Programación imperativa

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 15 de noviembre de 2020 a las 22:48:00

Índice general

1. Modelo de ejecución

1.1. Máquina de estados

La **programación imperativa** es un paradigma de programación basado en los conceptos de «***estado***» y «***sentencia***».

Un programa imperativo está formado por una **secuencia de sentencias**

El programa imperativo va pasando por diferentes **estados** a medida que se van ejecutando las sentencias que lo forman.

Por tanto, una **sentencia** es una instrucción que cambia el estado del programa.

El modelo de ejecución de un programa imperativo es el de una **máquina de estados**, es decir, un dispositivo abstracto que va pasando por diferentes estados a medida que el programa va ejecutándose.

El concepto de «***tiempo***» también es muy importante en programación imperativa, ya que el estado del programa va cambiando a lo largo del tiempo conforme se van ejecutando las sentencias que lo forman.

A su vez, el comportamiento del programa depende del estado en el que se encuentre.

Eso significa que, ante los mismos datos de entrada, una función en programación imperativa puede devolver **valores distintos en momentos distintos**.

En programación funcional, en cambio, el comportamiento de una función no depende del momento en el que se ejecute, ya que siempre devolverá los mismos resultados ante los mismos datos de entrada (*transparencia referencial*).

Eso significa que, para modelar el comportamiento de un programa imperativo, ya **no nos vale el modelo de sustitución** que hemos estado usando hasta ahora en programación funcional.

1.2. Sentencias

Las **sentencias** son las instrucciones principales que componen un programa imperativo.

La ejecución de una sentencia **cambia el estado interno del programa** provocando uno de estos **efectos**:

- **Cambiar las coordenadas** del proceso asociado al programa, normalmente mediante la llamada **sentencia de asignación**.
- Cambiar el **flujo de control** del programa, haciendo que la ejecución se bifurque (*salte*) a otra parte del mismo.

La principal diferencia entre una *sentencia* y una *expresión* es que las sentencias no denotan ningún valor, sino que son órdenes a ejecutar por el programa para producir un *efecto*.

- Las *expresiones* se *evalúan* y devuelven un *valor*.
- Las *sentencias* se *ejecutan* para producir un *efecto*.

En un lenguaje funcional puro:

- Un programa es una expresión.
- Ejecutar un programa consiste en evaluar dicha expresión usando las definiciones predefinidas del lenguaje y las definidas por el programador.
- Todo son expresiones, excepto las sentencias que producen el efecto de crear *ligaduras* (como las sentencias de definición, o de importación de módulos).
- Evaluar una expresión no produce ningún otro efecto salvo el de calcular su valor.
- Las expresiones devuelven siempre el mismo valor (tienen *transparencia referencial*).
- El comportamiento de un programa se puede modelar usando el *modelo de sustitución*.

En cambio, en un lenguaje imperativo:

- Los programas están formados por sentencias que, al ejecutarse, van cambiando el estado del programa.
- El valor de una expresión depende del estado de en el que se encuentre el programa en el momento de evaluar dicha expresión (no hay *transparencia referencial*).
- Evaluar una expresión puede provocar otros efectos (los llamados *efectos laterales*) más allá de calcular su valor.
- En muchos lenguajes imperativos es posible colocar una expresión donde debería ir una sentencia (aunque no al revés).

  Esto sólo resultaría útil en caso de que la evaluación de la expresión provocara *efectos laterales*. De lo contrario, el valor de la evaluación se perdería sin más y no habría servido de nada calcularlo.

1.3. Secuencia de sentencias

Un programa imperativo está formado por una **secuencia de sentencias**.

Ejecutar un programa imperativo es provocar los **cambios de estado** que dictan las sentencias en el **orden** definido por el programa.

Las sentencias del programa van provocando **transiciones** entre estados, haciendo que la máquina pase de un estado al siguiente.

Para modelar el comportamiento de un programa imperativo tendremos que saber en qué estado se encuentra el programa, para lo cual tendremos que seguirle la pista desde su estado inicial al estado actual.

Eso básicamente se logra «ejecutando» mentalmente el programa sentencia por sentencia y llevando la cuenta de los cambios que van produciendo conforme se van ejecutando.

