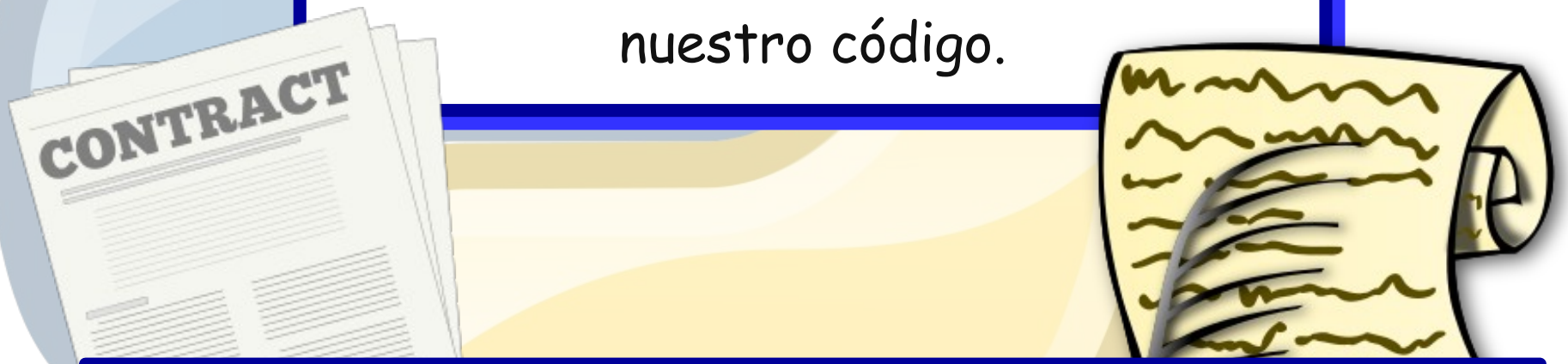
Contratos y Mutabilidad

Por Aurelio Gallardo, BY-NC-SA

¿Qué es *programar por contratos*?

Necesidad de estipular tanto lo que necesita como lo que devuelve nuestro código.



Se debe documentar en nuestro código cómo deben ser los parámetros recibidos, cómo va a ser lo que se devuelve, y qué sucede con los parámetros en caso de ser modificados. Esto es fundamental para que lo usen o comprendan otros programadores.

Precondiciones y Postcondiciones

Precondiciones: 1) Dividendo y divisor son números. 2) divisor <> 0

```
def dividir (dividendo, divisor):
    cociente = dividendo / divisor
    return cociente
print dividir (20.12, 10)
```

Las condiciones que deben cumplir los parámetros de entrada.

las condiciones que cumplirá el valor de retorno, y los parámetros recibidos, en caso de que hayan sido alterados, siempre que se hayan cumplido las precondiciones.

Postcondiciones: cociente es un número.

Aseveraciones

Aseveración: el comando **assert** puede comprobar, ejecutando el código, si se cumplen o no las pre o post-condiciones

```
def dividir (dividendo, divisor):
    assert divisor !=0, "El divisor no puede ser cero"
    cociente = dividendo / divisor
    return cociente
print dividir (25, 0)
```

Lanza un error (assertion error) antes de ejecutar el cálculo

Ejemplo final

```
def dividir (dividendo, divisor):
    """ Calculo de la división
     Pre: Recibe dos números, divisor debe ser distinto de 0.
     Post: Devuelve un número real, con el cociente de ambos.
     """
    assert divisor !=0, "El divisor no puede ser cero"
    return dividendo / ( divisor * 1.0 )
```

1.- Ajustamos Pre y post condiciones

2.- Quizás Necesites aseveraciones

3.- Los resultados Deben cumplir Las post-condiciones

Invariante de ciclo

Invariante de ciclo: una invariante de ciclo es aquella condición que ***se cumple siempre*** en una iteración.

```
def maximo(lista):
    """Devuelve el elemento máximo de la lista o None si estar vacía.
     Pre: lista con elementos comparables.
     Post: devuelve elemento máximo o None si la lista es vacía.
     Invariable: max_elem siempre es el máximo en la iteración, de cualquier
porción de la lista analizada  """
    if not len(lista):
      return None
    max_elem = lista[0]
    for elemento in lista:
        if elemento > max_elem:
            max_elem = elemento
    return max_elem

lista=[0, 23, 67, 15]
print ("El máximo de la lista es %d") % (maximo(lista))
```

