos en minúsculas y viceversa.

Ejemplo:

```
cadena = "Mi Diario Python"
cadena.swapcase()

# 'mI dIARIO pYTHON'
```

strig.startswith (prefijo)

El método startswith() devuelve True si la cadena comienza con el prefijo, de lo contrario devuelve False.

Ejemplo:

```
cadena = "Mi Diario Python"
cadena.startswith("Mi")

# True
```

string.split(sep)

El método split() devuelve una lista de las palabras en la cadena, utilizando a sep como la cadena delimitadora.

Ejemplo:

```
cadena = "Luis, Jose, Maria, Sofia, Miguel"
cadena.startswith(",")

# ['Luis', 'Jose', 'Maria', 'Sofia', 'Miguel']
```

string.zfill(ancho)

El método zfill() devuelve una copia de la cadena que se rellena con 0 dígitos ASCII para hacer una cadena de *ancho* de longitud .

Ejemplo:

```
"356".zfill(6)
# '000356'
```

Formateo de strings

Construir una única cadena de caracteres a partir de otras más pequeñas es algo que estarás haciendo prácticamente siempre al momento de escribir código, sea en Python o en cualquier otro lenguaje. Por esta razón, tener un buen dominio de las herramientas para llevarlo a cabo es fundamental para diseñar un mejor código: más eficiente, más legible

Método format.

Los datos que queremos incluir dentro de la cadena se pasan como argumentos a la función **format**. El número dentro de las llaves indica la posición del argumento que será reemplazado en aquel lugar. Como **nombre** es el primer argumento, empezando a contar desde el cero, entonces será ubicado en donde se encuentra **{0}**.

```
edad = 20
"Tu nombre es {0} y tienes {1} años.".format(nombre,
edad)

# 'Tu nombre es Juan y tienes 20 años.'
```

Esto resulta bastante práctico ya que podemos alterar la cadena sin necesidad de cambiar el orden de los argumentos.

```
edad = 20
"Tienes {1} años y te llamas {0}.".format(nombre,
edad)

# 'Tienes 20 años y te llamas Juan.'
```

Incluso puedes colocar las llaves tantas veces como quieras sin necesidad de repetir los argumentos, lo que hubiese sido una limitación en el sistema anterior.

```
"{0} {1} {0} {1} {1} {0} {0}".format(nombre, edad)
```

```
# 'Juan 20 Juan 20 20 Juan Juan'
```

Si te resulta más cómodo, a partir de Python 2.7 puedes omitir los números dentro las llaves.

```
"Tu nombre es {} y tienes {} años".format(nombre, edad)

# 'Tu nombre es Juan y tienes 20 años'
```

Aunque lógicamente, omitiendo las posiciones no te permitirá realizar repeticiones, pero no será necesario en la mayoría de los casos.

Otra característica que añade este nuevo sistema es la posibilidad de especificar con un nombre determinado los valores que queremos incluir.

```
"Tu nombre es {a} y tienes {b} años".format(a = nombre, b = edad)

# 'Tu nombre es Juan y tienes 20 años'
```

Si bien en el ejemplo puede resultar algo trivial, puede serte de utilidad para cadenas más complejas. No hay problema en combinar ambos métodos.

```
"Tu nombre es {0} y tienes {1} años. Mides {altura} metros".format(nombre, edad, altura = 1.75)

# 'Tu nombre es Juan y tienes 20 años. Mides 1.75 metros'
```

Este sistema de formateo incluye las mismas herramientas que describimos anteriormente. Simplemente añadimos dos puntos y especificamos los diversos parámetros. Por ejemplo:

```
for i in range (-3, 4):
      print("{0:+}".format(i))
-3
-2
-1
+0
+1
+2
+3
```

Existen algunas diferencias. Por ejemplo, al indicar la cantidad de caracteres que queremos imprimir como mínimo, este sistema por defecto utilizará una alineación izquierda, mientras que el anterior derecha. Para cambiar este comportamiento utilizamos los caracteres «<" (izquierda) y ">» (derecha).

```
for nombre in ("Juan", "Pedro", "Francisco"):
      print("{0:>9}".format(nombre))

     Juan
    Pedro
Francisco
```

Empleando el carácter «^» causará que los nombres se centren

```
for nombre in ("Juan", "Pedro", "Francisco"):
      print("{0:^9}".format(nombre))


  Juan
  Pedro
Francisco
```

Para convertir a hexadecimal:

```
"0x{:X}".format(1000) # omitimos el 0
# '0x3E8'
```

Al igual que en el método anterior, para imprimir literalmente llaves, deben colocarse dos veces.

```
"{{}}".format()
# '{}'
```

Este sistema de formateo estándar de Python está explicado íntegramente en el documento PEP 3101 — Advanced String Formatting (ver referencia).