Al decir que un programa imperativo está formado por una *secuencia* de sentencias, estamos diciendo que importa mucho el orden en el que están colocadas las sentencias dentro del programa.

En general, un programa imperativo se comportará de forma diferente si se cambia el orden en el que se ejecutan sus sentencias.

Por eso, si se ejecuta *A* antes que *B*, el programa seguramente no producirá el mismo efecto que si se ejecuta *B* antes que *A*.

Por ejemplo, muchas veces el funcionamiento de una sentencia *B* depende del efecto producido por una sentencia *A* anterior. Por tanto, en ese caso, *A* debería ejecutarse antes que *B*.

2. Asignación destructiva

2.1. Identidad

Todos los valores se almacenan en una zona de la memoria conocida como el **montículo**.

Cada vez que aparece un nuevo dato en el programa, el intérprete lo crea dentro del montículo a partir de una determinada dirección de la memoria y ocupando el espacio de memoria que se necesite en función del tamaño que tenga el dato.

Se denomina **identidad del dato** a un valor abstracto y único que va asociado siempre al dato, que no cambia nunca durante toda la vida del dato, y que sirve para identificar, localizar y acceder al dato dentro del montículo.

Generalmente, la identidad de un dato coincide con la **dirección de comienzo** de la zona que ocupa el dato dentro del montículo, aunque ese es un detalle de funcionamiento interno del intérprete.

2.2. Variables y referencias

Una **variable** es un lugar en la **memoria** donde se puede **almacenar la identidad** de un dato almacenado en el montículo.

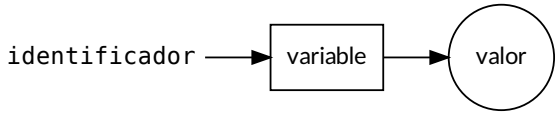
En tal caso, decimos que la variable **es una referencia al dato**, o que **contiene una referencia al dato**, o que **hace referencia al dato** o que **apunta al dato**.

Por abuso del lenguaje, también se suele decir que la variable **almacena o contiene el dato**, aunque eso no es estrictamente cierto.

El valor de una variable (o mejor dicho, la referencia que contiene) **puede cambiar** durante la ejecución del programa, haciendo que la variable pueda *apuntar* a distintos valores durante la ejecución del programa.

A partir de ahora, un identificador no se liga directamente con un valor, sino que tendremos:

- Una **ligadura** entre un identificador y una **variable**.
- La variable **hace referencia** al valor.



Este comportamiento es el propio de los **lenguajes de programación *orientados a objetos*** (como Python o Java), que son los lenguajes imperativos más usados a día de hoy.

Otros lenguajes imperativos más «clásicos» se comportan, en general, de una forma diferente.

En esos lenguajes (como C o Pascal), los valores se almacenan directamente dentro de las variables, es decir, las variables son contenedores que almacenan valores.

Por tanto, el compilador tiene que reservar espacio suficiente en la memoria para cada variable del programa de manera que dicha variable pueda contener un dato de un determinado tamaño y que ese dato «quepa» dentro de la variable.

De todos modos, algunos lenguajes de programación tienen un comportamiento híbrido, que combina ambas técnicas:

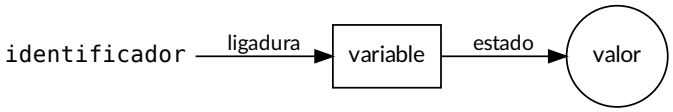
- En Java, existen *tipos primitivos* (cuyos valores se almacenan directamente en las variables) y *tipos referencia* (cuyos valores se almacenan en el montículo y las variables contienen referencias a esos valores).
- En C, los valores se almacenan dentro de las variables, pero es posible reservar memoria dinámicamente dentro del montículo y almacenar en una variable un *puntero* al comienzo de dicha zona de memoria, lo que permite crear y destruir datos en tiempo de ejecución.

2.3. Estado

La **ligadura** es la asociación que se establece entre un identificador y una variable.

El **estado de una variable** es el valor al que hace referencia una variable en un momento dado.

Por tanto, el estado es la asociación que se establece entre una variable y un valor (es decir, la referencia que contiene).



Tanto las ligaduras como los estados pueden cambiar durante la ejecución de un programa imperativo.

