

Programación imperativa (II)

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/11/07 a las 14:17:00

Índice

1. Mutabilidad	1
1.1. Estado de un dato	1
1.2. Tipos mutables e inmutables	2
1.2.1. Valores inmutables	2
1.2.2. Valores mutables: listas	7
1.3. Alias de variables y valores mutables	11
1.4. Identidad	15
1.4.1. <code>is</code>	17
2. Cambios de estado ocultos	18
2.1. Funciones puras	18
2.2. Funciones impuras	18
2.3. Efectos laterales	18
3. Entrada y salida	20
3.1. Conceptos básicos	20
3.1.1. Entrada y salida por consola	22
3.1.2. <code>print</code>	22
3.1.3. <code>input</code>	24
3.2. Ejecución de scripts por lotes	25
3.2.1. Argumentos de la línea de órdenes	26

1. Mutabilidad

1.1. Estado de un dato

Ya hemos visto que en programación imperativa es posible cambiar el estado de una variable asignándole un nuevo valor (un nuevo dato).

Al hacerlo, no estamos cambiando el valor en sí, sino que estamos sustituyendo el valor de la variable por otro nuevo, mediante el uso de la asignación destructiva.

Sin embargo, también existen valores que poseen su propio **estado interno** y es posible cambiar dicho estado, no asignando un nuevo valor a la variable que lo contiene, sino **modificando el interior de dicho valor**.

Es decir: no estaríamos cambiando el contenido de la variable (haciendo que ahora contenga una referencia a un nuevo valor) sino **el estado interno del valor** al que hace referencia la variable.

Los valores que permiten cambiar su estado interno se denominan **mutables**.

1.2. Tipos mutables e inmutables

En Python existen tipos cuyos valores son *inmutables* y otros que son *mutables*.

Un valor **inmutable** es aquel cuyo **estado interno NO puede cambiar** durante la ejecución del programa.

Los tipos inmutables en Python son los números (`int` y `float`), los booleanos (`bool`), las cadenas (`str`), las tuplas (`tuple`), los rangos (`range`) y los conjuntos congelados (`frozenset`).

Un valor **mutable** es aquel cuyo **estado interno puede cambiar** durante la ejecución del programa **sin cambiar su identidad**.

Muchos valores mutables son **colecciones de elementos** (datos compuestos o **contenedores**) y cambiar su estado interno es cambiar su **contenido**, es decir, los elementos que contiene.

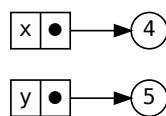
Los principales tipos mutables predefinidos en Python son las listas (`list`), los conjuntos (`set`) y los diccionarios (`dict`).

1.2.1. Valores inmutables

Un valor de un tipo inmutable no puede cambiar su estado interno durante la ejecución del programa.

Si tenemos:

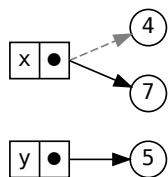
```
x = 4  
y = 5
```



y hacemos:

```
x = 7
```

quedaría:

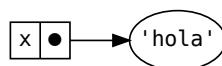


Lo que hace la asignación `x = 7` no es transformar el valor `4` convirtiéndolo en el `7` como si fuera una metamorfosis, sino que la variable `x` apunte a otro valor distinto. El valor `4` en sí mismo no cambia en ningún momento.

Las **cadenas** también son *datos inmutables* y, por tanto, con ellas ocurre exactamente igual.

Si tenemos:

```
x = 'hola'
```

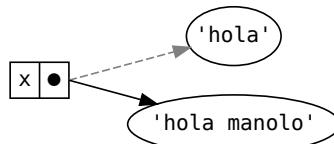


y luego hacemos:

```
x = 'hola manolo'
```

se crea una nueva cadena y se la asignamos a la variable `x`.

Es decir: la cadena '`hola`' original no se modifica (no se le añade '`manolo`' detrás), sino que la nueva **sustituye** a la anterior en la variable:



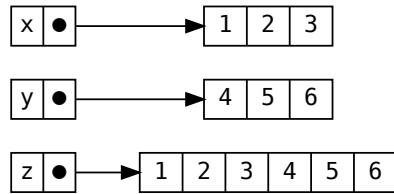
Las **tuplas** también son *datos inmutables*, por lo que, una vez creadas, no se puede cambiar su contenido.

Por ejemplo, si tenemos una tupla `(1, 2, 3)` y le concatenamos otra tupla `(4, 5, 6)`, el resultado es una nueva tupla creada a partir de las otras dos (que permanecen inalteradas):

```

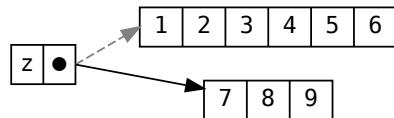
>>> x = (1, 2, 3)
>>> y = (4, 5, 6)
>>> z = x + y
>>> z
(1, 2, 3, 4, 5, 6)
>>> x
(1, 2, 3)
>>> y
(4, 5, 6)

```



Si ahora le asignamos otra tupla a `z`, ésta pasa a apuntar a la nueva tupla, sin modificar a la anterior:

`z = (7, 8, 9)`



1.2.1.1. Secuencias

Las **cadenas** y las **tuplas**, entre otros, son *datos inmutables*, así que no podemos modificarlos.

Pero también son **datos compuestos** de otros datos (sus *elementos* o *componentes*) a los que podemos acceder y con los que podemos operar individualmente, aunque no podamos cambiarlos ni sustituirlos por otros, ya que forman parte de datos compuestos inmutables.

De hecho, las cadenas y las tuplas pertenecen a la familia de las **secuencias**, que son colecciones de elementos ordenados según la posición que ocupan dentro de la secuencia.

Por tanto, con las cadenas y las tuplas podemos usar las **operaciones comunes a cualquier secuencia** de elementos.

La siguiente tabla recoge las operaciones comunes sobre secuencias, ordenadas por prioridad ascendente. `s` y `t` son secuencias del mismo tipo, `n`, `i`, `j` y `k` son enteros y `x` es un dato cualquiera que cumple con las restricciones que impone `s`.