El siguiente código crea una suerte de pirámide a partir de un número determinado de caracteres.

```
n = 20
for i in range(1,n,2):
```

```
      print("  {0:^{1}}".format("A"*i, n))


           A
          AAA
         AAAAA
        AAAAAAA
       AAAAAAAAA
      AAAAAAAAAAA
     AAAAAAAAAAAAA
    AAAAAAAAAAAAAAA
   AAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAA
```

f-strings

Con la idea de mejorar la legibilidad respecto de los sistemas anteriores, en Python 3.6 surgen las denominadas **«f-strings»** o **«cadenas-f»**. Cuando una cadena esté precedida por una **«f»**, utilizando llaves se podrán ubicar expresiones dentro de la misma que serán luego ejecutadas.

```
nombre = "Juan"
edad = 20
altura = 1.75
f"Te llamas {nombre}, tienes {edad} años y mides
```

```
{altura} metros."
# 'Te llamas Juan, tienes 20 años y mides 1.75
metros.'
```

Nótese que son expresiones, no simples identificadores, a diferencia de los métodos anteriores.

```
f"Tu nombre al revés es {nombre[::-1]}."
# 'Tu nombre al revés es nauJ.'

f"Dentro de 5 años tendrás {edad + 5} años."
# 'Dentro de 5 años tendrás 25 años.'
```

En versiones anteriores, utilizando **str.format** se intentaba emular este funcionamiento vía el siguiente código.

```
"Tu nombre es {nombre} y tienes {edad}
años".format(**locals())

# 'Tu nombre es Juan y tienes 20 años'
```

Ya que **locals()** retorna un diccionario con los nombres de los objetos y sus respectivos valores.

Volviendo al nuevo sistema, también incluye las características de **str.format** con la misma sintaxis.

```
for nombre in ("Juan", "Pedro", "Francisco"):
      print(f"{nombre:>9}")


     Juan
    Pedro
```

```
Francisco
```

Template

Esta es una forma menos potente pero que te puede resolver problemas concretos de forma sencilla. En este caso, lo que se usa es el símbolo del dólar, **$**, para poder usar variables dentro del texto.

```
t = string.Template("Hola, ¿cómo estás, $name?")
print(t.substitute(name = "Leia"))

# 'Hola, ¿cómo estás, Leia'
```

Fecha y hora

Python viene con una variedad de objetos útiles que se pueden usar de inmediato. Los objetos de fecha son ejemplos de tales objetos. Los tipos de fecha son difíciles de manipular desde cero, debido a la complejidad de fechas y horas. Sin embargo, los objetos de fecha de Python facilitan enormemente la conversión de fechas en los formatos de cadena deseables.

El formateo de fechas es una de las tareas más importantes a las que te enfrentarás como programador. Las diferentes regiones del mundo tienen diferentes formas de representar fechas / horas, por lo tanto, su objetivo como programador es presentar los valores de fecha de una manera que los usuarios puedan leer.

Por ejemplo, es posible que deba representar un valor de fecha numéricamente como “02-23-2018”. Por otro lado, es posible que deba escribir el mismo valor de fecha en un formato de texto más largo, como “23 de febrero de 2018”. En otro escenario, es posible que desee extraer el mes en formato de cadena de un valor de fecha con formato numérico.

Python datetime El módulo, como probablemente adivinó, contiene métodos que pueden usarse para trabajar con valores de fecha y hora. Para usar este módulo, primero lo importamos a través del import declaración de la siguiente manera:

```
import datetime
```

Podemos representar valores de tiempo usando el ***time clase***. Los atributos de la time la clase incluye la hora, minuto, segundo y microsegundo.

Los argumentos a favor del ***time*** las clases son opcionales. Aunque si no especifica ningún argumento obtendrá un tiempo de 0, que es poco probable que sea lo que necesita la mayor parte del tiempo.

Por ejemplo, para inicializar un objeto de tiempo con un valor de 1 hora, 10 minutos, 20 segundos y 13 microsegundos, podemos ejecutar el siguiente comando:

```
t = datetime.time(1, 10, 20, 13)
```

Para ver la hora, usemos la función print():

Salida:

```
01:10:20.000013
```

Es posible que necesite ver la hora, el minuto, el segundo o el microsegundo solamente, así es como puede hacerlo:

```
print('hour', t.hour)
```

Salida:

```
hour: 1
```

Los minutos, segundos y microsegundos del tiempo anterior se pueden recuperar de la siguiente manera:

```
print('Minutes', t.minute)
print('Seconds', t.second)
print('Microsecond', t.microsecond)
```

Salida:

```
Minutes: 10
Seconds: 20
Microseconds: 13
```

Los valores para la fecha del calendario se pueden representar mediante el date clase. Las instancias tendrán atributos para año, mes y día.

