5.2.6. Adición de elementos a una lista

Podemos añadir elementos a una lista, esto es, hacerla crecer. ¿Cómo? Una idea que parece natural, pero que no funciona, es asignar un valor a *a*[*len*(*a*)] (siendo *a* una variable que contiene una lista), pues de algún modo estamos señalando una posición más a la derecha del último elemento. Python nos indicará que estamos cometiendo un error:

```
>>> a = [1, 2, 3] ↲
>>> a[len(a)] = 4 ↲
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Una idea mejor consiste en utilizar el operador +:

```
>>> a = [1, 2, 3] ↲
>>> a = a + 4 ↲
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

Algo ha ido mal. ¡Claro!, el operador de concatenación trabaja con *dos listas*, no con *una lista y un entero*, así que el elemento a añadir debe formar parte de una lista... aunque ésta sólo tenga un elemento:

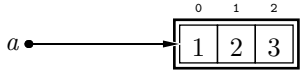
```
>>> a = a + [4] ↲
>>> a ↲
[1, 2, 3, 4]
```

Existe otro modo efectivo de añadir elementos a una lista: mediante el método *append* (que en inglés significa «añadir»). Observa cómo usamos *append*:

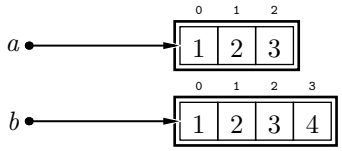
```
>>> a = [1, 2, 3] ↲
>>> a.append(4) ↲
>>> a ↲
[1, 2, 3, 4]
```

Hay una diferencia fundamental entre usar el operador de concatenación + y usar *append*: la concatenación *crea* una nueva lista copiando los elementos de las listas que participan como operandos y *append modifica* la lista original. Observa qué ocurre paso a paso en el siguiente ejemplo:

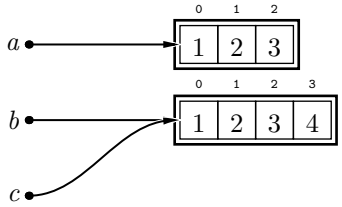
```
>>> a = [1, 2, 3] ↲
```



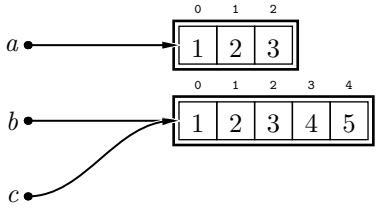
```
>>> b = a + [4] ↲
```




```
>>> c = b ↲
```




```
>>> c.append(5) ↲
```




```
>>> print a ↲
[1, 2, 3]
>>> print b ↲
[1, 2, 3, 4, 5]
>>> print c ↲
[1, 2, 3, 4, 5]
```

¿Por qué complicarse la vida con *append*, cuando la concatenación hace lo mismo y nos asegura trabajar con una copia de la memoria? Por eficiencia: es más eficiente hacer *append* que concatenar. Concatenar supone crear una lista nueva en la que se copian todos y cada uno de los elementos de las listas concatenadas. Es decir, la concatenación del siguiente ejemplo supone la copia de 1001 elementos (los 1000 de la lista original y el que añadimos):

```
>>> a = range(1000) ↲
>>> a = a + [0] ↲
```

Sin embargo, el *append* de este otro ejemplo equivalente trabaja sobre la lista original y le añade una celda cuyo contenido es 0:[3]

```
>>> a = range(1000) ↲
>>> a.append(0) ↲
```

En este ejemplo, pues, el *append* ha resultado unas 1000 veces más eficiente que la concatenación.

Desarrollemos un ejemplo práctico. Vamos a escribir un programa que construya una lista con todos los números primos entre 1 y *n*. Como no sabemos a priori cuántos hay,

[3] No siempre es *más* eficiente añadir que concatenar. Python puede necesitar memoria para almacenar la lista resultante de añadir un elemento y, entonces, ha de efectuar una copia del contenido de la lista. Pero esto supone entrar en demasiado detalle para el nivel de este texto.

construiremos una lista vacía e iremos añadiendo números primos conforme los vayamos encontrando.

En el tema anterior ya estudiamos un método para determinar si un número es primo o no, así que no nos detendremos en volver a explicarlo.

obten_primos.py

```
n = raw_input('Introduce el valor máximo: ')

primos = []
for i in range(1, n+1):
    # Determinamos si i es primo.
    creo_que_es_primo = True
    for divisor in range(2, n):
        if num % divisor == 0:
            creo_que_es_primo = False
            break
    # Y si es primo, lo añadimos a la lista.
    if creo_que_es_primo:
        primos.append(i)

print primos
```

EJERCICIOS

▶ 229 Diseña un programa que construya una lista con los n primeros números primos (ojo: no los primos entre 1 y n, sino los n primeros números primos). ¿Necesitas usar *append*? ¿Puedes reservar en primer lugar un vector con n celdas nulas y asignarle a cada una de ellas uno de los números primos?