

# Introducción a la Programación

Prof. Agustín Gravano

Primer semestre de 2022

Clase teórica 12: Listas (continuación)

**Atención:** Los temas nuevos presentados en esta clase **no** deben usarse en la resolución del Trabajo Práctico 1.

# Listas (List[T]) · Operaciones

Dadas las variables **a, b** de tipo List[T]; **x, y, z** de tipo T; **i** de tipo int:

- ▶ **a = list()** Crea una lista de T vacía.
- ▶ **a = []** Crea una lista de T vacía.
- ▶ **a = [x, y, z]** Crea una lista de T no vacía.
- ▶ **len(a)** Devuelve la longitud de la lista.
- ▶ **x in a** Consulta pertenencia de un elemento en la lista.
- ▶ **a.append(x)** Agrega un elemento al final de la lista.
- ▶ **a.pop()** Elimina el elemento de la última posición y lo devuelve.
- ▶ **a[i]** Lee el valor en la i-ésima posición.
- ▶ **a[i] = x** Sobreescribe la i-ésima posición.
- ▶ **a.insert(i, x)** Inserta un elemento en la i-ésima posición.
- ▶ **a.pop(i)** Elimina la i-ésima posición y devuelve su valor.
- ▶ **a == b** Compara dos listas.
- ▶ **a + b** Concatena dos listas.
- ▶ **a \* i** Concatena i veces seguidas la lista a.

Esas son algunas **operaciones básicas** de listas que ofrece Python.

Muchos otros lenguajes de programación ofrecen operaciones similares.

Python ofrece otras operaciones más complejas, que no están disponibles en muchos otros lenguajes. Veamos algunas...

## Listas (List[T]) · Índices y sublistas (*slices*)

```
lista:List[str] = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']  
len(lista)      # Devuelve 8  
lista[7]         # Último elemento  
lista[8]         # Error: índice fuera de rango  
lista[-1]        # Último elemento (!)
```

Python permite obtener fácilmente **sublistas** (o *slices*):

```
lista[i:j]        # Sublista desde i hasta j (no inclusive).  
                  # i,j pueden ser negativos, e incluso omitirse.  
lista[1:6]        # ['b', 'c', 'd', 'e', 'f']  
lista[:2]         # ['a', 'b']  
lista[-4:-1]      # ['e', 'f', 'g']  
lista[-2:]        # ['g', 'h']  
  
lista[i:j:s]       # Sublista desde i hasta j (no inclusive), en  
                  # incrementos de s. Por defecto, s vale 1.  
lista[1:6:2]       # ['b', 'd', 'f']  
lista[::4]         # ['a', 'e']  
lista[6::-2]       # ['g', 'e', 'c', 'a']
```

Los slices también funcionan sobre str: `'algoritmo'[6:2:-1] → 'tiro'`

# Listas (List[T]) · Operaciones

`lista[i:j:s]` → sublista desde *i* hasta *j* (no incl.) en incrementos de *s*

Veamos otras operaciones útiles de listas en Python.

`range(i, j, s)` → construye el rango de números enteros desde *i* hasta *j* (no inclusive), en incrementos de *s* (*i*, *s* pueden omitirse; por defecto, *i* vale 0, *s* vale 1).

```
list(range(1, 5))           # devuelve [1, 2, 3, 4]
list(range(1, 11, 3))       # devuelve [1, 4, 7, 10]
list(range(100, 0, -30))    # devuelve [100, 70, 40, 10]
list(range(5))              # devuelve [0, 1, 2, 3, 4]
```

**Detalle técnico:** `range()` devuelve un elemento de tipo `range`. Lo convertimos a lista para instanciar sus enteros (para *desplegar* el rango), pero la conversión no es necesaria en general.