Llamemos al today método para ver la fecha de hoy:

```
import datetime


today = datetime.date.today()
print(today)
```

Salida: sería la fecha actual, con el siguiente formato.

```
2023-02-14
```

El código devolverá la fecha de hoy, por lo tanto, el resultado que vea dependerá del día en que ejecute el script anterior.

Ahora llamemos al ctime método para imprimir la fecha en otro formato:

```
print('ctime', today.ctime())
```

Salida

```
ctime Tue Feb 14 00:00:00 2023
```

los ***ctime*** El método usa un formato de fecha y hora más largo que los ejemplos que vimos antes. Este método se utiliza principalmente para convertir el tiempo Unix (el número de segundos desde el 1 de enero de 1970) a un formato de cadena.

Y así es cómo podemos mostrar el año, el mes y el día usando el date clase:

```
print('Year', today.year)
print('Month', today.month)
print('Day', today.day)
```

Salida

```
Year 2023

Month 2

Day 14
```

Conversiones

Fechas en cadenas con strtime

Ahora que sabe cómo crear objetos de fecha y hora, aprendamos a formatearlos en cadenas más legibles.

Para lograr esto, usaremos el ***strftime*** método. Este método nos ayuda a convertir objetos de fecha en cadenas legibles. Toma dos parámetros, como se muestra en la siguiente sintaxis:

```
time.strftime(format,t)
```

El primer parámetro es la cadena de formato, mientras que el segundo parámetro es la hora de formatear, que es opcional.

Este método también se puede utilizar en ***datetime*** objeto directamente, como se muestra en el siguiente ejemplo:

```
import datetime


x=datetime.datetime(2023,2,14)
print(x.strftime("%b %d %Y %H: %M: %S"))
```

Salida

```
Feb 14 2023 00: 00: 00
```

Hemos utilizado las siguientes cadenas de caracteres para formatear la fecha:

- %b: Devuelve los primeros tres caracteres del nombre del mes. En nuestro ejemplo, devolvió “Sep”
- %d: Devuelve el día del mes, del 1 al 31. En nuestro ejemplo, devolvió “15”.
- %Y: Devuelve el año en formato de cuatro dígitos. En nuestro ejemplo, devolvió “2018”.
- %H: Devuelve la hora. En nuestro ejemplo, devolvió “00”.
- %M: Devuelve el minuto, de 00 a 59. En nuestro ejemplo, devolvió “00”.
- %S: Devuelve el segundo, de 00 a 59. En nuestro ejemplo, devolvió “00”.

No pasamos un tiempo, por lo tanto, los valores de tiempo son todos “00”. El siguiente ejemplo muestra cómo se puede formatear la hora también:

```
import datetime


x=datetime.datetime(2023,2,14, 15,21,33)
print(x.strftime("%b %d %Y %H: %M: %S"))
```

Salida

```
Feb 14 2023 15: 21: 33
```

Cadenas a fechas con strtime

los ***strftime*** El método nos ayudó a convertir objetos de fecha en cadenas más legibles. los ***strptime*** El método hace lo contrario, es decir, toma cadenas y las convierte en objetos de fecha que Python puede entender.

Aquí está la sintaxis del método:

```
datetime.strftime(string,format)
```

El parámetro **string** es el valor en formato de cadena que queremos convertir en formato de fecha. los format parámetro es la directiva que especifica el formato que debe tomar la fecha posterior a la conversión.

Por ejemplo, digamos que necesitamos convertir la cadena “15/9/18” en una **datetime** objeto.

Primero importamos el **datetime** módulo. Usaremos el **from** palabra clave para poder hacer referencia a las funciones específicas del módulo sin el formato de punto:

```
from datetime import datetime
```

Luego podemos definir la fecha en forma de cadena:

```
str = "11/10/93"
```

Python no podrá entender la cadena anterior como una fecha y hora hasta que la convierta en una ***datetime*** objeto. Podemos hacerlo con éxito llamando al ***strptime*** método.

Ejecute el siguiente comando para convertir la cadena:

```
date_object = datetime.strptime(str, '%m/%d/%y')
```

Llamemos ahora al print función para mostrar la cadena en datetime formato:

```
print(date_object)
```

Salida

```
1993-11-10 00:00:00
```

Como puede ver, la conversión se realizó correctamente.

Puede ver que la barra diagonal “/” se ha utilizado para separar los distintos elementos de la cadena. Esto le dice al strptime método en qué formato está nuestra fecha, que en nuestro caso “/” se usa como separador.

