Tabla de contenidos

1. Contenidos mínimos IIC1103
2. Tipos de datos
3. Funciones
4. Archivos
5. Objetos
6. Profundización
   (a) Más sobre Listas
      i. Listas por comprensión
      ii. *Slicing* de listas
   (b) Más sobre *Strings*
      i. Secuencia de escape
      ii. Métodos disponibles en *Strings*
      iii. Uso de variables dentro de un *String*
      iv. Bonus: Mejorando la impresión de los *Strings*

Contenidos mínimos IIC1103

El curso IIC1103 - Introducción a la programación es requisito para este curso. Es por eso, que para este curso se espera y asume tener conocimiento sobre los siguientes tópicos:

1. Resolución de problemas con algoritmos.
2. Tipos de datos, variables, operadores y expresiones.
3. Control de flujo condicional.
4. Control de flujo cíclico.
5. Funciones importadas, propias y recursivas.
6. Manejo de strings.
7. Listas simples y listas de listas.
8. Abrir y crear archivos.
9. Programación orientada a objetos.
10. Ordenamiento.

Aquí te dejaremos un resumen de los contenidos mínimos más importantes a saber en el lenguaje Python.

Tipos de datos

Integer y float

Representan números enteros (`int`) y números decimales (`float`), y nos permiten trabajar con operadores aritméticos.

Ejemplo:

```
x = 10
y  = 20
z = x + y # 30
```

String

Es un objeto que representa cadenas de caracteres y nos permite trabajar con texto.

Ejemplo:

```
s = 'Bienvenidos a IIC2233'
```

Boolean

Representan el valor de verdad de una expresión. Por ejemplo, la expresión resultante de una operación de comparación 1 >= 0 es Verdadera o `True`, mientras que la expresión 10 == 5 es Falsa o `False`.

Ejemplo:

```
a = True
b = 3 > 4
print(a == b) # False
```

Listas

Colección finita de elementos ordenados, los cuales usualmente son otros tipos de datos como los vistos anteriormente. Las listas son mutables y su contenido puede ser accedido utilizando el índice correspondiente al orden en que se encuentra en la lista.

Ejemplo:

```
lista = [10, 'variable', True, 5.5]
print(lista[0]) # 10
```

Funciones

Las funciones permiten encapsular código que resuelve una tarea específica. Existen funciones que vienen con python y se denominan *built-in functions*. Pueden ver en la documentación oficial una lista con todas ellas. Además de las funciones que están siempre presentes, podemos incorporar nuevas funcionalidades mediante la importación de **módulos**. Los módulos proveen diversas funciones agrupadas.

Importar módulo completo

```
import modulo

modulo.funcion_1(parametro_1, parametro_2, ..., parametro_n)

modulo.funcion_2(parametro_1, parametro_2, ..., parametro_n)
```

Importar función(es) específica(s)

```
from modulo import funcion_1, funcion_2, ..., funcion_n

funcion_1(parametro_1, parametros_2, ..., parametro_n)
```

Aliasing Es posible darle el nombre que nosotros queramos al módulo o a sus funciones. Esto se denomina *aliasing*.

‘‘‘python import modulo as alias

alias.funcion_1(parametro_1, parametro_2, ..., parametro_n)

Funciones definidas por el programador Aparte de las *built-in functions* y las funciones importadas desde módulos, podemos definir nuestras propias funciones para utilizarlas en nuestros programas. Para crear una nueva función utilizamos la *keyword* `def`, entregamos parámetros y podemos opcionalmente utilizar la *keyword* `return` para retornar un valor.

```
def nombre_funcion(parametro_1, parametro_2, ..., parametro_n):
    bloque_de_codigo_de_la_funcion
    ...
    bloque_de_codigo_de_la_funcion
    return valor_de_retorno
```

```
def sumar(numero1: int, numero2: int) -> int:
    """
    Retorna la suma de dos números enteros.

    Args:
        numero1 (int): Primer número a sumar.
        numero2 (int): Segundo número a sumar.

    Returns:
        int: Resultado de sumar numero1 y numero2.
    """
    # Calculamos la suma de los dos números recibidos
    suma = numero1 + numero2

    # Retornamos el resultado de la suma
    return suma
```

Para utilizar la función debemos llamarla, y de retornar un valor debemos almacenar este en una variable.

```
salida = nombre_funcion(entrada_1, entrada_2, ..., entrada_n)
```

Las funciones pueden recibir 0 o más parámetros y retornar 0 o más valores. Lo habitual es retornar 1 valor. Cuando se alcanza el primer `return`, la función termina.

```
# Ejemplo de uso
resultado = sumar(10,20)
print(resultado)
```

```
30
```

Finalmente, los parámetros de una función pueden tener valores por defecto, es decir, en caso de no entregar dicho parámetro cuando se llame la función, se utilizará el valor por defecto.