Invariante de ciclo

Los invariantes de ciclo son importantes analizarlos y delimitarlos, e imponer las condiciones iniciales de ellos antes de la iteración

```
def potencia(b, n):
    "Devuelve la potencia n del número b, si n mayor que 0."
    p = 1 #IMPORTANTE: empiezo en el valor p^0
    for i in range(n):
        p = p * b
    return p
```

Variable inmutable

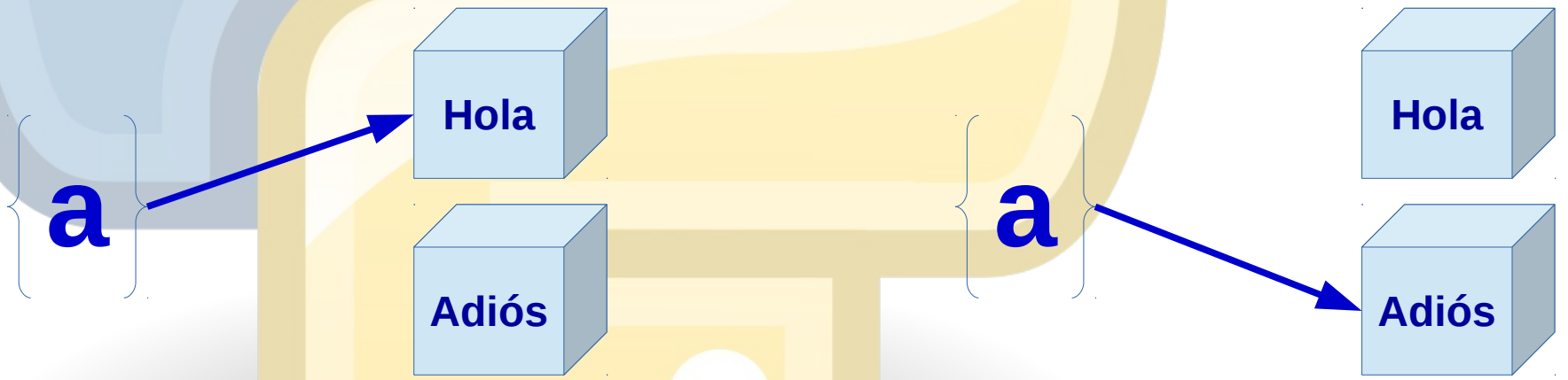
A una variable inmutable puedo cambiarle el valor, pero no su contenido. Ejemplo: a la variable **a** asigno la cadena "**Hola**". Puedo cambiarla por "**Adiós**", pero no puedo cambiar la "**H**" por "**J**"