Pero, ¿y si el día / mes / año estuviera separado por un “-“? Así es cómo manejarías eso:

```
from datetime import datetime
```

```
str = "11-10-93"

date_object = datetime.strptime(str, '%m-%d-%y')

print(date_object)
```

Salida

```
1993-11-10 00:00:00
```

Expresiones Regulares

Las expresiones regulares son expresiones ***comodín*** que definen patrones de caracteres a emparejar y extraer de una cadena de texto. Para entenderlo con un ejemplo, definamos «**Python**» como nuestra expresión regular y **«Programa en Python**» como nuestra cadena de texto. En este caso, nuestra cadena de texto contiene una instancia de nuestra expresión regular. Si bien este ejemplo ilustrativo puede resultar sencillo, en este post veremos como las expresiones regulares se ayudan de caracteres especiales para poder emparejar cualquier patrón de texto, pudiendo llegar a considerarse un lenguaje en sí mismas. También cabe mencionar que las expresiones regulares no son una característica exclusiva de Python, ya que también están presentes en otros lenguajes de programación como Java, JavaScript, Ruby, entre otros.

Componentes de las expresiones Regulares.

Literales:

Cualquier carácter se encuentra a sí mismo, a menos que se trate de un *metacaracter* con significado especial. Una serie de caracteres encuentra esa misma serie en el texto de entrada, por lo tanto la plantilla "raul" encontrará todas las apariciones de "raul" en el texto que procesamos.

Secuencia de escape:

La sintaxis de las expresiones regulares nos permite utilizar las secuencias de escape que ya conocemos de otros lenguajes de programación para esos casos especiales como ser finales de línea, tabs, barras diagonales, etc. Las principales secuencias de escape que podemos encontrar, son:

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

Métodos para usar expresiones regulares

El uso de las expresiones regulares en Python viene dado por el paquete **re**, que hay que importar a nuestro código. Algunos de los métodos proporcionados en este paquete son:

- **re.search(***patrón, cadena*): busca la primera ocurrencia de la expresión regular definida en *patrón* dentro del *string cadena*. El resultado se devuelve en un objeto Match en caso de que exista tal ocurrencia, o en un objeto *None* en caso contrario. Siguiendo con el ejemplo anterior:

```
import re
re.search("Python", "Programa en Python")

<re.Match object; span=(12, 18), match='Python'>
```

- **re.findall(***patrón, cadena***)**: devuelve una lista que contiene todas las ocurrencias de la expresión regular definida en *patrón* dentro del **string *cadena***. Las ocurrencias se devuelven en el mismo orden en que se han encontrado.

```
import re
re.findall("Python", "Aprende con Alf es mi blog
favorito de Python, y gracias a él me estoy
convirtiendo en todo un Pythonista")

['Python', 'Python']
```

- **re.split**(*separador, cadena*): divide la cadena tomando en cuenta las ocurrencias del separador. El resultado se devuelve en una lista.

```
re.split("@", "nombre.apellido@ejemplo.tld")
['nombre.apellido', 'ejemplo.tld']
```

Caracteres para definir expresiones regulares.

Hasta ahora hemos visto ejemplos muy simples del uso de las expresiones regulares. En este apartado vamos a ver los distintos caracteres o *comodines* que podemos utilizar para formar patrones de búsqueda más complejos.

Listas

Las listas en Python son un tipo contenedor, compuesto, que se usan para almacenar conjuntos de elementos relacionados del mismo tipo o de tipos distintos.

Puede ser:

- **Heterogéneas**: pueden estar conformadas por elementos de distintos tipo, incluidos otras listas.
- **Mutables**: sus elementos pueden modificarse.

El tipo de dato lista tiene algunos métodos más. Aquí están todos los métodos de los objetos lista:

List.append(x)

Agrega un ítem al final de la lista. Equivale a a[len(a):] = [x].

list.extend(iterable)

Extiende la lista agregándole todos los ítems del iterable. Equivale a **a[len(a):] = iterable**.

list.insert(i, x)

Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto **a.insert(0, x)** inserta al principio de la lista y **a.insert(len(a), x)** equivale a **a.append(x)**.

list.remove(x)

Quita el primer ítem de la lista cuyo valor sea x. Lanza un ValueError si no existe tal ítem.

list.pop([i])

Quita el ítem en la posición dada de la lista y lo retorna. Si no se especifica un índice, **a.pop()** quita y retorna el último elemento de la lista. (Los corchetes que encierran a i en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)

list.clear()

Elimina todos los elementos de la lista. Equivalente a del **a[:]**.

list.index(x[, start[, end]])

Retorna el índice basado en cero del primer elemento cuyo valor sea igual a x. Lanza una excepción ValueError si no existe tal elemento.

Los argumentos opcionales *start* y *end* son interpretados como la notación de rebanadas y se usan para limitar la búsqueda a un segmento

particular de la lista. El índice retornado se calcula de manera relativa al inicio de la secuencia completa en lugar de con respecto al argumento start.

list.count(x)

Retorna el número de veces que x aparece en la lista.

list.sort(, key=None, reverse=False)*

Ordena los elementos de la lista in situ (los argumentos pueden ser usados para personalizar el orden de la lista, ver **sorted()** para su explicación).

list.reverse()

Invierte los elementos de la lista in situ.

list.copy()

Retorna una copia superficial de la lista. Equivalente a **a[:]**.