Para lograr esto, al momento de definir la función, se incluye el valor junto al parámetro de la siguiente forma: `parametro_1=VALOR`. Por ejemplo, haremos que la función `sumar` tenga por defecto los valores `0` para cada parámetro.

```python
def sumar(n1: int = 0, n2: int = 0) -> int:
    """Retorna la suma de dos números."""
    print("Valor n1: ", n1)
    print("Valor n2: ", n2)
    return n1 + n2

resultado = sumar()
print(resultado)
```

```
Valor n1:  0
Valor n2:  0
0
```

```python
print(sumar(10)) # Daremos solo el primer parámetro, el segundo utilizará el por defecto
```

```
Valor n1:  10
Valor n2:  0
10
```

```python
print(sumar(10, 13)) # Daremos ambos parámetros para no utilizar los por defecto
```

```
Valor n1:  10
Valor n2:  13
23
```

Archivos

Un archivo es un documento binario con un nombre y una extensión, por ejemplo `tarea1.py`. Debemos partir abriendo y cerrando el archivo.

- `open(path, modo)`: Retorna un objeto que representa al archivo en `path`. El parámetro `modo` indica lo que podemos hacer con el archivo.
  - `modo='r'`: lectura (por defecto)
  - `modo='w'`: escritura
  - `modo='a'`: *append*
- `file.close()`: Cierra el archivo.

La diferencia entre los modos escritura y *append*, es que:

- En el modo escritura, se crea un nuevo archivo en `path`. Si el archivo ya existe, borra su contenido.
- En el modo *append*, si el archivo ya existe, el contenido nuevo se agrega a continuación del ya existente.

Para leer archivos, Python provee de distintas funciones y métodos. Para las siguientes instrucciones, se asume que se guardó lo retornado por `open(path, modo)` en una variable de nombre `file`.

- `file.readline()`: Retorna la siguiente línea del archivo.
- `file.readlines()`: Retorna en una variable del tipo lista todo el contenido del archivo.

Para crear archivos, usaremos:

- `file.write(s)`: Escribe el string `s` en el archivo (**no** agrega el caracter de fin de liínea `'\n'`).
- `print(s, file=file)`: Escribe el string `s` en el archivo (**sí** agrega el caracter de fin de línea `'\n'`).

Objetos

La Programación Orientada a Objetos es un paradigma de programación que consiste en modelar un problema preocupándose de los **objetos** que en él participan: sus atributos, comportamiento e interacción. En el área de desarrollo de software, un **objeto** es una colección de **datos** que además tiene **comportamientos** asociados. Por una parte, los datos **describen** a los objetos, mientras que los comportamientos **representan acciones** que ocurren en ellos.

La **Programación orientada a objetos** u **OOP** (*Object-oriented Programming*) es un *paradigma de programación* (una manera de programar) en el cual los programas modelan las funcionalidades a través de la interacción entre **objetos** por medio de sus datos y sus comportamientos.

En OOP los objetos son descritos de manera general mediante **clases**. Una clase describe los datos que caracterizan a un objeto; a estos datos los llamamos **atributos**. Una clase también describe los comportamientos de los objetos, y a estos comportamientos los llamamos **métodos**. Cada vez que creamos un objeto a partir de una clase, decimos que estamos *instanciando* esa clase, por lo tanto **un objeto es una instancia de una clase**.

```python
class Persona:
    """
    Clase que representa a una persona con nombre, apellido y edad.

    Atributos:
        nombre (str): Nombre de la persona.
        apellido (str): Apellido de la persona.
        edad (int): Edad de la persona. Por defecto 25.
    """

    def __init__(self, nombre: str, apellido: str, edad: int = 25) -> None:
        """
        Inicializa una nueva instancia de Persona.

        Args:
            nombre (str): Nombre de la persona.
            apellido (str): Apellido de la persona.
            edad (int, optional): Edad de la persona. Por defecto 25.
        """
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad  # Edad por defecto si no se pasa como argumento

    def __str__(self) -> str:
        """
        Devuelve una representación en string de la persona.

        Returns:
            str: Mensaje con el nombre completo de la persona.
        """
        return f'Mi nombre es {self.nombre} {self.apellido}'

    def saludar(self, otra_persona: "Persona") -> None:
        """
        Saluda a otra persona mostrando su nombre.

        Args:
```

```
            otra_persona (Persona): Instancia de Persona a quien se saluda.
        """
        print(f'Hola {otra_persona.nombre}, soy {self.nombre} y te saludo.')


persona1 = Persona("Francesca", "Aguirre")
persona2 = Persona("Renato", "Leal")
print(persona2)
persona1.saludar(persona2)


Mi nombre es Renato Leal
Hola Renato, soy Francesca y te saludo.
```