**Ejercicio:** Completar la expresión `list(range(__, __, __))` para que devuelva `[-10, -5, 0, 5, 10]`.

# Operaciones de List[str] y str

Si tenemos una **lista de strings** y los queremos **unir** en un solo string, con un string **separador** entre cada par de elementos, usamos **join**.

```
colores:List[str] = ['verde', 'azul', 'rojo']  
';'.join(colores)      # devuelve 'verde;azul;rojo'  
'_#_'.join(colores)    # devuelve 'verde_#_azul_#_rojo'
```

La operación inversa es **split**.

```
'verde;azul;rojo'.split(';')  
'verde_#_azul_#_rojo'.split('_#_')  
# ambas devuelven la lista ['verde', 'azul', 'rojo']
```

**Ejercicio:** Evaluar las siguientes expresiones (pensarlas primero en papel).

```
1  x:str = 'dificilísimo'  
2  x.split('i')  
3  x.split('i')[2:]  
4  'i'.join(x.split('i')[2:])  
5  'fa' + 'i'.join(x.split('i')[2:])
```

## Listas (List[T]) · Ejemplo

```
1  def sumar(xs:List[int]) -> int:
2      '''
3      Requiere: nada.
4      Devuelve: la suma de los elementos de xs.
5      '''
6      vr:int = 0
7      i:int = 0
8      while i < len(xs):
9          vr = vr + xs[i]
10         i = i + 1
11     return vr
```

Predicado invariante del ciclo:

- ▶  $0 \leq i \leq \text{len}(xs)$
- ▶ `vr` vale la suma de los elementos de `xs`, entre las posiciones 0 e  $i - 1$  (inclusive).

# Listas (List[T]) · Iteradores

La instrucción **for** nos permite **iterar** sobre los elementos de una lista:

```
mundiales:List[str] = ['Corea-Japón', 'Alemania',  
                        'Sudáfrica', 'Brasil', 'Rusia']  
  
for m in mundiales:  
    print(m)
```

Entonces, podemos escribir sumar de esta forma:

```
1  def sumar(xs:List[int]) -> int:  
2      '''  
3      Requiere: nada.  
4      Devuelve: la suma de los elementos de xs.  
5      '''  
6      vr:int = 0  
7      for elem in xs:  
8          vr = vr + elem  
9      return vr
```

# Listas (List[T]) · Iteradores

```
1  def sumar1(xs:List[int])->int:
2      vr:int = 0
3      i:int = 0
4      while i < len(xs):
5          vr = vr + xs[i]
6          i = i + 1
7      return vr
```

```
1  def sumar2(xs:List[int])->int:
2      vr:int = 0
3      for elem in xs:
4          vr = vr + elem
5      return vr
```

Las funciones `sumar1` y `sumar2` implementan **el mismo algoritmo**: recorrer la lista `xs`, de izquierda a derecha, llevando en `vr` la suma parcial de los elementos visitados.

El código de `sumar2` es mucho más claro, gracias a que el iterador `for` elimina la necesidad del índice `i`.

**¡Pero cuidado!** ¿Cuál es el predicado invariante del ciclo de `sumar2`? Debería ser el mismo, pero ahora no tenemos la `i` para expresarlo.

**¿ $\mathcal{I}$ : `vr` vale la suma de los elementos de `xs` visitados hasta ahora?** 🤔

Ganamos claridad en el código, pero perdemos precisión en la justificación de su correctitud (que es importantísima también). **Es un fino equilibrio.**



## while vs. for · Otro ejemplo

Problema: Devolver el primer número primo que aparece en una lista.

Algoritmo: Recorrer la lista de izquierda a derecha. Para cada elemento, si es primo: devolverlo y terminar; si no, seguir.

```
1  def primer_primo_v1(xs:List[int]) -> int:
2      '''Requiere: hay al menos un número primo en la lista xs.
3          Devuelve: xs[j] para alguna posición j, tal que xs[j] es
4              primo y no hay primos en las posiciones anteriores a j.
5          '''
6      i:int = 0
7      while not es_primo(xs[i]):
8          i = i + 1
9      return xs[i]
```

$\mathcal{I}$ :  $0 \leq i \leq \text{len}(xs)-1$ , y en las posiciones anteriores a  $i$  no hay primos.