Operación	Resultado
<code>x in s</code>	<code>True</code> si algún elemento de <code>s</code> es igual a <code>x</code>
<code>x not in s</code>	<code>False</code> si algún elemento de <code>s</code> es igual a <code>x</code>
<code>s + t</code>	La concatenación de <code>s</code> y <code>t</code>
<code>s * n</code>	Concatenar <code>s</code> consigo misma <code>n</code> veces
<code>n * s</code>	
<code>s[i]</code>	(Indexación) El <code>i</code> -ésimo elemento de <code>s</code>
<code>s[i:j]</code>	Rodaja de <code>s</code> desde <code>i</code> hasta <code>j</code>
<code>s[i:j:k]</code>	Rodaja de <code>s</code> desde <code>i</code> hasta <code>j</code> con paso <code>k</code>
<code>len(s)</code>	Longitud de <code>s</code>

Operación	Resultado
<code>min(s)</code>	El elemento más pequeño de <code>s</code>
<code>max(s)</code>	El elemento más grande de <code>s</code>
<code>s.index(x[, i[, j]])</code>	El índice de la primera aparición de <code>x</code> en <code>s</code> (desde el índice <code>i</code> inclusive y antes del <code>j</code>)
<code>s.count(x)</code>	Número total de apariciones de <code>x</code> en <code>s</code>

El **operador de indexación** permite acceder al elemento situado en la posición (también llamada **índice**) indicada entre corchetes:

```
+---+---+---+---+---+---+
s | P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
-6  -5  -4  -3  -2  -1
```

```
>>> s[2]
't'
>>> s[-2]
'o'
```

Los índices positivos (del `0` en adelante) empiezan a contar desde el comienzo de la secuencia (o sea, desde el primer elemento, el que está situado más a la izquierda).

Los índices negativos (del `-1` hacia atrás) empieza a contar desde el final de la secuencia (o sea, desde el último elemento, el que está situado más a la derecha).

El **slicing** (*hacer rodajas*) es una operación que consiste en obtener una **subsecuencia** a partir de una secuencia, indicando los **índices** de los elementos *inicial* y *final* de la misma, así como un posible *paso*:

```
con paso positivo
+---+---+---+---+---+---+
s | P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1

con paso negativo
+---+---+---+---+---+---+
s | P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
-7  -6  -5  -4  -3  -2  -1
```

```
>>> s[0:2]
'Py'
>>> s[-5:-4]
'y'
>>> s[-4:-5]
''
```

```
>>> s[0:4:2]
'Pt'
>>> s[-3:-6]
..
>>> s[-3:-6:-1]
'hty'
```

Es más fácil trabajar con las rodajas si suponemos que los índices se encuentran *entre* los elementos.

El elemento *final* nunca se alcanza.

Si el *paso* es negativo, la rodaja se hará al revés (de derecha a izquierda).

En la rodaja `s[i:j:k]`, los tres valores *i*, *j* y *k* son opcionales, así que se pueden omitir.

Si se omite *k*, se entiende que es `1`.

Si se omite *i*, se entiende que queremos la rodaja desde el elemento más la izquierda de la secuencia (si *k* es positivo) o más a la derecha (si *k* es negativo).

Si se omite *j*, se entiende que queremos la rodaja hasta el elemento más a la derecha de la secuencia (si *k* es positivo) o más a la izquierda (si *k* es negativo).

Si el índice *i* está más a la izquierda que el índice *j*, *k* debería ser positivo (de lo contrario, devolvería la secuencia vacía).

Si el índice *i* está más a la derecha que el índice *j*, *k* debería ser negativo (de lo contrario, devolvería la secuencia vacía).

Si *i* = *j*, devuelve la secuencia vacía.

Casos particulares notables:

- `s[:n]` es la rodaja desde el primer elemento de `s` hasta la posición *n*.
- `s[n:]` es la rodaja desde la posición *n* hasta el final de `s`.

Siempre se cumple que `s == s[:n] + s[n:]`.

- `s[::-1]` es la rodaja invertida desde la posición *n* hasta el primer elemento de `s`, con los elementos al revés.
- `s[:]` devuelve una *copia* de `s`.
- `s[::-1]` devuelve una copia invertida de `s`.

Ejercicio

1. Dada la siguiente tupla:

```
a = (0, 11, 22, 33, 44, 55, 66, 77, 88, 99)
```

¿Qué valor devuelven las siguientes expresiones?

- a. `a[0:7:1]`
- b. `a[0:7:2]`
- c. `a[0:7]`
- d. `a[2:7:1]`

- e. `a[7:2:1]`
- f. `a[7:2:-1]`
- g. `a[:7:1]`
- h. `a[5::1]`
- i. `a[::5:-1]`
- j. `a[-5::-1]`
- k. `a[5:-1:1]`
- l. `a[-1:5:-1]`

1.2.2. Valores mutables: listas

Los valores de tipos **mutables**, en cambio, pueden cambiar su estado interno durante la ejecución del programa sin cambiar su identidad.

El tipo mutable más frecuente es la **lista** (`list`).

Una lista es como una tupla pero que puede cambiar sus elementos, aumentar o disminuir de tamaño.

Eso significa que **una lista puede cambiar su contenido y, por tanto, su estado interno**.

Los literales de tipo `list` se crean separando sus elementos con comas y encerrándolos entre corchetes [y]:

```
>>> l = [124, 333, 'a', 3.2, 9, 53]
>>> l
[124, 333, 'a', 3.2, 9, 53]
```

También se pueden crear listas a partir de otros datos compuestos (cadenas, tuplas, rangos, etcétera) usando la función `list`:

```
>>> list((1, 2, 3))
[1, 2, 3]
>>> list('hola')
['h', 'o', 'l', 'a']
>>> list(range(0, 6))
[0, 1, 2, 3, 4, 5]
```

No se puede crear una lista con `list` a partir de un dato no compuesto:

```
>>> list(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Para ello, lo mejor sería encerrar directamente el valor entre corchetes:

```
>>> [1]
[1]
```

Las listas son secuencias mutables y, como tales, se pueden modificar usando ciertas operaciones:

- Los operadores de **indexación** y **slicing** combinados con = y `del`:

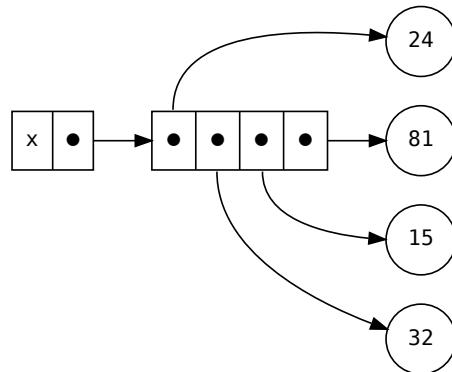
l		124		333		'a'		3.2		9		53	
0		1		2		3		4		5		6	
-6		-5		-4		-3		-2		-1			

```
>>> l = [124, 333, 'a', 3.2, 9, 53]
>>> l[3]
3.2
>>> l[3] = 99
>>> l
[124, 333, 'a', 99, 9, 53]
>>> l[0:2] = [40]
>>> l
[40, 'a', 99, 9, 53]
>>> del l[3]
>>> l
[40, 'a', 99, 53]
```

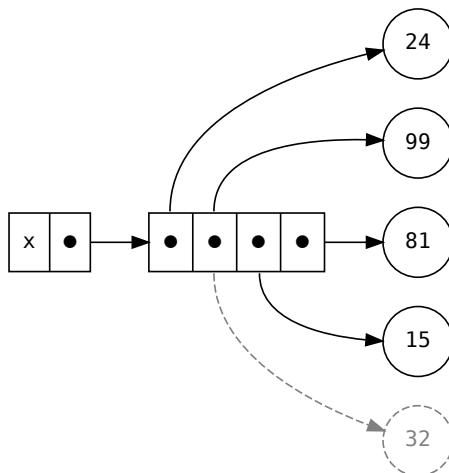
- Métodos como `append`, `clear`, `insert`, `remove`, `reverse` o `sort`.

Al cambiar el estado interno de una lista no se crea una nueva lista, sino que **se modifica la ya existente**:

```
>>> x = [24, 32, 15, 81]
>>> x[1] = 99
>>> x
[24, 99, 15, 81]
```



La lista antes de cambiar `x[1]`

La lista después de cambiar `x[1]`

Las siguientes tablas muestran todas las **operaciones** que nos permiten **modificar listas**. En ellas, `s` y `t` son listas, y `x` es un valor cualquiera.

Operación	Resultado
<code>s[i] = x</code>	El elemento i -ésimo de <code>s</code> se sustituye por <code>x</code>
<code>s[i:j] = t</code>	La rodaja de <code>s</code> desde i hasta j se sustituye por <code>t</code>
<code>s[i:j:k] = t</code>	Los elementos de <code>s[i:j:k]</code> se sustituyen por <code>t</code>
<code>del s[i:j]</code>	Elimina los elementos de <code>s[i:j]</code> Equivale a hacer <code>s[i:j] = []</code>
<code>del s[i:j:k]</code>	Elimina los elementos de <code>s[i:j:k]</code>

Operación	Resultado
<code>s.append(x)</code>	Añade <code>x</code> al final de <code>s</code> ; es igual que <code>s[len(s):len(s)] = [x]</code>
<code>s.clear()</code>	Elimina todos los elementos de <code>s</code> ; es igual que <code>del s[:]</code>
<code>s.copy()</code>	Crea una copia <i>superficial</i> de <code>s</code> ; es igual que <code>s[:]</code>
<code>s.extend(t)</code>	Extiende <code>s</code> con el contenido de <code>t</code> ; es igual que <code>s[len(s):len(s)] = t</code>
<code>s *= n</code>	Modifica <code>s</code> repitiendo su contenido n veces
<code>s.insert(i, x)</code>	Inserta <code>x</code> en <code>s</code> en el índice i ; es igual que <code>s[i:i] = [x]</code>

Operación	Resultado
<code>s.pop([i])</code>	Extrae el elemento <code>i</code> de <code>s</code> y lo devuelve (por defecto, <code>i</code> vale -1)
<code>s.remove(x)</code>	Quita el primer elemento de <code>s</code> que sea igual a <code>x</code>
<code>s.reverse()</code>	Invierte los elementos de <code>s</code>
<code>s.sort()</code>	Ordena los elementos de <code>s</code>

Partiendo de `x = [8, 10, 7, 9]`:

Ejemplo	Valor de <code>x</code> después
<code>x.append(14)</code>	[8, 10, 7, 9, 14]
<code>x.clear()</code>	[]
<code>x += [4, 5]</code>	[8, 10, 7, 9, 4, 5]
<code>x *= 2</code>	[8, 10, 7, 9, 8, 10, 7, 9]
<code>x.insert(3, 66)</code>	[8, 10, 7, 66, 9]
<code>x.pop()</code>	[8, 10, 7]
<code>x.remove(7)</code>	[8, 10, 9]
<code>x.reverse()</code>	[9, 7, 10, 8]
<code>x.sort()</code>	[7, 8, 9, 10]

Es importante tener en cuenta que, si `s` es una lista (o algún otro objeto mutable) no es lo mismo hacer:

```
s += t
```

que hacer:

```
s = s + t
```

En el primer caso, estamos *extendiendo* la lista `s` y, por tanto, la estamos *mutando*. Es decir, estamos cambiando internamente la lista sin cambiar su identidad ni crear una lista nueva.

En el segundo caso, no estamos mutando la lista `s`, sino que estamos creando una lista nueva a partir de la concatenación de las dos listas originales, y luego hacemos que `s` apunte a la nueva lista, perdiéndose la referencia a la lista `s` original. Por tanto, la identidad del objeto almacenado en `s` sí ha cambiado.

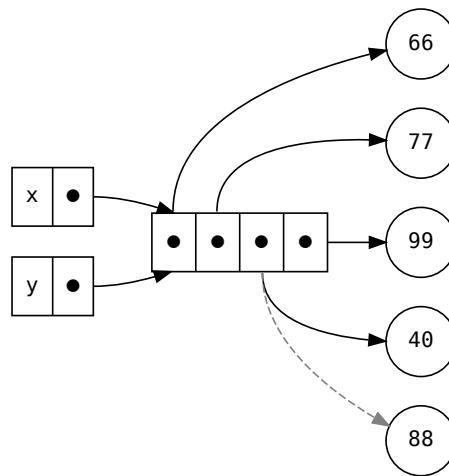
Lo mismo se puede decir de `s *= n` con respecto a `s = s * n`.

1.3. Alias de variables y valores mutables

Ya vimos que cuando una variable que tiene un valor se asigna a otra, ambas variables pasan a **compartir el mismo valor**, produciéndose el fenómeno conocido como **alias de variables**:

```
x = [66, 77, 88, 99]
y = x # x se asigna a y; ahora y tiene el mismo valor que x
```

Esto se debe a que las variables almacenan **referencias** a los valores, no los valores en sí mismos (éstos se almacenan en el montículo).



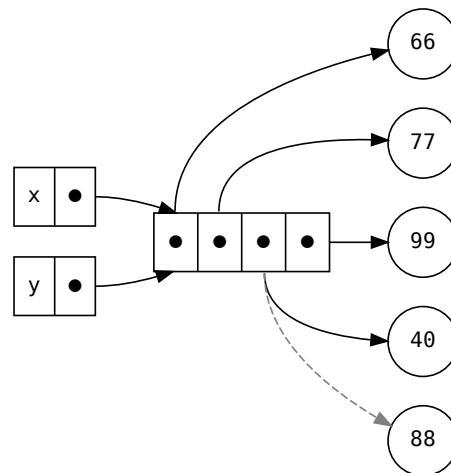
Si el valor es **mutable** y cambiamos su **contenido** desde **x**, también cambiará **y**, pues ambas variables apuntan al **mismo dato**:

```
>>> y[2] = 40
>>> x
[66, 77, 40, 99]
```

No es lo mismo cambiar el **valor** que cambiar el **contenido** del valor.

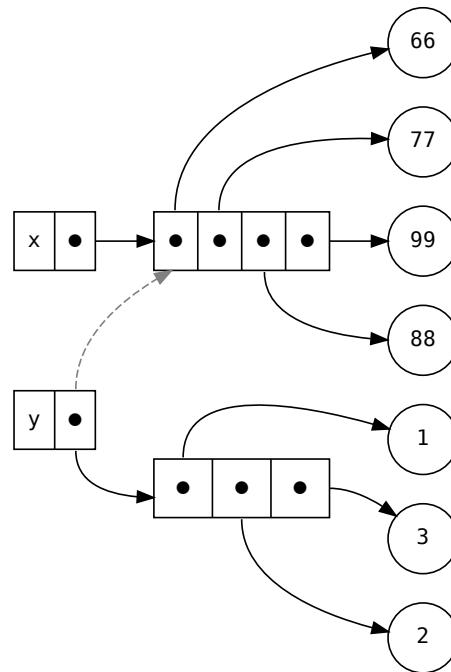
Cambiar el **contenido** es algo que sólo se puede hacer si el valor es **mutable** (por ejemplo, cambiando un elemento de una lista):

```
>>> x = [66, 77, 88, 99]
>>> y = x
>>> y[2] = 40
```



Cambiar el **valor** es algo que **siempre** se puede hacer (da igual la mutabilidad) simplemente **asignando** a la variable **un nuevo valor**:

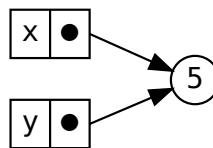
```
>>> x = [66, 77, 88, 99]
>>> y = x
>>> y = [1, 2, 3]
```



Cuando los valores son inmutables, no importa si se comparten o no, ya que no se pueden modificar. De hecho, el intérprete a veces crea valores inmutables nuevos y otras veces comparte los valores inmutables ya existentes.

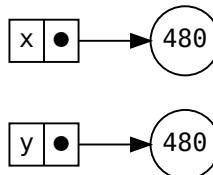
Por ejemplo, el intérprete de Python crea internamente todos los números enteros comprendidos entre -5 y 256 , por lo que todas las variables de nuestro programa que contengan el mismo número dentro de ese intervalo compartirán el mismo valor (serán *alias*):

```
x = 5 # está entre -5 y 256
y = 5
```



Se comparte el valor

```
x = 480 # no está entre -5 y 256
y = 480
```

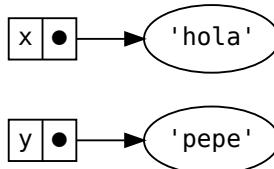


No se comparte el valor

También crea valores compartidos cuando contienen exactamente las mismas cadenas.

Si tenemos:

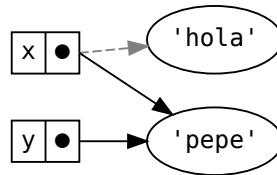
```
x = 'hola'
y = 'pepe'
```



y hacemos:

```
x = 'pepe'
```

quedaría:



El intérprete aprovecharía la cadena ya creada (buscándola primero en el montículo) y no crearía una nueva, para ahorrar memoria.

Por tanto, ambas variables contendrían la misma referencia a la misma cadena (o dicho, como también suele decirse, las dos variables serían dos referencias a la misma cadena).

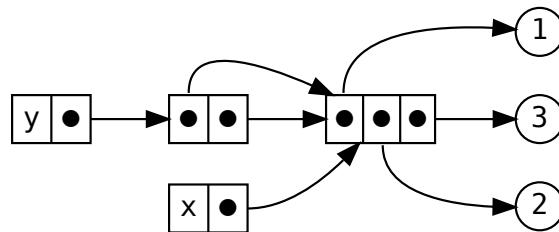
También se comparten valores si se usa el mismo dato varias veces, aunque sea un dato mutable.

De nuevo, esto se debe a que en todo caso estamos accediendo al dato mutable a través de la misma referencia.

Por ejemplo, si hacemos:

```
>>> x = [1, 2, 3]
>>> y = [x, x]
>>> y
[[1, 2, 3], [1, 2, 3]]
```

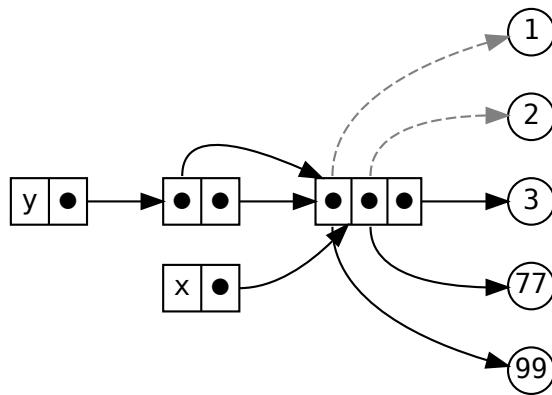
se compartiría la lista `x`, por lo que nos quedaría:



Y si ahora hacemos:

```
>>> y[0][0] = 99
>>> x[1] = 77
>>> y
[[99, 77, 3], [99, 77, 3]]
```

nos quedaría:



1.4. Identidad

Se denomina **identidad del valor** a un número entero que va asociado siempre a ese valor, el cual es único y constante durante toda la existencia del valor.

Ese número *identifica* al valor de entre todos los valores almacenados en el montículo, y lo distingue de entre todos los demás valores, ya sean iguales o distintos a él.

Dos valores distintos deben tener identidades distintas.

En cambio, dos valores iguales pueden o no tener la misma identidad.

En caso de que dos valores tengan la misma identidad, querrá decir que esos dos valores son **idénticos**, es decir, que son realmente **el mismo valor** almacenado en una única zona del montículo y al que se está accediendo a través de la **misma referencia**.

Dicho de otra forma: NO serían dos valores almacenados en dos zonas distintas del montículo y accesibles a través de dos referencias distintas.

La identidad de un valor **nunca cambia** durante la ejecución del programa o durante la misma sesión con el intérprete interactivo, incluso aunque el dato sea mutable y cambie su contenido.

En cambio, la identidad del valor sí que puede cambiar (y lo normal es que cambie) entre dos ejecuciones distintas del mismo programa, o entre dos sesiones distintas con el intérprete interactivo.

En Python, la identidad de un valor se consulta usando la función `id`:

```
>>> id('holo')
140294723570672
>>> id('adiós')
140587522259616
```

```
>>> id(5)
140666458866032
>>> id(400)
140635543277200
```

Si hacemos:

```
>>> id('prueba')
139905258241392
>>> id('prueba')
139905255890928
```

puede parecer que la identidad del valor '`prueba`' ha cambiado, ya que hemos consultado dos veces su identidad usando la función `id` y en cada caso nos ha devuelto resultados diferentes.

Sin embargo, lo que ocurre es que esos dos '`prueba`' **no son el mismo valor**, sino dos *valores iguales* que se han creado en momentos diferentes y que ocupan zonas diferentes en la memoria, por lo que tienen *identidades diferentes* (aunque los valores sean iguales).

En cambio, si hacemos:

```
>>> x = 'prueba'
>>> id(x)
139754569626160
>>> id(x)
139754569626160
```

se obtiene el mismo resultado, ya que sólo hay un único valor '`prueba`' en la memoria y, por tanto, la identidad es la misma en ambos casos.

Los conceptos de *identidad* y de *referencia* están relacionados.

Para comprobar si dos valores son **idénticos** (es decir, si son realmente *el mismo valor*) podríamos hacer una de estas dos comprobaciones:

- Comprobar si tienen la misma identidad.
- Comprobar si se usan las mismas referencias para acceder a ellos (es decir, si apuntan al mismo lugar en el montículo).

Así que, para saber si dos datos son **idénticos** (es decir, si ambos contienen *el mismo valor*, no simplemente **dos valores iguales**), podríamos intentar comparar sus referencias y ver su coinciden.

Pero el lenguaje Python no nos permite comparar directamente dos referencias, ya que, cuando hacemos:

```
x == y
```

en realidad estamos preguntando si son iguales sus *valores*, no sus *referencias*.

Por tanto, para saber si dos datos son idénticos, debemos usar otro mecanismo que no sea el de comparar referencias.

Una forma de saber si dos valores son *el mismo valor* (es decir, si son **idénticos**), es usar la función `id`.

Sabemos que la función `id` aplicada a un valor devuelve la **identidad** del valor, que es un número único para cada valor.

Por esa razón, si dos variables tienen el mismo `id`, significa que ambas apuntan al mismo valor en la memoria y que, por tanto, son **referencias al mismo valor**.

En general, si dos datos tienen el mismo `id`, decimos que son **idénticos**, porque en realidad son *el mismo dato*.

```
>>> id('hola') == id('hola')
True
>>> x = 'hola'
>>> y = 'hola'
>>> id(x) == id(y)
True
```

```
>>> x = [1, 2, 3, 4]
>>> y = [1, 2, 3, 4]
>>> id(x) == id(y)
False
>>> y = x
>>> id(x) == id(y)
True
```

1.4.1. `is`

Otra forma más directa de comprobar si dos datos son realmente el mismo dato en memoria (es decir, si son **idénticos**) es usar el operador `is`, que comprueba si los dos datos tienen la misma **identidad**:

Su sintaxis es:

```
<is> ::= <valor1> is <valor2>
```

Es un operador relacional que devuelve `True` si `<valor1>` y `<valor2>` tienen la misma **identidad** (es decir, si son **el mismo dato en memoria** y, por tanto, son **idénticos**) y `False` en caso contrario.

En la práctica, equivale a hacer `id(<valor1>) == id(<valor2>)`.

Lo normal es usar el `is` con variables y, en tal caso, devuelve `True` si los datos que almacenan las variables son realmente el mismo dato.

No es correcto usarlo con literales inmutables (y el intérprete lo avisa con un `SyntaxWarning`), ya que, en tal caso, devuelve siempre `True`.

En cambio, con literales mutables (como `[]`) devuelve siempre `False`.

También existe el operador `is not`, que es el contrario al `is` y cuya sintaxis es:

```
<is_not> ::= <valor1> is not <valor2>
```

El operador `is not` es un operador relacional que, aplicado a dos valores, devolverá `True` si los valores **no** son idénticos.