Para definir una clase en Python, usamos la *keyword* `class`. Dentro de la definición de la clase, definimos todos los atributos dentro de un método especial: el método `__init__()`. Este método se llama durante la **inicialización** de la clase, cada vez que ejecutamos:

```
persona1 = Persona("Francesca", "Aguirre")
persona2 = Persona("Renato", "Leal")
```

de esta manera todos los objetos que vayamos crear de la clase `Persona` poseerán estos atributos inicializados.

Podemos crear **métodos de la instancia**, los cuales reciben como primer argumento el nombre `self`, el cual es una referencia a la instanca de la clase sobre la que están actuando.

```
persona1.saludar(persona2)
```

estamos invocando al método `saludar`, donde `self` es una referencia al instancia `persona1` sobre la cual se accede a su atributo `nombre` y al atributo `nombre` de otra instancia de la misma clase. El nombre `self` no es un *keyword*, por lo que puede ser cualquier nombre. Sin embargo, en la práctica usar `self` se ha convertido en una convención.

Si no te acuerdas de alguno de estos contenidos, te recomendamos preguntar en las *discussions* del GitHub del curso.

Profundización

A continuación, se profundizará con más detalle en 2 contenidos que te serán de ayuda recurrentemente en el curso: *string* y listas.

Más sobre listas

Las **listas** (`list`) se utilizan para manejar datos de forma **ordenada** y **mutable**. Los contenidos pueden ser accedidos utilizando el índice correspondiente al orden en que se encuentran en la lista. El *orden* de los elementos de una lista, y *los elementos mismos* pueden cambiar mediante métodos que manipulan la lista.

Las listas también pueden ser heterogéneas, lo que significa que pueden contener objetos pertenecientes a clases o tipos de datos distintos, incluyendo otras listas. Si bien no existe ninguna restricción de Python sobre los tipos de datos de las listas, estas tienen una relación estrecha con los **arreglos** disponibles en otros lenguajes de programación, por lo que, dado que los arreglos suelen tener estas restricciones, es más común ver listas con tipos de datos homogéneos que heterogéneos.

En una lista, los elementos que se agregan usando `append` se ponen al final de la lista.

Podemos crear listas de las siguientes maneras:

```
# Creamos una lista vacía y agregamos elementos incrementalmente.
# En este caso agregamos dos tuplas al final de la lista.
lista = list()                # También puede ser con lista=[]
lista.append(2015)   # Aquí estamos agregando UN elemento, que es un entero
lista.append(2016)   # Luego de esto, la lista contiene 2 elementos que son enteros
print(lista)
print(len(lista))

# También es posible agregar los objetos explícitamente al definirla por primera vez
lista = [1, 'string', 20.5, [False,True]]
lista.append('último elemento')
print(lista)

# Extraemos un elemento usando el índice respectivo
print(lista[1])
print(len(lista))
```

```
[2015, 2016]
2
[1, 'string', 20.5, [False, True], 'último elemento']
string
5
```

A veces es necesario agregar nuevos elementos contenidos en otras listas. En estos casos resulta muy útil agregar la lista completa y no cada elemento de forma individual con `append()`. Para eso podemos utilizar el método `extend()`.

```
bandas = ['Radiohead', 'City and Colour', 'toe']
print(bandas)

nuevas_bandas = ['Young the Giant', 'Portugal. The Man', 'Twenty One Pilots']
bandas.extend(nuevas_bandas)
print(bandas)
```

```
['Radiohead', 'City and Colour', 'toe']
['Radiohead', 'City and Colour', 'toe', 'Young the Giant', 'Portugal. The Man', 'Twenty One Pilots']
```

También es posible insertar elementos en posiciones específicas mediante el método `insert(posición, elemento)`.

```
print(bandas)
bandas.insert(2, 'Of Monsters and Men')
print(bandas)
```

```
['Radiohead', 'City and Colour', 'toe', 'Young the Giant', 'Portugal. The Man', 'Twenty One Pilots']
['Radiohead', 'City and Colour', 'Of Monsters and Men', 'toe', 'Young the Giant', 'Portugal. The Man',
```

Listas por comprensión Desde el punto de vista de la lógica, la definición de comprensión es: ”Conjunto de caracteres que forman un concepto”. Así mismo, las listas por comprensión se pueden ver como listas formadas por un conjunto de objetos que cumplen con un concepto o condición en particular. En Python, podemos crear **listas por comprensión**. Esta es una forma más práctica para crear listas en pocas líneas de código.