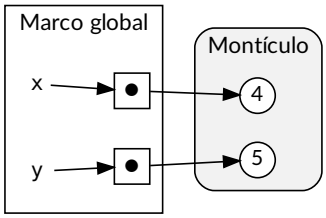
El **estado de un programa** es el conjunto de los estados de todas sus variables (más cierta información auxiliar gestionada por el intérprete).

2.4. Marcos en programación imperativa

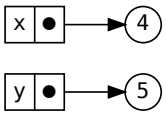
Hasta ahora, los marcos contenían ligaduras entre identificadores y valores.

A partir de ahora, un marco contendrá:

- Las **ligaduras entre identificadores y variables**, y
- El **estado de cada variable**, es decir, la referencia que contiene cada variable en un momento dado.



Para simplificar la representación gráfica, generalmente dibujaremos cada identificador al lado de su correspondiente variable (representando la *ligadura*), y la variable apuntando a su valor en el montículo (representando el *estado*).



El montículo como tal normalmente no lo dibujaremos, ya que sabemos que los valores se almacenan en él.

Igualmente, a veces tampoco dibujaremos el marco si se sobreentiende cuál es (o si no tiene importancia en ese momento).

A veces, y llegado el caso, también dibujaremos el valor directamente almacenado en la variable que le apunta, para simplificar (aunque sabemos que eso no es lo que ocurre en Python).

2.5. Sentencia de asignación

La forma más básica de cambiar el estado de una variable es usando la **sentencia de asignación**.

Es la misma instrucción que hemos estado usando hasta ahora para ligar valores a identificadores, pero ahora, en el paradigma imperativo, tiene otro significado:

```
x = 4      # se lee: «asigna el valor 4 a la variable x»
```

El efecto que produce es el de almacenar, en la variable ligada al identificador `x`, la *identidad* del valor `4` almacenado en el montículo.

A partir de este momento, `x` pasa a ser una referencia al valor `4`.

Normalmente se dice (mal dicho) que «*la variable `x` pasa a valer `4`*».

Se dice que la asignación es **destructiva** porque, al cambiarle el valor a una variable, **el nuevo valor sustituye a su valor anterior** en esa variable.

Por ejemplo, si ahora hacemos:

```
x = 9
```

El valor de la variable a la que está ligada el identificador `x` pasa ahora a ser `9`, sustituyendo el valor `4` anterior.

Por abuso del lenguaje, se suele decir:

«se asigna el valor 9 a la variable x»

o:

«se asigna el valor 9 a la variable ligada al identificador x»

en lugar de la forma correcta:

«se asigna una referencia al valor 9 en la variable ligada al identificador x».

Aunque esto simplifica las cosas a la hora de hablar, hay que tener cuidado, porque llegado el momento es posible tener:

- Varios identificadores distintos ligados a la misma variable (ocurre en algunos lenguajes, aunque no en Python ni Java).
- Un mismo identificador ligado a distintas variables en diferentes puntos del programa.
- Varias variables apuntando al mismo valor.

Cada nueva asignación provoca un cambio de estado en el programa.

En el ejemplo anterior, el programa pasa de estar en un estado en el que la variable `x` vale `4` a otro en el que la variable vale `9`.

Al final, un programa imperativo se puede reducir a una **secuencia de asignaciones** realizadas en el orden dictado por el programa.

Este modelo de funcionamiento está estrechamente ligado a la arquitectura de un ordenador: hay una memoria formada por celdas que contienen datos que pueden cambiar a lo largo del tiempo según dicten las instrucciones del programa que controla al ordenador.

2.5.0.1. Un ejemplo completo

Cuando se ejecuta la siguiente instrucción:

```
x = 2500
```

ocurre lo siguiente:

1. Se crea el valor `2500` en el montículo.

   En determinadas situaciones, no crea un nuevo valor si ya había otro exactamente igual en el montículo, pero éste no es el caso.

2. El intérprete identifica a qué variable está ligado el identificador `x` consultando el entorno (si no existía dicha variable, la crea en ese momento y la liga a `x` en el marco actual).
3. Almacena en la variable una referencia al valor (es decir, la identidad del valor).