Un ejemplo que usa la mayoría de los métodos de la lista:

```
fruits = ['orange', 'apple', 'pear', 'banana',
'kiwi', 'apple', 'banana']

fruits.count('apple')
# 2

fruits.count('tangerine')
# 0

fruits.index('banana')
# 2
```

```
fruits.index('banana',4) # Encuentra la siguiente
banana comenzando en la posición 4
# 6

fruits.reverse()
print(fruits)
# ['banana', 'apple', 'kiwi', 'banana', 'pear',
'apple', 'orange']

fruits.append('grape') # Agrega el elemento a la
lista

fruits.sort()
print(fruits)
# ['apple', 'apple', 'banana', 'banana', 'grape',
'kiwi', 'orange', 'pear']

fruits.pop()
# 'orange'
```

Los métodos como *insert*, *remove* o *sort* que únicamente modifican la lista no tienen impreso un valor de retorno – retorna el valor por defecto None. Esto es un principio de diseño para todas las estructuras de datos mutables en Python.

Otra cosa que puedes observar es que no todos los datos se pueden ordenar o comparar. Por ejemplo, [None, 'hello', 10] no se puede ordenar ya que los enteros no se pueden comparar con strings y *None* no se puede comparar con los otros tipos. También hay algunos tipos que no tienen una relación de orden definida. Por ejemplo, 3+4j < 5+7j no es una comparación válida.

Listas y String

A partir de una cadena de caracteres, podemos obtener una lista con sus componentes usando la función **split**.

Si queremos obtener las palabras (separadas entre sí por espacios) que componen la cadena **xs** escribiremos simplemente **xs.split()**:

```
c = " Una cadena con espacios "
c.split()
# ['Una', 'cadena', 'con', 'espacios']
```

En este caso **split** elimina todos los blancos de más, y devuelve sólo las palabras que conforman la cadena.

Si en cambio el separador es otro carácter (por ejemplo la arroba, @), se lo debemos pasar como parámetro a la función **split**. En ese caso se considera una componente todo lo que se encuentra entre dos arrobas consecutivas. En el caso particular de que el texto contenga dos arrobas una a continuación de la otra, se devolverá una componente vacía:

```
d = "@@Una@@@cadena@@@con@@espacios@"
d.split("@")
# ['', '', 'Una', '', '', 'cadena', '', '', 'con',
'', 'espacios', '']
```

La *casi–inversa* de split es una función join que tiene la siguiente sintaxis:

```
<separador>.join(<lista de componentes a unir>)
```

y que devuelve la cadena que resulta de unir todas las componentes separadas entre sí por medio del *separador*:

```
xs = ['aaa', 'bbb', 'cccc']
" ".join(xs) # ojo! hay que agregar el espacio
# 'aaa bbb cccc'


", ".join(xs)
# 'aaa, bbb, cccc'


"@@".join(xs)
# 'aaa@@bbb@@cccc'
```

Usando listas como pilas

Los métodos de lista hacen que resulte muy fácil usar una lista como una pila, donde el último elemento añadido es el primer elemento retirado («último en entrar, primero en salir»). Para agregar un elemento a la cima de la pila, utiliza **append**(). Para retirar un elemento de la cima de la pila, utiliza **pop**() sin un índice explícito. Por ejemplo:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack

# [3, 4, 5, 6, 7]

stack.pop()

# 7

stack
# [3, 4, 5, 6]

stack.pop()

# 6
```

```
stack.pop()

# 4

stack
# [3, 4]
```

Usando listas como colas

También es posible usar una lista como una cola, donde el primer elemento añadido es el primer elemento retirado («primero en entrar, primero en salir»); sin embargo, las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento (porque todos los otros elementos tienen que ser desplazados por uno).

Para implementar una cola, utiliza **collections.deque** el cual fue diseñado para añadir y quitar de ambas puntas de forma rápida. Por ejemplo:

```
from collections import deque

queue = deque(["Eric", "John", "Michael"])

queue.append("Terry")

queue.append("Graham")

queue.popleft()

# 'Eric'

queue.popleft()

# 'John'

print(queue)

# deque(['Michael', 'Terry', 'Graham'])
```

Comprensión de listas

Las comprensiones de listas ofrecen una manera concisa de crear listas. Sus usos comunes son para hacer nuevas listas donde cada elemento es el resultado de algunas operaciones aplicadas a cada miembro de otra secuencia o iterable, o para crear un segmento de la secuencia de esos elementos para satisfacer una condición determinada.

Por ejemplo, asumamos que queremos crear una lista de cuadrados, como:

```
squares = []

for x  in range(10):
      squares.append(x**2)
squares

# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Nota que esto crea (o sobreescribe) una variable llamada x que sigue existiendo luego de que el bucle haya terminado. Podemos calcular la lista de cuadrados sin ningún efecto secundario haciendo:

```
squares = list(map(lambda x: x**2, range(10)))
```

o, un equivalente:

```
squares = [x**2 for x in range(10)]
```

Que es más conciso y legible.