Por ejemplo, ya tenemos la lista `bandas`, pero ahora queremos construir una lista con el largo de cada uno de los elementos. Una forma de hacerlo sería la siguiente:

```
largo_de_bandas = []

for nombre in bandas:
    largo_de_bandas.append(len(nombre))

print(largo_de_bandas)
```

```
[9, 15, 19, 3, 15, 17, 17]
```

Usando **listas por comprensión**, podemos definir lo mismo de forma más clara y concisa siguiendo la siguiente sintaxis:

```
nueva_lista = [expresión for elemento in lista]
```

```
largo_de_bandas = [len(nombre) for nombre in bandas]

print(largo_de_bandas)
```

```
[9, 15, 19, 3, 15, 17, 17]
```

La sentencia `if` se puede usar dentro de una lista por comprensión para construir la lista incluyendo solamente los elementos que cumplan una cierta condición. En el siguiente ejemplo guardaremos **los nombres de las bandas** que tengan un **largo menor a 10 caracteres**. La sintaxis al usar un `if` en **listas por comprensión** es la siguiente:

```
nueva_lista = [expresión for elemento in lista if condición]
```

```
bandas_con_nombre_corto = [nombre for nombre in bandas if len(nombre) < 10]

print(bandas_con_nombre_corto)
```

```
['Radiohead', 'toe']
```

El código anterior es equivalente a hacer lo siguiente:

```
bandas_con_nombre_corto = []
for nombre in bandas:
    if len(nombre) < 10:
        bandas_con_nombre_corto.append(nombre)

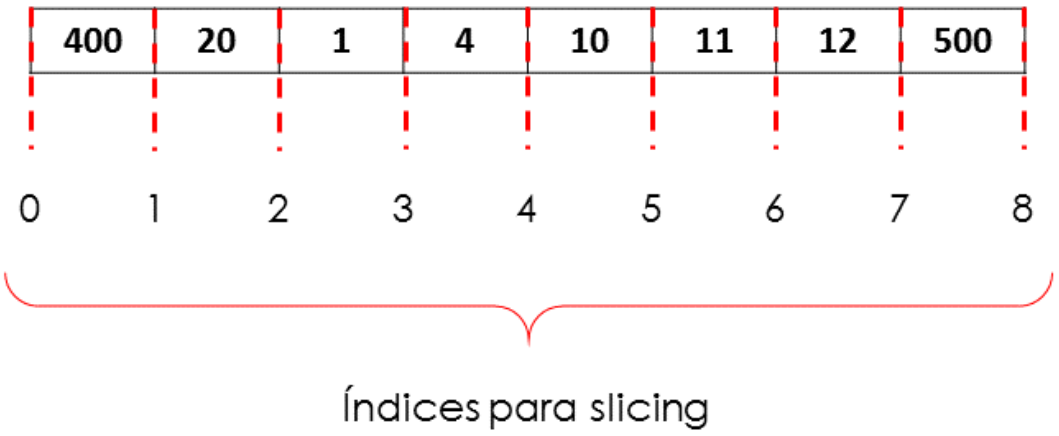
print(bandas_con_nombre_corto)
```

```
['Radiohead', 'toe']
```

Slicing* de listas** Es posible tomar secciones de la lista usando la notación de ***slicing. En esta notación, los índices indican *desde donde* y *hasta donde* deseamos recuperar datos de la lista. La sintaxis de la notación de *slicing* es:

```
secuencia[inicio:término:pasos]
```

Por defecto, el número de pasos es 1. La siguiente figura muestra un ejemplo de cómo se deben considerar los índices al usar la notación de *slicing*.



Forma general de hacer *slicing* en Python:

- `a[start:end]`: retorna los elementos desde `start` hasta `end - 1`.
- `a[start:]`: retorna los elementos desde `start` hasta el final del arreglo.
- `a[:end]`: retorna los elementos desde el principio hasta `end - 1`.
- `a[:]`: crea una copia (*shallow*) del arreglo completo. Es decir, el arreglo retornado está en una nueva dirección de memoria, pero los elementos que están en este nuevo arreglo, hacen referencia a la dirección de memoria de los elementos del arreglo inicial.
- `a[start:end:step]`: retorna los elementos desde `start` hasta no pasar `end`, en pasos de a `step`.
- `a[-1]`: retorna el último elemento en el arreglo.
- `a[-n:]`: retorna los últimos `n` elementos en el arreglo.
- `a[:-n]`: retorna todos los elementos del arreglo menos los últimos `n` elementos.