```
>>> x = 500
>>> y = 500
>>> x is y
False
>>> x is not y
True
```

```
>>> y = x
>>> x is y
True
>>> 500 is 500
<stdin>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
True
>>> x = 'hola'
>>> y = 'hola'
>>> x is y
True
>>> x is not y
False
```

2. Cambios de estado ocultos

2.1. Funciones puras

Las **funciones puras** son aquellas que cumplen que:

- su valor de retorno depende únicamente del valor de sus argumentos, y
- calculan su valor de retorno sin provocar cambios de estado observables en el exterior de la función.

Una llamada a una función pura se puede sustituir libremente por su valor de retorno sin afectar al resto del programa (es lo que se conoce como **transparencia referencial**).

Las funciones *puras* son las únicas que existen en programación funcional.

2.2. Funciones impuras

Por contraste, una función se considera **impura** si:

- su valor de retorno depende de algo más que de sus argumentos, o
- provoca cambios de estado observables en el exterior de la función.

En éste último caso decimos que la función provoca **efectos laterales**.

Toda función que provoca efectos laterales es impura, pero no todas las funciones impuras provocan efectos laterales (puede ser impura porque su comportamiento se vea afectado por los efectos laterales provocados por otras partes del programa).

2.3. Efectos laterales

Un **efecto lateral** (*side effect*) es cualquier cambio de estado provocado por una parte del programa (por ejemplo, una función) que puede observarse desde otras partes del mismo, las cuales podrían verse afectadas por ese efecto de una manera poco evidente o impredecible.

Una función puede provocar efectos laterales, o verse afectada por efectos laterales provocados por otras partes del programa.

En cualquiera de estos casos, tendríamos una función **impura**.

Los casos típicos de efectos laterales en una función son:

- Cambiar el valor de una variable global.
- Cambiar el estado de un argumento mutable.
- Realizar una operación de entrada/salida.

En un lenguaje imperativo **se pierde la transparencia referencial**, ya que el valor de una función puede depender no sólo de los valores de sus argumentos, sino también de los valores de las variables libres, los cuales pueden cambiar durante la ejecución del programa:

```
>>> suma = lambda x, y: x + y + z
>>> z = 2
>>> suma(3, 4)
9
>>> z = 20
>>> suma(3, 4)
27
```

Por tanto, cambiar el valor de una variable global (en cualquier parte del programa) es considerado un **efecto lateral**, ya que puede alterar el comportamiento de otras partes del programa de formas a menudo impredecibles o poco evidentes.

Cuando el efecto lateral lo produce la propia función también estamos perdiendo transparencia referencial, pues en tal caso no podemos sustituir libremente la llamada a la función por su valor de retorno, ya que ahora la función hace *algo más que calcular dicho valor*, y ese *algo* es un efecto observable fuera de la función.

Por ejemplo, una función que imprime por la pantalla o escribe en un archivo del disco está provocando un efecto observable fuera de la función, por lo que tampoco es una función pura y, por tanto, no cumple la transparencia referencial.

Lo mismo pasa con las funciones que modifican algún argumento mutable. Por ejemplo:

```
>>> ultimo = lambda x: x.pop()
>>> lista = [1, 2, 3, 4]
>>> ultimo(lista)
4
>>> ultimo(lista)
3
>>> lista
[1, 2]
```

Los efectos laterales hacen que sea más difícil razonar sobre el funcionamiento del programa, porque las funciones impuras no pueden verse como simples correspondencias entre los datos de entrada y el resultado de salida, sino que además hay que tener en cuenta los **efectos laterales** que producen en otras partes del programa.

Por ello, se debe **evitar**, siempre que sea posible, escribir funciones impuras.

Ahora bien: muchas veces, la función que se desea escribir tiene efectos laterales porque esos son, precisamente, los efectos deseados.

Por ejemplo, una función que actualice los salarios de los empleados en una base de datos, a partir del salario base y los complementos.

En ese caso, es importante **documentar** adecuadamente la función para que, quien desee usarla, sepa perfectamente qué efectos produce más allá de devolver un resultado.

3. Entrada y salida

3.1. Conceptos básicos

En principio, el **exterior** de una función es el resto del programa del que forma parte.

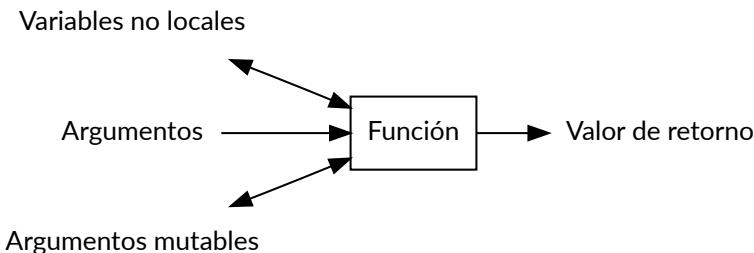
Visto así, una función (que es un *subprograma*) puede comunicarse con su exterior principalmente mediante:

- el paso de argumentos y
- la devolución de su valor de retorno,



Comunicación básica de una función con el exterior

Aunque sabemos que también puede interactuar con su exterior mediante **efectos laterales** como el uso de variables no locales y la modificación de argumentos mutables.

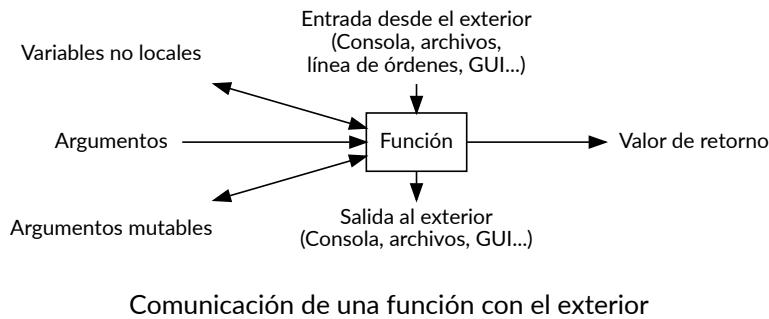


Comunicación de una función con el exterior incluyendo efectos laterales

Pero además, una función puede interactuar con el exterior del programa realizando **operaciones de entrada/salida (E/S)**.

El **exterior de un programa** puede ser, entre otros:

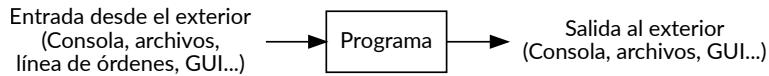
- El teclado.
- La pantalla.
- Un archivo del disco duro.
- Otro ordenador de la red.



Las operaciones de E/S se consideran **efectos laterales** porque:

- pueden producir cambios en el exterior, o
- pueden hacer que el resultado de una función dependa de los datos leídos del exterior y, por tanto, ya no sólo dependería de sus argumentos.

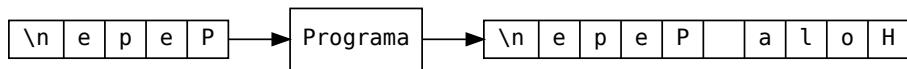
Gracias a ello, un programa puede comunicarse con el exterior mediante operaciones de lectura en la entrada y operaciones de escritura en la salida, que son efectos laterales.



Comunicación de un programa con el exterior

De esta forma, el programa puede interactuar con el usuario, solicitarle datos y proporcionarle resultados.

En Python (así como en otros lenguajes de programación), la E/S de información desde y hacia el exterior se realiza por medio de **flujos** (del inglés, *streams*), que son secuencias de caracteres o de bytes:



Ejemplo de un programa que saluda

Esos flujos actúan como «colas» de elementos (caracteres o bytes), de forma que:

- En la entrada, los elementos están a la espera de ser consumidos por el programa.
- En la salida, los elementos se van encolando a medida que el programa los va generando y volcando al exterior.

Dependiendo de las características del **flujo de entrada**, hay dos posibilidades:

- a. *Lectura secuencial*: El programa sólo puede acceder al primer elemento de la cola, de manera que va leyéndolos de uno en uno y se van sacando de la cola a medida que se van leyendo.

- b. *Lectura directa*: El programa puede moverse libremente por el flujo y acceder a cualquier elemento del mismo, sin importar la posición en la que esté el elemento dentro del flujo.

Asimismo, dependiendo de las características del **flujo de salida**, también hay dos posibilidades:

- Escritura secuencial*: El programa sólo puede escribir al final de la cola de salida.
- Escritura directa*: El programa puede escribir elementos libremente en cualquier posición dentro del flujo.

Finalmente, algunos flujos pueden actuar como flujo de entrada y de salida al mismo tiempo y, en tal caso, el programa podría leer y escribir datos sobre el mismo flujo.

3.1.1. Entrada y salida por consola

Por **consola** entendemos de forma genérica la pantalla y el teclado del ordenador.

Por tanto, la E/S por consola se refiere a las operaciones de lectura de datos por el teclado y escritura a la pantalla.

Mientras no se diga lo contrario, el sistema operativo conecta la consola a dos flujos llamados **entrada estándar** y **salida estándar**, de forma que el teclado está conectado a la entrada estándar y la pantalla a la salida estándar.

Esos flujos se pueden *redireccionar* a otros archivos o dispositivos usando las redirecciones de la shell: <, > o |:

```
$ python programa.py < entrada.txt  
$ python programa.py > salida.txt
```

En Python, esos dos flujos se pueden manipular mediante los objetos `sys.stdin` y `sys.stdout`, respectivamente.

También existe el flujo `sys.stderr` que representa la **salida estándar de errores** del sistema operativo.

El intérprete ya abre automáticamente los flujos `sys.stdin`, `sys.stdout` y `sys.stderr` nada más arrancar y los conecta a la entrada estándar, la salida estándar y la salida estándar de errores, respectivamente.

3.1.2. print

La función `print` imprime (escribe) por la salida (normalmente la pantalla) el valor de una o varias expresiones.

Produce un efecto lateral porque cambia el exterior del programa, afectando al estado de un dispositivo de salida.

Su firma es:

```
print(<expresión>[, <expresión>]* [, sep=<expresión>][, end=<expresión>])
```

El `sep` es el *separador* y su valor por defecto es ' ' (un espacio).

El `end` es el *terminador* y su valor por defecto es '`\n`'.

Las expresiones se convierten en cadenas antes de imprimirse.

Por ejemplo:

```
>>> print('hola', 'pepe', 23)
hola pepe 23
```

3.1.2.1. Paso de argumentos por palabras clave

Normalmente, los argumentos se pasan a los parámetros posicionalmente (lo que se denomina **paso de argumentos posicional**).

Según este método, los argumentos se asignan a los parámetros correspondientes según la posición que ocupan en la llamada a la función (el primer argumento se asigna al primer parámetro, el segundo al segundo parámetro y así sucesivamente).

En Python también existe el **paso de argumentos por palabra clave**, donde cada argumento se asigna a su parámetro indicando en la llamada el nombre del parámetro y el valor de su argumento correspondiente separados por un `=`, como si fuera una asignación.

Esta técnica se usa en la función `print` para indicar el separador o el terminador de la lista de expresiones a imprimir.

Si se combinan ambas técnicas en una misma llamada, los argumentos pasados por palabras clave se ponen al final.

Por ejemplo:

```
>>> print('hola', 'pepe', 23, sep='*')
hola*pepe*23
>>> print('hola', 'pepe', 23, end='-')
hola pepe 23-
```

3.1.2.2. El valor `None`

Es importante resaltar que la función `print` **no devuelve** el valor de las expresiones, sino que las **imprime** (provoca el efecto lateral de cambiar la pantalla haciendo que aparezcan nuevos caracteres).

Por tanto, no es lo mismo hacer:

```
>>> 'Hola mundo'
'Hola mundo'
```

que hacer:

```
>>> print('Hola mundo')
Hola mundo
```

La función `print` como tal no devuelve ningún valor, pero como en Python todas las funciones deben devolver *algún* valor, en realidad lo que ocurre es que `print` **devuelve un valor `None`**.

`None` es un valor especial que significa «**ningún valor**» y se utiliza principalmente para casos en los que no tiene sentido que una función devuelva un valor determinado, como es el caso de `print`.

Pertenece a un tipo de datos especial llamado `NoneType` cuyo único valor posible es `None`, y para comprobar si un valor es `None` se usa `<valor> is None`.

Sólo existe un único valor `None` en el montículo, que se crea justo al arrancar el intérprete.

Podemos comprobar que, efectivamente, `print` devuelve `None`:

```
>>> print('hola', 'pepe', 23) is None
hola pepe 23 # ésto es lo que imprime print
True        # ésto es el resultado de comprobar si el valor de print es None
```

Otra forma, usando variables:

```
>>> x = print('hola', 'pepe', 23)
hola pepe 23 # ésto es lo que imprime print
>>> x is None
True         # ésto es el resultado de comprobar si el valor de x es None
```

3.1.3. `input`

La función `input` lee datos introducidos desde la entrada (normalmente el teclado) y devuelve el valor del dato introducido, que siempre es una **cadena** a la cual se le ha eliminado el posible salto de línea final.

Su firma es:

```
input([prompt: str]) -> str
```

Por ejemplo:

```
>>> nombre = input('Introduce tu nombre: ')
Introduce tu nombre: Ramón
>>> print('Hola,', nombre)
Hola, Ramón
```

En primer lugar, provoca el *efecto lateral* de alterar el estado del dispositivo de salida imprimiendo el `prompt`.

Además, provoca el *efecto lateral* de alterar el estado del dispositivo de entrada, ya que se espera a que desde el exterior se introduzca el dato solicitado consumiendo los caracteres que haya en la entrada hasta el salto de línea.

Eso hace que sea *impura* por partida triple: provoca dos efectos laterales y puede devolver un resultado distinto cada vez que se la llama.

3.2. Ejecución de scripts por lotes

A partir de ahora, ya podemos escribir programas que se comuniquen directamente con el usuario mediante la entrada y salida por consola.

Por tanto, ya no necesitamos ejecutar las sentencias dentro de una sesión en el intérprete interactivo y podemos pasárselas a los usuarios para que lo ejecuten.

En Python, los programas se almacenan en archivos llamados *scripts*.

Los usuarios ejecutan los programas directamente desde el sistema operativo, llamando al intérprete por lotes y pasándole el nombre del *script* que desea ejecutar.

Por ejemplo, para ejecutar el *script* `programa.py`, el usuario escribiría lo siguiente en un terminal del sistema operativo:

```
$ python programa.py
```

Observar que no se usa ninguna opción en la orden; solamente el nombre del archivo.

Nosotros, los programadores, durante el desarrollo del programa podemos ejecutarlo directamente dentro del entorno integrado de desarrollo (IDE).

Por ejemplo, desde Visual Studio Code podemos ejecutar el *script* actual sin salirnos del editor usando una de las siguientes opciones:

- Pulsando en el icono del triángulo verde situado en la esquina superior derecha de la ventana del editor:



- Pulsando la tecla `F1` y escribiendo:

`>Python: Run Python File in Terminal`

en el panel de comandos que aparece.

En cualquiera de los dos casos, Visual Studio Code abre un terminal integrado y ejecuta ahí dentro el comando `python programa.py` como si lo hubiéramos escrito nosotros desde el sistema operativo.

Ejercicio

2. Explicar las diferencias entre estas tres formas de ejecutar el intérprete. Indicar en qué casos es conveniente usar cada una:
 - a. `$ python`
 - b. `$ python script.py`
 - c. `$ python -i script.py`

3.2.1. Argumentos de la línea de órdenes

El usuario del programa puede pasárselos directamente desde la línea de órdenes del sistema operativo al arrancar el programa con el intérprete por lotes.

Para acceder a esos argumentos desde el programa, puede usarse el atributo `argv` del módulo `sys`. `sys.argv` es una lista que el intérprete rellena automáticamente con los argumentos usados por el usuario del script al ejecutarlo desde la línea de órdenes.

El primer elemento de la lista (es decir, `sys.argv[0]`) siempre contiene el nombre del script.

Los restantes elementos (`sys.argv[1]`, `sys.argv[2]`, etcétera) contendrán los argumentos pasados al script, en el orden en el que aparecen en la línea de órdenes.

Por ejemplo, un programa puede saludar al usuario dirigiéndose a él por su nombre, el cual tendrá que pasarlo mediante la línea de órdenes de la siguiente forma:

```
$ python saluda.py Ricardo
```

Al ejecutar el script `saluda.py`, el nombre del script aparecerá en `sys.argv[0]` y el primer argumento enviado al mismo (en este caso, el nombre del usuario `Ricardo`) aparecerá en `sys.argv[1]`:

```
import sys
print(';Hola,', sys.argv[1])
```

Si el usuario no le pasa ningún argumento en la línea de órdenes, `sys.argv[1]` no existirá y se producirá un error `IndexError` al intentar acceder a él:

```
$ python saluda.py
Traceback (most recent call last):
  File "saluda.py", line 2, in <module>
    print(';Hola,', sys.argv[1])
IndexError: list index out of range
```

El siguiente programa suma dos números indicados por el usuario mediante la línea de órdenes.

En este caso, hay que tener en cuenta que los argumentos siempre se pasan como cadenas, así que habrá que convertirlos previamente a números:

```
import sys
x = float(sys.argv[1])
y = float(sys.argv[2])
print('La suma de', x, '+', y, 'es', x + y)
```

Un ejemplo de ejecución de este programa sería:

```
$ python suma.py 4 3
La suma de 4.0 + 3.0 es 7.0
```

Bibliografía

- Aguilar, Luis Joyanes. 2008. *Fundamentos de Programación*. Aravaca: McGraw-Hill Interamericana de España.
- Pareja Flores, Cristóbal, Manuel Ojeda Aciego, Ángel Andeyro Quesada, and Carlos Rossi Jiménez. 1997. *Desarrollo de Algoritmos y Técnicas de Programación En Pascal*. Madrid: Ra-Ma.
- Python Software Foundation. n.d. "Sitio Web de Documentación de Python." <https://docs.python.org/3>.