Ejemplo de Consola

```
>>> a="Hola"
>>> a="Adiós"
>>> print a
Adiós
>>> a[0]="J"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Variable inmutable

En realidad, al cambiar el dato en **a**, lo que hacía es apuntar el nombre de la variable **a**, hacia un nuevo dato.



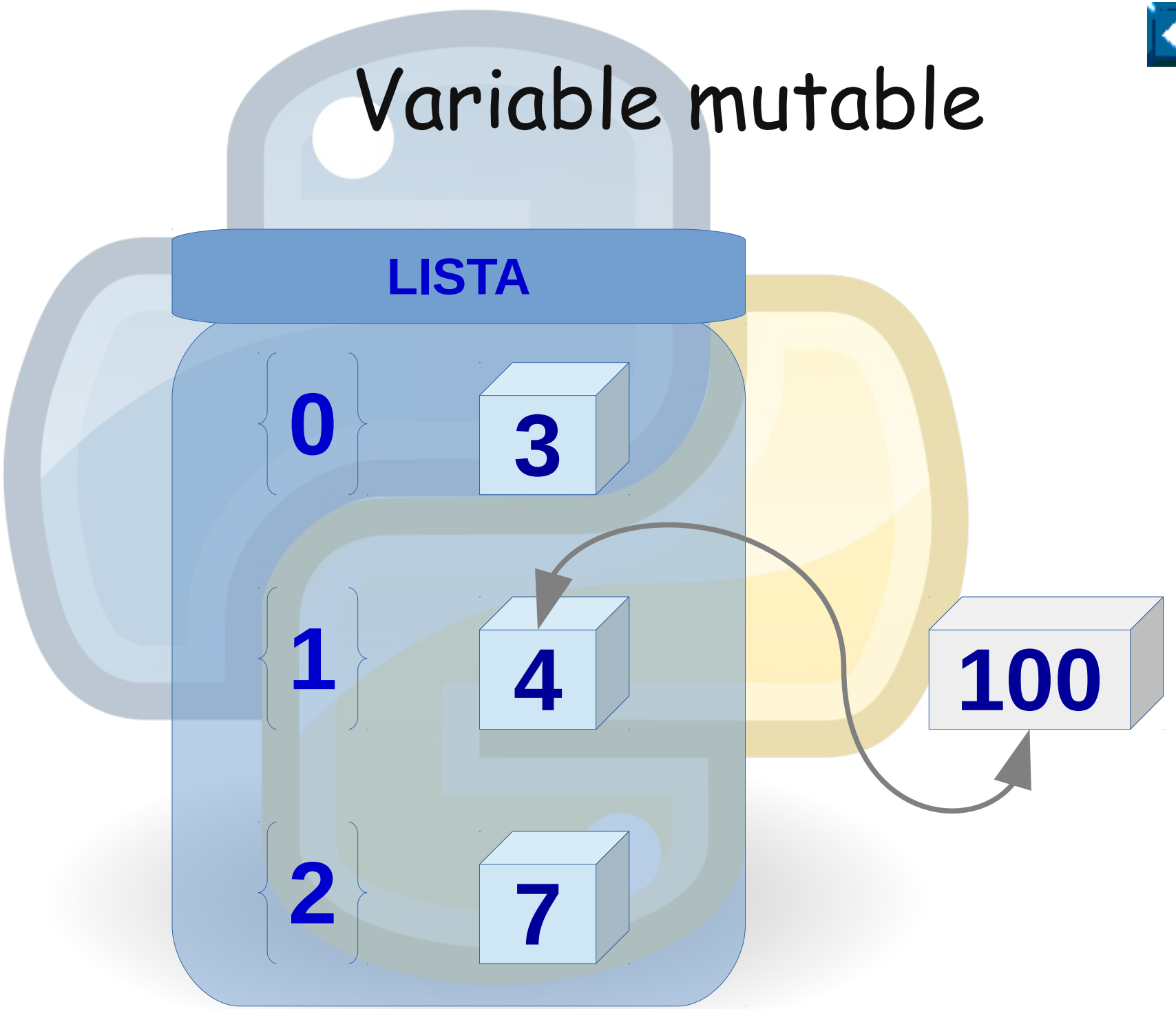
Variable mutable

A una variable mutable sí puedo cambiarle el valor y el contenido.