Podemos extraer un elemento específico desde una lista mediante indexación. Es posible recuperar una porción completa de la lista utilizando la notación de *slicing*.

```
# Tomando una tajada particular, en este caso desde la posición 2 hasta la anterior a 6
numeros = [6, 7, 2, 4, 10, 20, 25]
print(numeros[2:6])

# tomando una sección desde la posición 2 hasta el final de la lista
print(numeros[2:])

# tomando una sección desde el principio hasta la posición anterior a 5
print(numeros[:5:])

# lo mismo anterior, pero saltando 2 posiciones a la vez
print(numeros[:5:2])

# invirtiendo una lista
print(numeros[::-1])
```

```
[2, 4, 10, 20]
[2, 4, 10, 20, 25]
[6, 7, 2, 4, 10]
[6, 2, 10]
[25, 20, 10, 4, 2, 7, 6]
```

Las listas pueden ser ordenadas utilizando el método `sort()`. Esto ordena las listas en sí mismas (*in-place*) y no devuelve nada, es decir, el resultado no es asignable a una nueva lista.

```
numeros = [6, 7, 2, 4, 10, 20, 25]
print(numeros)

# Ordenamos en sentido ascendente.
# Observar como a no recibe ninguna asignación después de que la lista numeros es ordenada
a = numeros.sort()
print(a)
print(numeros)

# Ordenamos en sentido descendente
numeros.sort(reverse=True)
print(numeros)
```

```
[6, 7, 2, 4, 10, 20, 25]
None
[2, 4, 6, 7, 10, 20, 25]
[25, 20, 10, 7, 6, 4, 2]
```

Las listas han sido optimizadas para ser una estructura flexible y fácil de manejar. También se pueden recorrer con la notación de `for`

```
piezas = [["Alfil", 2], ["Peón", 8], ["Rey", 1], ["Reina", 1], ["Caballo", 2], ["Torre", 2]]

# Por cada recorrido en el ciclo, la variable pieza recibe un elemento de la lista,
# de acuerdo al orden de la lista
for pieza in piezas:
    # El modificador ":8" permite que el texto ocupe al menos 8 espacios en la línea
    print(f'tipo de pieza: {pieza[0]:8} - cantidad: {pieza[1]}')
```

```
tipo de pieza: Alfil    - cantidad: 2
tipo de pieza: Peón     - cantidad: 8
tipo de pieza: Rey      - cantidad: 1
tipo de pieza: Reina    - cantidad: 1
tipo de pieza: Caballo  - cantidad: 2
tipo de pieza: Torre    - cantidad: 2
```

En este ejemplo, cada elemento de la lista `piezas` es una lista que contiene 2 elementos: primero el tipo de pieza como `str`, y luego la cantidad de ellas como `int`.

Más sobre *Strings*

Son objetos que representan cadenas de caracteres y nos permiten trabajar con texto. Como tal vez sabrás, los *strings* corresponden a una secuencia inmutable de caracteres. En Python 3, todos los *strings* se representan en Unicode, codificación que permite representar virtualmente cualquier carácter en cualquier

lenguaje que veremos posteriormente con más detalle. Entonces, pensemos que en Python un *string* es una secuencia inmutable de caracteres Unicode.

```
# Ejemplo de string
cadena = "mi cadena de texto"
print(cadena[0], cadena[-1])
```

```
m o
```

Como se ve en el ejemplo anterior, podemos acceder a un elemento del *string*, pero no podemos modificarlo. Recordemos que un *string* es una ***secuencia inmutable***, por lo tanto, no es posible modificar su valor una vez creado.

```
# Intentaremos cambiar la primera letra por una mayúscula
cadena[0] = "M"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[20], line 2
      1 # Intentaremos cambiar la primera letra por una mayúscula
----> 2 cadena[0] = "M"

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

Para "modificar" un *string* hay que crear uno nuevo:

```
cadena = "Mi cadena de texto"
print(cadena)
```

```
Mi cadena de texto
```

También se pueden concatenar *strings* para formar uno nuevo.