2.6. Evaluación de expresiones con variables

Al evaluar expresiones, las variables actúan de modo similar a las ligaduras de la programación funcional, pero ahora los valores de las variables pueden cambiar a lo largo del tiempo, por lo que deberemos *seguirle la pista* a los cambios que sufran dichas variables.

Todo lo visto hasta ahora sobre marcos, ámbitos, sombreado, entornos, etc. se aplica igualmente a las variables.

Por ejemplo:

```
>>> x = 4
>>> y = 3
>>> x * y + 5    # esta expresión vale 17 porque 'x' vale 4 y 'y' vale 3
17
>>> x = 9
>>> x * y + 5    # la misma expresión ahora vale 32 porque 'x' vale 9
32
```

2.7. Constantes

En programación funcional no existen las variables y un identificador sólo puede ligarse a un valor (un identificador ligado no puede re-ligarse a otro valor distinto).

- En la práctica, eso significa que un identificador ligado actúa como un valor constante que no puede cambiar durante la ejecución del programa.
- El valor de esa constante es el valor al que está ligado el identificador.

Pero en programación imperativa, los identificadores se ligan a variables, que son las que realmente apuntan a los valores.

Una **constante** en programación imperativa sería el equivalente a una variable cuyo valor no puede cambiar durante la ejecución del programa.

Muchos lenguajes de programación permiten definir constantes, pero **Python no es uno de ellos**.

En Python, una constante **es una variable más**, pero **es responsabilidad del programador** no cambiar su valor durante todo el programa.

Python no hace ninguna comprobación ni muestra mensajes de error si se cambia el valor de una constante.

En Python, por **convenio**, los identificadores ligados a una variable con valor constante se escriben con todas las letras en **mayúscula**:

```
PI = 3.1415926
```

El nombre en mayúsculas nos recuerda que `PI` es una constante, aunque nada nos impide cambiar su valor (cosa que debemos evitar):

```
PI = 99
```

Sólo es un convenio entre programadores, que no tiene por qué cumplirse siempre.

2.8. Tipado estático vs. dinámico

Cuando una variable tiene asignado un valor, al ser usada en una expresión actúa como si fuera ese valor.

Como cada valor tiene un tipo de dato asociado, también podemos hablar del **tipo de una variable**.

El **tipo de una variable** es el tipo del dato al que hace referencia la variable.

Si a una variable se le asigna otro valor de un tipo distinto al del valor anterior, el tipo de la variable cambia y pasa a ser el del nuevo valor que se le ha asignado.

Eso quiere decir que **el tipo de una variable podría cambiar durante la ejecución del programa**.

A este enfoque se le denomina **tipado dinámico**.

Lenguajes de tipado dinámico:

Son aquellos que **permiten** que el tipo de una variable **cambie** durante la ejecución del programa.

En contraste con los lenguajes de tipado dinámico, existen los llamados **lenguajes de tipado estático**.

En un lenguaje de tipado estático, el tipo de una variable se define una sola vez (en la fase de compilación o justo al empezar a ejecutarse el programa), y **no puede cambiar** durante la ejecución del mismo.

Definición:

Lenguajes de tipado estático:

Son aquellos que asocian forzosamente un tipo a cada variable del programa desde que comienza a ejecutarse y **prohíben** que dicho tipo **cambie** durante la ejecución del programa.

Estos lenguajes disponen de instrucciones que permiten *declarar* de qué tipo serán los datos que se pueden asignar a una variable.

Por ejemplo, en Java podemos hacer:

```
String x;
```

con lo que declaramos que a `x` sólo se le podrán asignar valores de tipo `String` desde el primer momento y a lo largo de toda la ejecución del programa.

A veces, se pueden realizar al mismo tiempo la declaración del tipo y la asignación del valor:

```
String x = "Hola";
```

Otros lenguajes disponen de un mecanismo conocido como **inferencia de tipos**, que permite *deducir* automáticamente el tipo de una variable.

Por ejemplo, en Java podemos hacer:

```
var x = "Hola";
```

El compilador de Java deduce que la variable `x` debe ser de tipo `String` porque se le está asignando una cadena (el valor `"Hola"`).

Normalmente, los lenguajes de tipado estático son también lenguajes compilados y también fuertemente tipados.

Asimismo, los lenguajes de tipado dinámico suelen ser lenguajes interpretados y a veces también son lenguajes débilmente tipados.