```
>>> lista = [3, 4, 7]
>>> print lista
[3, 4, 7]
>>> lista[1]=100
>>> print lista
[3, 100, 7]
>>>
```

Ejemplo de Consola

Variable mutable



¿Dónde apuntan las variables?

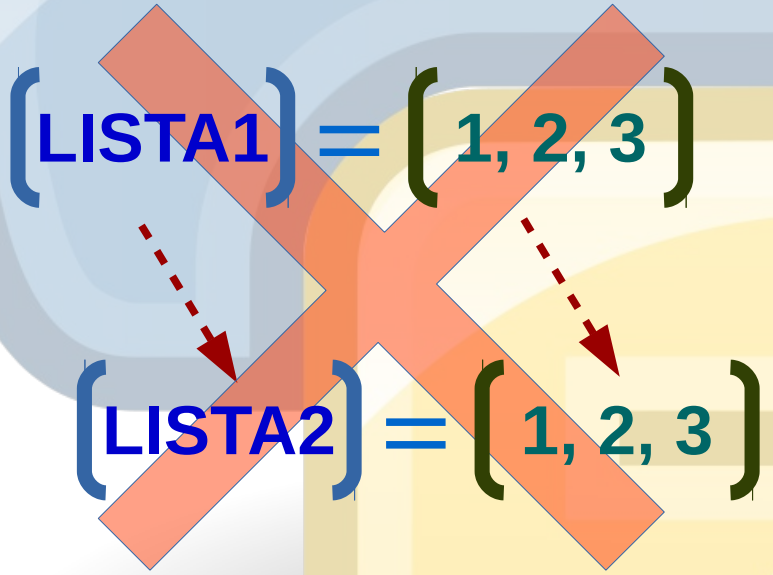
```
>>> lista1=[1,2,3]
>>> lista2=lista1
>>> print lista1, lista2
[1, 2, 3] [1, 2, 3]
>>> lista1[0]=1000
>>> print lista1, lista2
[1000, 2, 3] [1000, 2, 3]
```

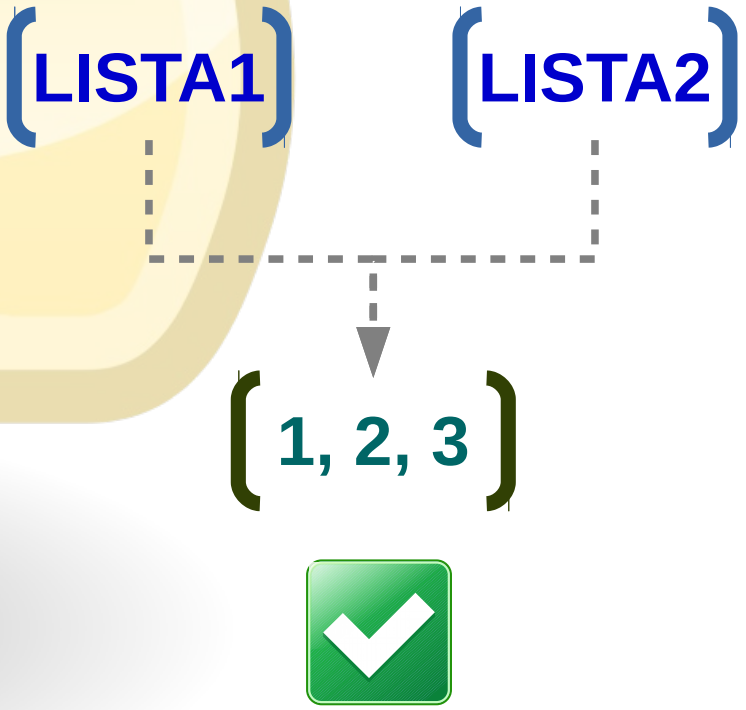
Ejemplo de Consola

¿¿?? ¿Qué ha ocurrido aquí? ¡¡No he modificado lista2, pero sin embargo al cambiar lista1[0] también se ha modificado lista2[0]

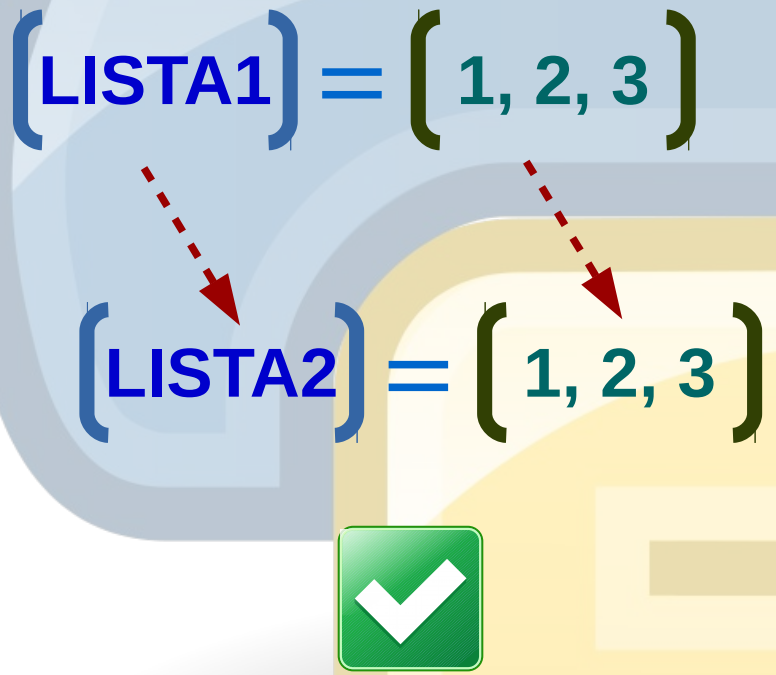
¿Dónde apuntan las variables?

El truco consiste en que lista2 no es una copia de los datos. **No es otra lista**. ***Es otro nombre para lista1 que apunta al mismo dato***





Código para "copiar" lista



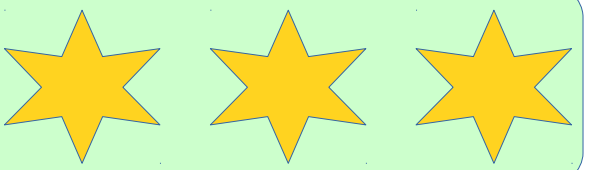