```
otra_cadena = ""
for caracter in cadena:
    otra_cadena += caracter
print(otra_cadena)
```

```
Mi cadena de texto
```

A continuación, algunas formas distintas de crear un *string* en Python:

```
a = "programando"
b = 'mucho'
c = '''un string
con múltiples
lineas'''
d = """Multiples con
     comillas dobles"""
e = ("Tres" " strings" " juntos")
f = "un string " + "concatenado"
g = ("Otra forma de string que nos permite "
     "utilizar más de una línea pero en verdad solo es una,"
```

```
     " lo que es muy útil para cumplir PEP-8 :)")

print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
```

```
programando
mucho
un string
con múltiples
lineas
Multiples con
     comillas dobles
Tres strings juntos
un string concatenado
Otra forma de string que nos permite utilizar más de una línea pero en verdad solo es una, lo que es muy
```

Secuencia de escape El \ se denomina carácter de escape y se combina con otros caracteres para darles un nuevo significado.

Secuencia	Significado
\”	Comilla doble
\'	Comilla simple
\n	Salto de línea
\t	Tabulador
\\	Barra inversa o *Backslash*

```
a = "hola quiero escribir comillas dobles \" y seguir escribiendo"
print(a)
```

```
hola quiero escribir comillas dobles " y seguir escribiendo
```

Métodos disponibles en *Strings* La clase `str` tiene muchos métodos para manipular *strings*. La función `dir` nos permite obtener la lista:

```
print(dir(str))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '
```

Los siguientes métodos nos permiten responder preguntas sobre un *string* (una instancia) en particular:

- `str.isdigit()`: ¿Está compuesto solo por dígitos?
- `str.endswith(s)`: ¿Termina con este otro *string* `s`?
- `str.index(s)`: ¿En qué posición empieza este otro *string* `s`?

Veamos algunos ejemplos:

```python
# El método isalpha retorna True si todos los caracteres del string
# están en el alfabeto de algún lenguaje.
print("abñ".isalpha())
# Si hay algún número, espacio o puntuación dentro del string, retornará False.
print("t/".isalpha())

# El método isdigit retorna True si todos los caracteres en el string
# son dígitos numéricos.
print("34".isdigit())

# El método startswith retorna True si el string empieza exactamente
# con el substring entregado.
s = "estoy programando"
print(s.startswith("est"))
# Mientras que endswith, retorna True si termina con el substring entregado.
print(s.endswith("do"))

# Devuelve el índice donde comienza en el string la secuencia que
# se pasa como argumento.
print(s.find("y p"))

# El método index retorna el índice donde comienza la secuencia.
# Acepta dos argumentos opcionales:
# - la posición inicial donde comenzar la búsqueda y
# - la posición final (hasta dónde llega buscando)
print(s.index("y", 4, 10))
print(s.index("p", 5, 10))
```

```
True
False
True
True
True
4
4
6
```

A diferencia de los anteriores, existen otros métodos que actúan directamente sobre el *string* y retornan nuevos *strings*, sin modificar el original, ya que **los *strings* son objetos inmutables**. Algunos ejemplos útiles:

```python
string_inicial = "hola a todos, cómo están"

# Retorna una lista de strings a partir de separar el original en base a " ".
string2 = string_inicial.split(" ")
print(string2)

# Concatena todos los elementos de la secuencia dada por medio del string "#".
string3 = "#".join(string2)
print(string3)

# Entrega un nuevo string en que se reemplazó cada " " por "**".
print(string_inicial.replace(" ", "**"))

# Notar que el string inicial sigue intacto, solo obtuvimos nuevas versiones.
```

```
print(string_inicial)
```

```
['hola', 'a', 'todos,', 'cómo', 'están']
hola#a#todos,#cómo#están
hola**a**todos,**cómo**están
hola a todos, cómo están
```

Uso de variables dentro de un *String* A lo largo del curso, seguramente has visto este uso de *strings*. Los *f-strings* son una forma de formatear *strings* que permite añadir expresiones de Python directamente en un *string*. Para indicarle a Python que utilizaremos esta funcionalidad, debemos anteponer el *string* con la letra `f` (de ahí el nombre *f-string*).

Este método es muy útil para completar un *string* que estemos usando como "plantilla", para insertar valores de variables dentro de un *string*. Las ventajas de esta sintaxis son que el código queda más legible, conciso y es menos susceptible a errores, además de ser más rápida que otras formas de concatenación.

```
nombre = "Juan Pérez"
nota = 4.5

if nota >= 4.0:
    resultado = "aprobado"
else:
    resultado = "reprobado"

print(f"Hola {nombre}, estás {resultado}. Tu nota fue un {nota}.")
```

```
Hola Juan Pérez, estás aprobado. Tu nota fue un 4.5.
```

Si queremos incluir las *llaves* (`{`, `}`) dentro del *string*, podemos agregar un *escape character* que permite invocar una interpretación alternativa de los caracteres siguientes. Más concretamente, las llaves en este caso, se utilizan para encapsular la variable que queremos imprimir. Sin embargo, la representación alternativa, sería la representación literal de la llave. Luego, para este caso, si queremos imprimir una llave, esto se logrará con una doble llave.

Veamos un ejemplo. Digamos que buscamos imprimir una simple definición de una clase en Java.