Una lista de comprensión consiste de corchetes rodeando una expresión seguida de la declaración for y luego cero o más declaraciones for o if. El resultado será una nueva lista que sale de evaluar la expresión en el contexto de los for o if que le siguen. Por ejemplo, esta lista de comprensión combina los elementos de dos listas si no son iguales:

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] is x!= y]

[(1, 3),(1, 4),(2, 3),(2, 1),(2, 4),(3, 1),(3, 4)]
```

y es equivalente a:

```
combs = []

for x in [1, 2, 3]:
      for y in [3, 1, 4]:
            if x!= y:
                  combs.append((x, y))

combs

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Nota como el orden de los **for y if** es el mismo en ambos pedacitos de código.

Si la expresión es una tupla (como el **(x, y)** en el ejemplo anterior), debe estar entre paréntesis.

Las comprensiones de listas pueden contener expresiones complejas y funciones anidadas:

```
from math import pi

[str(round(pi, i)) for i in range(1, 6)]
# ['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Ordenamiento

sort()

La forma más sencilla de ordenar una lista en Python es utilizar el método **sort**() de la clase **list**.

Este método modifica la propia lista, ordenando los elementos de manera ascendente (el método siempre devuelve **None**):

```
numeros = [16, 4, 9, 1, 3, 20, 8]
numeros.sort()
numeros

# [1, 3, 4, 8, 9, 16, 20]


palabras = ["hola", "coche", "avión", "manzana", "perro",
"gato"]
palabras.sort()
palabras

# ['avión', 'coche', 'gato', 'hola', 'manzana', 'perro']
```

Para listas que contienen elementos de tipos heterogéneos, llamar a la función **sort**() provocará que el intérprete lance un error.

```
lista = [1, "a", 5, "g", 3, "c"]
lista.sort()

# TypeError: '<' not supported between instances of 'str'
and 'int'
```

sorted()

La función incorporada **sorted**() que se puede utilizar para ordenar cualquier *iterable*, no solo listas. Esta función es muy parecida al método sort() pero, a diferencia de este, sí que devuelve una nueva lista.

Ejemplo:

```
numeros = [16, 4, 9, 1, 3, 20, 8]
ordenados = sorted(numeros)
ordenados

# [1, 3, 4, 8, 9, 16, 20]

numeros

# [16, 4, 9, 1, 3, 20, 8]
```

Al igual que el método **sort**(), llamar a **sorted**() con una lista heterogénea de elementos hará que el intérprete lance un error.

Por defecto, tanto **sort**() como **sorted**() ordenan los elementos de manera ascendente, para lo cuál, aplican el operador **<**.

Matrices

Cuando necesitamos manejar muchos datos, generalmente hay soluciones más efectivas que tener muchas variables. Por ejemplo, si hay que guardar 100 números, suele ser más eficiente almacenar esos 100 datos "**juntos**", formando una "**matriz**", en vez de usar 100 variables distintas.

La palabra "**matriz**" es una traducción del inglés "**array**". Algunos autores lo traducen alternativamente como tabla, vector o incluso "arreglo".

Normalmente, en una matriz podremos acceder individualmente a cada uno de sus elementos usando corchetes: el primer dato sería algo como "**datos[0]**", y el último de 10 elementos sería "**datos[9]**".

Vamos a ver un ejemplo que prepare un array con 6 elementos, nos pida 6 datos y luego los muestre en orden contrario al que se han introducido:

```
datos = [0, 0, 0, 0, 0, 0]

for i in range (1, 7):

      datos[i - 1] = int(input("Dime el dato, número {}: 
".format(i)))

print("Los datos al revés son: ")

for i in range(6, 0, -1):

      print(datos[i-1])
```

En Python, los arrays pueden ir aumentando de tamaño (comparado con otros lenguajes de programación, en este sentido se parecen más a lo que en muchos lenguajes de programación se conoce como "**listas**"). Así, podemos partir de un array vacío e ir añadiendo elementos con ".**append**":

```
datos = []

for i in range (1, 7):
      nuevoDato = int(input("Dime el dato, número {}:
".format(i)))

print("Los datos al revés son: ")

for i in range(6, 0, -1):

      print(nuevoDato[i-1])