```
>>> lista1 = [1,2,3]
>>> lista2=[k for k in lista1]
>>> print lista1, lista2
[1, 2, 3] [1, 2, 3]
>>> lista1[0]=1000
>>> print lista1, lista2
[1000, 2, 3] [1, 2, 3]
```

Ejemplo de Consola

Parámetros mutables e inmutables de funciones

Las funciones reciben parámetros que pueden ser **mutables** o **inmutables**.

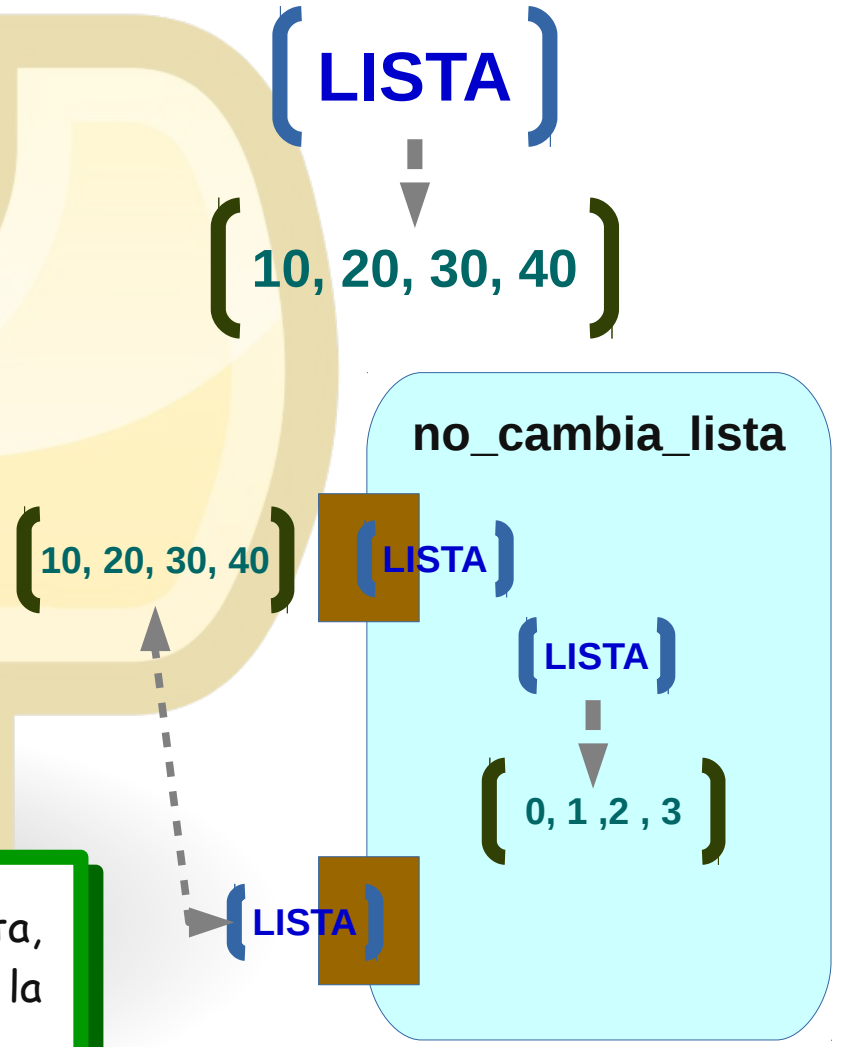
Si dentro del cuerpo de la función se modifica uno de estos parámetros para que apunte a otro valor, este cambio no se verá reflejado fuera de la función. Si, en cambio, ***se modifica el contenido de alguno de los parámetros mutables***, este cambio sí se verá reflejado fuera de la función.

Dificultad: 

NO HAY CAMBIOS