```
# Con estas variables generaremos el string
clase = "MiClase"
salida = "'hola mundo'"

# En nuestra plantilla utilizamos llaves dobles cuando queremos mantenerlas...
codigo = f"""
public class {clase}
{{
       public static void main(String[] args)
       {{
           System.out.println({salida});
       }}
}}"""

# ... pero en el resultado a imprimir solo se verá una llave simple
print(codigo)
```

```
public class MiClase
{
       public static void main(String[] args)
       {
           System.out.println('hola mundo');
       }
}
```

A veces queremos incluir muchas variables dentro de un *string*, pero como vimos, eso no es un problema. Sin embargo, a veces queremos asociar estas variables dentro de objetos más complejos, que hagan sentido en nuestra modelación. Pensemos en el siguiente ejemplo, dándole forma a un correo:

```
# Las variables necesarias para crear nuestro string
from_email = "cruz@ing.puc.cl"
to_email = "alumnos@iic2233.com"
message = ("\nEste es un mail de prueba.\n"
           "\nEspero que el mensaje te sea de mucha utilidad!")
subject = "IIC2233 - Este correo es urgente"

# Nuestro f-string
print(f"""
From: <{from_email}>
To: <{to_email}>
Subject: {subject}
{message}
""")
```

```
From: <cruz@ing.puc.cl>
To: <alumnos@iic2233.com>
Subject: IIC2233 - Este correo es urgente

Este es un mail de prueba.

Espero que el mensaje te sea de mucha utilidad!
```

```
# Las variables necesarias para crear nuestro string
emails = ("a@ejemplo.com", "b@ejemplo.com")
message = {
    'subject': "Tienes un correo",
    'message': "\nEste es un correo para ti",
}

# Nuestro f-string
print(f"""
From: <{emails[0]}>
To: <{emails[1]}>
Subject: {message['subject']}
{message['message']}
""")
```

```
From: <a@ejemplo.com>
To: <b@ejemplo.com>
Subject: Tienes un correo

Este es un correo para ti
```

También podemos usar un diccionario que contenga listas e indexar la lista dentro del *string*.

```
# Las variables necesarias para crear nuestro string
message = {
    "emails": ["yo@ejemplo.com", "tu@ejemplo.com"],
    "subject": "mira este correo",
    "message": "\nSorry, no era tan importante"
}

# Nuestro f-string
print(f"""
From: <{message['emails'][0]}>
To: <{message['emails'][1]}>
Subject: {message['subject']}
{message['message']}
""")
```

```
From: <yo@ejemplo.com>
To: <tu@ejemplo.com>
Subject: mira este correo

Sorry, no era tan importante
```

Esto puede ser aún mejor: podemos pasar cualquier objeto como argumento, por ejemplo, una instancia de una clase. Luego, dentro del *string* podemos acceder a cualquiera de los atributos del objeto.

```
# Esta clase representa a un e-mail
class EMail:
    def __init__(self, from_addr: str, to_addr: str, subject: str, message: str) -> None:
        self.from_addr = from_addr
        self.to_addr = to_addr
        self.subject = subject
        self.message = message


# Creamos nuestra instancia de la clase
email = EMail(
    "a@ejemplo.com",
    "b@ejemplo.com",
    "Tienes un correo",
    "\nQue tengas un lindo día\n\nSaludos :)"
)

print(f"""
```

```
From: <{email.from_addr}>
To: <{email.to_addr}>
Subject: {email.subject}
{email.message}""")


From: <a@ejemplo.com>
To: <b@ejemplo.com>
Subject: Tienes un correo

Que tengas un lindo día

Saludos :)
```

Otro elemento muy útil de Python es el uso de `str.format()`. Si queremos usar una plantilla, es decir, un *string* que queremos usar muchas veces reemplazando elementos con distintas variables, podemos crear un *string* con una notación similar a la de *f-strings* y usar `str.format()` para hacer el reemplazo de los valores.

```
correo_1 = "a@ejemplo.com"
correo_2 = "b@ejemplo.com"

saludar_usuario = "Hola {}, te damos la bienvenida!"

print(saludar_usuario.format(correo_1))
print(saludar_usuario.format(correo_2))


Hola a@ejemplo.com, te damos la bienvenida!
Hola b@ejemplo.com, te damos la bienvenida!
```

Sin embargo, dado que podemos hacer múltiples reemplazos a través del uso de múltiples llaves dentro del *string*, podemos asignarle un nombre a cada uno de estos espacios y elegir qué valor darle a ese espacio particular a través del uso de *keyword arguments*:

```
saludo_entre_usuarios = "Hola {saludado}, te saluda el usuario {saludador}"

print(saludo_entre_usuarios.format(saludado=correo_2, saludador=correo_1))


