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

```
# 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?

5.2.7. Lectura de listas por teclado

Hasta el momento hemos aprendido a construir listas de diferentes modos, pero nada hemos dicho acerca de cómo leer listas desde el teclado. La función que lee de teclado es *raw_input*, ¿funcionará también con listas?

```
>>> lista = raw_input('Dame una lista: ') ↲
Dame una lista: [1, 2, 3]
>>> lista ↲
'[1, 2, 3]'
```

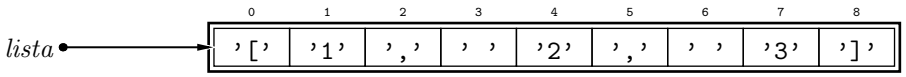
¿Ha funcionado? No. Lo que se ha leído es una cadena, no una lista. Se puede advertir en las comillas que rodean el texto de la respuesta. Podemos cerciorarnos accediendo a su primer elemento: si fuera una lista, valdría 1 y si fuera una cadena, `'['`.

```
>>> lista[0] ↲
'['
```

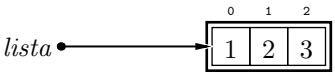
De todos modos, era previsible, pues ya dijimos en su momento que *raw_input* devolvía una cadena. Cuando queríamos obtener, por ejemplo, un entero, «encerrábamos» la llamada a *raw_input* con una llamada a la función *int* y cuando queríamos un flotante, con *float*. ¿Habrá alguna función similar para obtener listas? Si queremos una lista, lo lógico sería utilizar una llamada a *list*, que en inglés significa *lista*:

```
>>> lista = list(raw_input('Dame una lista: ')) ↲
Dame una lista: [1, 2, 3]
>>> lista ↲
['[', '1', ',', ' ', '2', ',', ' ', '3', ']']
```

¡Oh, oh! Tenemos una lista, sí, pero no la que esperábamos:



La función *list* devuelve una lista a partir de una cadena, pero cada elemento de la lista es un carácter de la cadena (por ejemplo, el 2 que ocupa la posición de índice 5 no es el entero 2, sino el carácter 2). No se interpreta, pues, como hubiéramos deseado, es decir, como esta lista de números enteros:



Para leer listas deberemos utilizar un método distinto. Lo que haremos es ir leyendo la lista elemento a elemento y construir la lista paso a paso. Este programa, por ejemplo, lee una lista de 5 enteros:

```
lista = []
for i in range(5):
    elemento = int(raw_input('Dame un elemento:'))
    lista = lista + [elemento]
```

Mejor aún: si usamos *append*, evitaremos que cada concatenación genere una lista nueva copiando los valores de la antigua y añadiendo el elemento recién leído.

```
lista = []
for i in range(5):
    elemento = int(raw_input('Dame un elemento:'))
    lista.append(elemento)
```

Existe un método alternativo que consiste en crear una lista con 5 celdas y leer después el valor de cada una:

```
lista = [0] * 5
for i in range(5):
    elemento[i] = int(raw_input('Dame un elemento:'))
```

Supongamos que deseamos leer una lista de enteros positivos cuya longitud es desconocida. ¿Cómo hacerlo? Podemos ir leyendo números y añadiéndolos a la lista hasta que nos introduzcan un número negativo. El número negativo indicará que hemos finalizado, pero no se añadirá a la lista:

```
lista = []
numero = int(raw_input('Dame un número: '))
while numero >= 0:
    lista.append(numero)
    numero = int(raw_input('Dame un número: '))
```

EJERCICIOS

▶ 230 Diseña un programa que lea una lista de 10 enteros, pero asegurándose de que todos los números introducidos por el usuario son positivos. Cuando un número sea negativo, lo indicaremos con un mensaje y permitiremos al usuario repetir el intento cuantas veces sea preciso.

▶ 231 Diseña un programa que lea una cadena y muestre por pantalla una lista con todas sus palabras en minúsculas. La lista devuelta no debe contener palabras repetidas.

Por ejemplo: ante la cadena

`'Una frase formada con palabras.  Otra frase con otras palabras.'`,

el programa mostrará la lista

`['una', 'frase', 'formada', 'con', 'palabras', 'otra', 'otras']`.

Observa que en la lista no aparece dos veces la palabra «frase», aunque sí aparecía dos veces en la cadena leída.

Lectura de expresiones Python

Hemos aprendido a leer enteros, flotantes y cadenas con *raw_input*, pero esa función no resulta útil para leer listas. Python pone a nuestro alcance otra función de lectura (*input*) de datos por teclado capaz de leer *expresiones* Python, y un literal de lista es una expresión.

Estudia estos ejemplos:

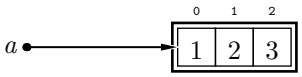
```
>>> a = input('Dame un número: ')
Dame un número: 2+2
>>> a
4
>>> b = input('Dame una cadena: ')
Dame una cadena: 'a'
>>> b
'a'
>>> c = input('Dame una lista: ')
Dame una lista: [1, 1+1, 3]
>>> c
[1, 2, 3]
```

A primera vista *input* parece mucho más flexible y útil que *raw_input*, pero presenta un gran inconveniente: el usuario de tus programas ha de saber programar en Python, ya que las expresiones deben seguir las reglas sintácticas propias del lenguaje de programación, y eso no es razonable. De todos modos, *input* puede resultarte de utilidad mientras desarrolles borradores de los programas que diseñes y manejen listas.

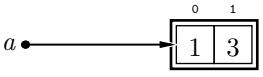
5.2.8. Borrado de elementos de una lista

También podemos eliminar elementos de una lista. Para ello utilizamos la sentencia **del** (abreviatura de «delete», que en inglés significa *borrar*). Debes indicar qué elemento deseas eliminar inmediatamente después de la palabra **del**:

```
>>> a = [1, 2, 3]
```




```
>>> del a[1]
```




```
>>> a
[1, 3]
```

La sentencia **del** *no* produce una copia de la lista sin la celda borrada, sino que modifica directamente la lista sobre la que opera. Fíjate en qué efecto produce si dos variables apuntan a la misma lista:

```
>>> a = [1, 2, 3]
>>> b = a
>>> del a[1]
>>> a
[1, 3]
>>> b
[1, 3]
```