```
def no_cambia_lista(lista):
    lista = range(len(lista))
    print lista

lista = [10, 20, 30, 40]
no_cambia_lista(lista)
print lista
```

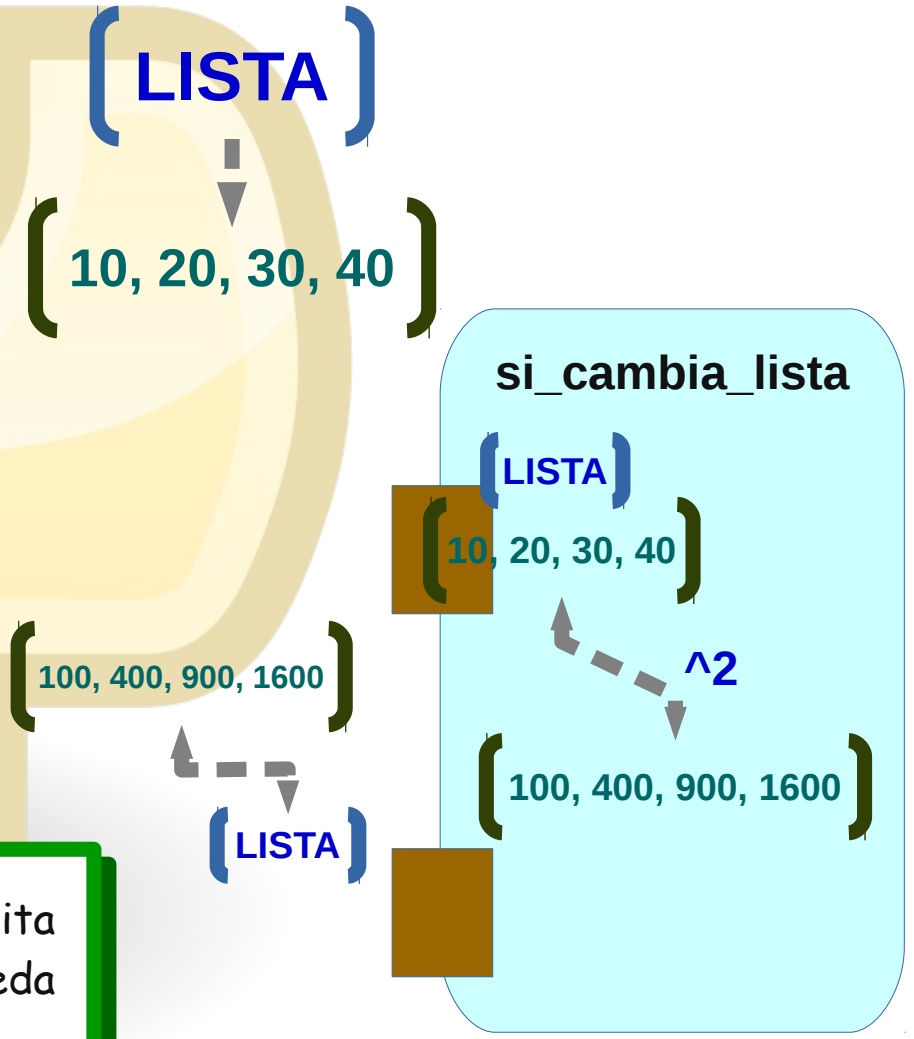


Es como si "lista" abandonase sus datos en la puerta, y ese nombre apuntase a algo distinto dentro de la función, y volviese a recuperar lo que tenía al salir.

SÍ HAY CAMBIOS

```
def si_cambia_lista(lista):
    for i in range(len(lista)):
        lista[i] = lista[i]**2
    print lista

lista = [10, 20, 30, 40]
print ("Antes de llamar la funcion: %s") %
(lista)
si_cambia_lista(lista)
print ("Despues de llamar la funcion: %s") %
(lista)
```



Es como si "lista", al llamar a lo que tiene, necesita "hacer pasar" lo que tenía asignado, y ya se queda modificado.

Analiza qué ocurre en estos casos

Caso 1

```
def cambia_lista(lista):
    lista = range(len(lista))
    for i in range(len(lista)):
        lista[i] = lista[i]**2
    print lista

lista = [10, 20, 30, 40]
print ("Antes de llamar la funcion: %s") %
(lista)
cambia_lista(lista)
print ("Despues de llamar la funcion: %s") %
(lista)
```

Caso 2

```
def cambia_lista(lista):
    for i in range(len(lista)):
        lista[i] = lista[i]**2
    lista = range(len(lista))
    print lista

lista = [10, 20, 30, 40]
print ("Antes de llamar la funcion: %s") %
(lista)
cambia_lista(lista)
print ("Despues de llamar la funcion: %s") %
(lista)
```

Conclusiones

Debemos establecer las condiciones de los parámetros de entrada de una función: **precondiciones**.

Se deben conocer las condiciones de los resultados devueltos por una función, o **postcondiciones.**

Se debe identificar las **invariantes de ciclo**, y las condiciones de estas invariantes deben cumplirse antes de ejecutarse cada iteración.

Si una **aseveración** no se cumple, generará un error.

Importante: una función no debe cambiar nunca un parámetro recibido en la entrada, a menos que esté diseñada explícitamente para ello.

Créditos

Resumen y ampliación del capítulo 10. Contratos y mutabilidad, del curso "Algoritmos de programación con Python".

http://librosweb.es/libro/algoritmos_python/capitulo_10.html