Hola b@ejemplo.com, te saluda el usuario a@ejemplo.com
```

Bonus: Mejorando la impresión de los *Strings* También podemos mejorar el formato de los *strings* que se imprimen. Por ejemplo, en casos como la impresión de una tabla con datos, muchas veces queremos que datos pertenecientes a la misma variable se vean alineados en columnas:

```
compra = [('leche', 2, 120), ('pan', 3.5, 800), ('arroz', 1.75, 960)]


print("PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL")
for producto, precio, cantidad in compra:
    subtotal = precio * cantidad
    print(f"{producto} | {cantidad} | ${precio} | ${subtotal}")

PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL
```

```
leche | 120 | $2 | $240
pan | 800 | $3.5 | $2800.0
arroz | 960 | $1.75 | $1680.0
```

Como puedes ver, los elementos de cada línea no quedan alineados, pero Python nos permite utilizar elementos extra en la formación de *strings* para darle una forma más estructurada:

```
print("PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL")
for producto, precio, cantidad in compra:
    subtotal = precio * cantidad
    print(f"{producto:8s} | {cantidad: ^8d} | ${precio: <5.2f} | ${subtotal: >7.2f}")
```

```
PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL
leche    |   120    | $2.00  | $ 240.00
pan      |   800    | $3.50  | $2800.00
arroz    |   960    | $1.75  | $1680.00
```

Notar que, dentro de cada llave, existe un ítem tipo diccionario; es decir, antes de los dos puntos, va la variable que utilizamos normalmente en nuestro *f-string*. Después de los dos puntos, por ejemplo, `8s`, significa que el dato es un *string* de ocho caracteres. Por defecto, si el *string* es más corto que los ocho caracteres, el resto se llenará con espacios (por la derecha). Notar que también, por defecto, si el *string* que ingresamos es más largo que los 8 caracteres, este **no** será truncado:

```
compra = [('lecheeeeeeeeeeeeeee', 2, 120), ('pan', 3.5, 800), ('arroz', 1.75, 960)]


print("PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL")
for producto, precio, cantidad in compra:
    subtotal = precio * cantidad
    print(f"{producto:8s} | {cantidad: ^8d} | ${precio: <5.2f} | ${subtotal: >7.2f}")
```

```
PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL
lecheeeeeeeeeeeeee |   120    | $2.00  | $ 240.00
pan      |   800    | $3.50  | $2800.00
arroz    |   960    | $1.75  | $1680.00
```

Podemos cambiar esta situación obligando a que el *string* sea truncado si se pasa del largo máximo. Basta con agregar un punto y la precisión luego del número que indica el largo del *string*. La precisión para tipos no numéricos indica el largo máximo de caracteres usados para este campo.

```
print("PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL")
for producto, precio, cantidad in compra:
    subtotal = precio * cantidad
    print(f"{producto:8.8s} | {cantidad: ^8d} | ${precio: <5.2f} | ${subtotal: >7.2f}")
```

```
PRODUCTO | CANTIDAD | PRECIO | SUBTOTAL
lecheeee |   120    | $2.00  | $ 240.00
pan      |   800    | $3.50  | $2800.00
arroz    |   960    | $1.75  | $1680.00
```

Para la cantidad de producto el formato es `{cantidad: ^8d}`. `cantidad` corresponde a la variable de la tupla en el recorrido actual, el espacio después de los dos puntos dice que los lugares vacíos deben ser llenados con espacios (en los tipos enteros, por defecto, se llenará con ceros), el símbolo `^` es para que el número quede centrado en el espacio disponible, `8d` significa que será un entero (***d**ecimal*) de hasta ocho dígitos.

Notar que siempre el orden de estos parámetros (aunque son opcionales) debe ser de izquierda a derecha después de los dos puntos: **primero el carácter para llenar los espacios vacíos, después el alineamiento, después el tamaño y finalmente el tipo**.

Para el precio, el formato es `{precio: <5.2f}`. El dato se leerá de la variable `precio`, luego los lugares que queden libres se llenarán con espacios, el símbolo < significa que el alineamiento es a la izquierda, el formato será un número decimal (***f****loat*) de hasta cinco caracteres, con dos decimales.

De la forma equivalente, para el subtotal el formato es `{subtotal: >7.2f}`. El dato se sacará de la variable `subtotal`, el carácter de llenado será espacio, el alineamiento es a la derecha y será un *float* de siete dígitos, dos de ellos decimales, incluyendo el `.` como carácter.

Para más información sobre esta notación y los parámetros que recibe, puedes revisar la documentación entregada por Python.