# TypeError: 'int' object is not subscriptable
```

Nota: No podemos crear el array vacío con "**datos = []**" y luego dar valor a un elemento con "**datos[0]=5**", porque obtendremos un mensaje de error que nos avisa de que nos hemos salido del rango de los datos. Deberemos reservar todas las o bien usar "**.append**".

También podemos saber la cantidad de datos con "**len(datos)**", eliminar un elemento con "**.remove**", insertar en una cierta posición con "**.insert**", o añadir toda una nueva lista de datos con "**+**".

Si un array va a contener muchos más datos, puede resultar incómodo eso de dar los valores iniciales uno por uno, y quizá no podamos emplear "**.append**" si los valores no los vamos a recibir exactamente en orden. En ese caso, se puede inicializar el array usando una orden "**for**" dentro de los corchetes, así:

```
datos = [0 for x in range(20)]

for i in range (0, len(datos)):
      datos[i] = int(input("Dime el dato, número {}:
".format(i+1)))

print("Los datos al revés son: ")

for i in range(len(datos),0,-1):

      print(datos[i-1])
```

Una forma más avanzada de crear un (falso) array vacío, para luego irlo rellenando, es usar unas llaves vacías, como en "**datos = { }**", aunque esto tiene un significado ligeramente distinto.

```
datos = {}

for i in range (1,7):
      datos[i-1] = int(input("Dime el dato, número {}:
".format(i)))

print("Los datos al revés son: ")

for i in range(6,0,-1):

      print(datos[i-1])
```

Para crear una matriz de dos dimensiones (realmente, una "lista de listas"), la alternativa más sencilla es indicar todos los valores iniciales, incluso aunque fueran cero todos ellos (pero pueden no serlo, o incluso ser de tipos de datos distintos):

```
datos = [
       ["uno", 2],
       ["a", "b", "c"]
]
print(datos)
```

Y si la matriz es de un tamaño grande, puede ser más cómodo crearla y rellenarla de forma repetitiva, así:

```
datos = [[ 0 for columna in range(0,5)] for fila in range
(0,4)]
datos[0][2] = 5
print(datos)
```

Tuplas

Se ha visto que las listas y cadenas tienen propiedades en común, como el indexado y las operaciones de seccionado. Estas son dos ejemplos de datos de tipo secuencia. Como Python es un lenguaje en evolución, otros datos de tipo secuencia pueden agregarse. Existe otro dato de tipo secuencia estándar: la tupla.

Una tupla consiste de un número de valores separados por comas, por ejemplo:

```
t = 12345, 54321, 'hello!'
t[0]

# 12345
```

No es posible asignar a los ítems individuales de una tupla, pero sin embargo sí se puede crear **tuplas** que contengan objetos mutables, como las listas.

A pesar de que las tuplas puedan parecerse a las listas, frecuentemente se utilizan en distintas situaciones y para distintos propósitos. Las tuplas son immutable y normalmente contienen una secuencia heterogénea de elementos que son accedidos al desempaquetar o indizar. Las listas son mutable, y sus elementos son normalmente homogéneos y se acceden iterando a la lista.

Un problema particular es la construcción de tuplas que contengan 0 o 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor (no alcanza con encerrar un único valor entre paréntesis). Feo, pero efectivo. Por ejemplo:

```
empty = ()

singleton = 'hello',
```

```
len(empty)

# 0

len(singleton)

# 1

singleton

# ('hello',)
```

La declaración t = 12345, 54321, 'hola!' es un ejemplo de ***empaquetado de tuplas***: los valores 12345, 54321 y 'hola!' se empaquetan juntos en una tupla. La operación inversa también es posible:

```
x, y, z = t
```

Esto se llama, apropiadamente, ***desempaquetado*** *de secuencias*, y funciona para cualquier secuencia en el lado derecho del igual. El desempaquetado de secuencias requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la secuencia. Notá que la asignación múltiple es en realidad sólo una combinación de empaquetado de tuplas y desempaquetado de secuencias.

Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados. A esta operación se la denomina *empaquetado de tuplas*.

```
a = 125
b = "\#"
c = "Ana"
d = a, b, c

len(d)
# 3

d
# (125, '\#', 'Ana')
```

Si se tiene una tupla de longitud k, se puede asignar la tupla a k variables distintas y en cada variable quedará una de las componentes de la tupla. A esta operación se la denomina *desempaquetado de tuplas*.

```
x, y, z = d

x
# 125

y
# "\#"

z
# "Ana"
```

Diccionario

Un diccionario es una estructura de datos y un tipo de dato en Python con características especiales que nos permite almacenar cualquier tipo de valor como enteros, cadenas, listas e incluso otras funciones. Estos diccionarios nos permiten además identificar cada elemento por una clave (Key).

Para definir un diccionario, se encierra el listado de valores entre llaves. Las parejas de clave y valor se separan con comas, y la clave y el valor se separan con dos puntos.