Demos: La cláusula *Requiere* garantiza la existencia de un número primo en la lista  $xs$ .  $i$  empieza en 0 y avanza de a 1. Así, es inevitable que en algún momento apunte a una posición con número primo. En ese momento se niega la condición y el ciclo termina.  $\square$

$\mathcal{I}$  vale antes del ciclo, porque  $i$  vale 0 y es cierto que no hay primos antes de 0.  $\mathcal{I}$  vale luego de cada iteración, porque la condición dice que  $xs[i]$  no es primo, lo que nos permite incrementar  $i$ . Al terminar el ciclo, la negación de la condición nos dice que  $xs[i]$  es primo, y por  $\mathcal{I}$  sabemos que no hay primos antes de la posición  $i$ ; esto equivale a afirmar que la función devuelve lo esperado de acuerdo a la especificación, tomando  $j=i$ .  $\square$

## while vs. for · Otro ejemplo

¿Cómo implementamos el mismo algoritmo con un iterador `for`?

```
1  def primer_primo_v2(xs:List[int]) -> int:
2      for x in xs:
3          if es_primo(x):
4              return x
```

¡El código es mucho más claro! Pero sumamos otro problema: poner `return` dentro de un ciclo o de un condicional es (en líneas generales) una mala práctica, que puede llevar a código difícil de entender.


Lo mismo vale para la interrupción de ciclos con `break`:

```
1  def primer_primo_v3(xs:List[int]) -> int:
2      for x in xs:
3          if es_primo(x):
4              break
5      return x
```


Esto nos deja ante **otro fino equilibrio**: el `for` puede llevarnos a escribir código simple y elegante, pero también a todo lo contrario...

```
def es_primo(n:int) -> bool:
    if n == 2:
        return True
    elif n == 3:
        return True
    elif n == 5:
        return True
    elif n % 2 == 0:
        return False
    elif n % 3 == 0:
        return False
    elif n % 5 == 0:
        return False
    elif n > 5:
        i:int = 7
        while i < n:
            if i * i > n:
                break
            else:
                if n % i == 0:
                    return False
                i = i + 2
        return True
    return False
```

```
def es_primo(n:int) -> bool:
    if n==1:
        return False
    i:int = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i = i + 1
    return True
```



```
def es_primo(n:int) -> bool:
    vr:bool = (n!=1)
    i:int = 2
    while i * i <= n and vr:
        if n % i == 0:
            vr = False
        i = i + 1
    return vr
```




## while vs. for · Algunos consejos generales

- ▶ Manejen las dos formas de escribir ciclos: `while` y `for`.
- ▶ Primero resuelvan todos los ejercicios de `while` de la Guía 4 antes de pasar a los ejercicios de `for`.
- ▶ Cada vez que deban escribir un ciclo, piensen cómo lo resolverían con `while` y con `for`, y elijan la forma más sencilla y que les brinde mayor seguridad.
- ▶ Si necesitan varios `breaks` y/o `returns`, cuidado: puede ser un mal indicio (aunque no necesariamente está mal).
- ▶ En general, eviten el código largo y complejo. Separen en funciones simples, de modo **que cada función haga una sola tarea**.
- ▶ Esta manera de trabajar nos ayuda a entender mejor el problema, y nos lleva a escribir código más claro y menos propenso a errores.

## Antes de terminar, algo más sobre **for**...

Con **for** también se puede iterar sobre **strings**:

```
1 palabra:str = 'Algoritmos'
2 i:int = 0
3 while i < len(palabra):
4     print(palabra[i])
5     i = i + 1
6 for letra in palabra:
7     print(letra)
```

Python permite iterar sobre otros tipos de datos: [archivos](#), [conjuntos](#), [diccionarios](#), etc., que veremos pronto. También es muy útil sobre **rangos**:

```
1 números:List[int] = [10, 9, 13, 70, 9, 11, 18, 21, 19]
2 pos_primos:List[int] = []
3 for i in range(len(números)):
4     if es_primo(números[i]):
5         pos_primos.append(i)
6 print("Hay primos en las posiciones:", pos_primos)
```

# Repaso de la clase de hoy

- ▶ Índices y sublistas (*slices*) en listas y en strings.
- ▶ Más operaciones: `range`, `join`, `split`.
- ▶ Ciclos con el iterador `for`.
- ▶ Ventajas y desventajas de `while` y `for`.

## **Bibliografía complementaria:**

- ▶ APPP2, capítulos 7 y 8.
- ▶ HTCSP3, capítulos 8 y 11.

Con lo visto, ya pueden resolver toda la Guía de Ejercicios 5.