```
diccionario = {'nombre': 'Carlos', 'edad': 22, 'cursos':
['Python', 'Django', 'JavaScript']}
```

Podemos acceder al elemento de un Diccionario mediante la clave de este elemento, como veremos a continuación

```
print(diccionario['nombre']) #'Carlos'
print(diccionario['edad']) #22
print(diccionario['cursos']) #['Python', 'Django',
'JavaScript']
```

También es posible insertar una lista dentro de un diccionario. Para acceder a cada uno de los cursos usamos los índices:

```
print(diccionario['cursos'])[0] #'Python'
print(diccionario['cursos'])[1] #'Django'
print(diccionario['cursos'])[2] #'JavaScript'
```

Para recorrer todo el Diccionario, podemos hacer uso de la estructura for:

```
for key in diccionario:
      print(key, ":", diccionario[key])
```

Métodos de los diccionarios

dict()

Recibe como parámetro una representación de un diccionario y si es factible, devuelve un diccionario de datos.

```
dic = dict(nombre = 'nestor', apellido = 'Plasencia', edad
= 22)
```

Zip()

Recibe como parámetro dos elementos iterables, ya sea una cadena, una lista o una tupla. Ambos parámetros deben tener el mismo número de elementos. Se devolverá un diccionario relacionando el elemento i-esimo de cada uno de los iterables.

```
dic = dict(zip('abcd', [1,2,3,4]))
```

items()

Devuelve una lista de tuplas, cada tupla se compone de dos elementos: el primero será la clave y el segundo, su valor.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.items()
```

Keys()

Retorna una lista de elementos, los cuales serán las claves de nuestro diccionario.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.keys()
```

Values()

Retorna una lista de elementos, que serán los valores de nuestro diccionario.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.values()
```

clear()

Elimina todos los ítems del diccionario dejándolo vacío

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.clean()
```

copy()

Retorna una copia del diccionario original.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.copy()
```

fromkeys()

Recibe como parámetros un iterable y un valor, devolviendo un diccionario que contiene como claves los elementos del iterable con el mismo valor ingresado. Si el valor no es ingresado, devolverá none para todas las claves.

```
dic = dict.fromkeys(['a', 'b', 'c', 'd'], 1]
```

Get()

Recibe como parámetro una clave, devuelve el valor de la clave. Si no lo encuentra, devuelve un objeto none.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dict.get('b')
```

pop()

Recibe como parámetro una clave, elimina esta y devuelve su valor. Si no lo encuentra, devuelve el error.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.pop('b')
```

setdefault()

Funciona de dos formas. En la primera como get

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.setdefault('a')
```

Y en la segunda forma, nos sirve para agregar un nuevo elemento a nuestro diccionario.

```
dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
items = dic.setdefault('e',5)
```

Update()

Recibe como parámetro otro diccionario. Si se tienen claves iguales, actualiza el valor de la clave repetida; si no hay claves iguales, este par clave-valor es agregado al diccionario.

```
dic1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
dic2 = {'c': 6, 'b': 5, 'e': 9, 'f': 10}
dic1.update(dic 2)
```

Iteración de diccionarios

Cuando iteramos sobre diccionarios, se pueden obtener al mismo tiempo la clave y su valor correspondiente usando el método **items**().

```
knights = {'gallahad':'the pure', 'robin': 'the brave'}
```

```
for k, v in knights.items():
      print(k, v)

gallahad the pure
robin the brave
```

Cuando se itera sobre una secuencia, se puede obtener el índice de posición junto a su valor correspondiente usando la función **enumerate**().

```
 for i, v in enumerate(['tic','tac','toe']):
      print(i,v)

0 tic

1 tac

2 toe
```

Para iterar sobre dos o más secuencias al mismo tiempo, los valores pueden emparejarse con la función **zip**().

Referencias

[1] Curso Python para principiantes https://www.youtube.com/watch?v=chPhlsHoEPo

[2] Métodos String https://www.freecodecamp.org/espanol/news/metodos-de-string-de-python-explicados-con-ejemplo/

[3] PEP 3101 — Advanced String Formatting. https://www.python.org/dev/peps/pep-3101/ https://www.recursospython.com/pep8es.pdf

[4] Fecha y Hora
https://codigofacilito.com/articulos/fechas-python

[5] Listas y cadenas
https://uniwebsidad.com/libros/algoritmos-python/capitulo-7/listas-y-cadenas
https://j2logo.com/python/tutorial/tipo-list-python/

[6] Matrices
https://www.cartagena99.com/recursos/alumnos/apuntes/introduccion%20matrices.pdf

[7] Métodos de búsqueda
https://uniwebsidad.com/libros/python/capitulo-7/metodos-de-busqueda-2

[8] Expresiones Regulares
https://relopezbriega.github.io/blog/2015/07/19/expresiones-regulares-con-python/

[9] Pilas y Colas
http://conocepython.blogspot.com/p/pilas-y-colas.html