Pero nada impide que un lenguaje de tipado dinámico pueda ser compilado, por ejemplo.

Los tres conceptos de:

- Compilado vs. interpretado
- Tipado fuerte vs. débil
- Tipado estático vs. dinámico

son diferentes aunque están estrechamente relacionados.

2.9. Asignación compuesta

Los operadores de **asignación compuesta** nos permiten realizar operaciones sobre una variable y luego asignar el resultado a la misma variable.

Tienen la forma:

⟨*asig_compuesta*⟩ ::= **identificador** ⟨*op*⟩**=** ⟨*expresión*⟩
⟨*op*⟩ ::= **+** | **-** | ***** | **/** | **%** | **//** | ****** | **&** | **|** | **^** | **>>** | **<<**

Operador	Ejemplo	Equivalente a
`=`	`x = 5`	`x = 5`
`+=`	`x += 5`	`x = x + 5`
`-=`	`x -= 5`	`x = x - 5`
`*=`	`x *= 5`	`x = x * 5`
`/=`	`x /= 5`	`x = x / 5`
`%=`	`x %= 5`	`x = x % 5`
`//=`	`x //= 5`	`x = x // 5`
`**=`	`x **= 5`	`x = x ** 5`

Operador	Ejemplo	Equivalente a
`&=`	`x &= 5`	`x = x & 5`
`|=`	`x |= 5`	`x = x | 5`
`^=`	`x ^= 5`	`x = x ^ 5`
`>>=`	`x >>= 5`	`x = x >> 5`
`<<=`	`x <<= 5`	`x = x << 5`

2.10. Asignación múltiple

Con la **asignación múltiple** podemos asignar valores a varias variables **al mismo tiempo** en una sola sentencia.

La sintaxis es:

⟨*asig_múltiple*⟩ ::= ⟨*lista_identificadores*⟩ **=** ⟨*lista_expresiones*⟩
⟨*lista_identificadores*⟩ ::= **identificador** (**,** **identificador**)*
⟨*lista_expresiones*⟩ ::= ⟨*expresión*⟩(**,** ⟨*expresión*⟩)*

con la condición de que tiene que haber tantos identificadores como expresiones.

Por ejemplo:

```
x, y = 10, 20
```

asigna el valor `10` a `x` y el valor `20` a `y`.

Lo interesante de la asignación múltiple es que **todas las asignaciones se llevan a cabo *a la vez, en paralelo***, no una tras otra.

Por ejemplo, si quisiéramos intercambiar los valores de `x` e `y` sin asignación múltiple, tendríamos que usar una variable auxiliar que almacenara el valor de una de las variables para no perderlo:

```
aux = x
x = y
y = aux
```

En cambio, si usamos la asignación múltiple, se puede hacer simplemente:

```
x, y = y, x
```

Lo que ocurre es que la `x` toma el valor que tenía `y` **justo antes de ejecutar la sentencia**, y la `y` toma el valor que tenía `x` **justo antes de ejecutar la sentencia**.

Por tanto, las asignaciones que se realizan en una asignación múltiple **no se afectan entre ellas**.

A la asignación múltiple también se la denomina **desempaquetado de tuplas**, ya que, técnicamente, es una asignación entre dos tuplas, como si se hubiera escrito así:

```
(x, y) = (10, 20)
```

Esto es así porque, en realidad, los paréntesis que rodean a una tupla casi nunca son estrictamente necesarios (salvo para la tupla vacía y para evitar ambigüedades) y, por tanto:

```
2, 3
```

es lo mismo que

```
(2, 3)
```

En consecuencia, lo que ocurre es que *se desempaquetan* las dos tuplas y se asigna cada elemento de la tupla derecha a la variable correspondiente de la tupla izquierda.

3. Mutabilidad

3.1. 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 estado de la variable (haciendo que ahora contenga un nuevo valor) sino **el estado interno** del propio valor al que hace referencia la variable.

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

3.2. 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.

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

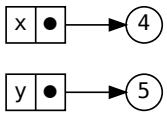
Los principales tipos mutables predefinidos en Python son la lista (`list`), los conjuntos (`set`) y los diccionarios (`dict`).

3.2.1. Inmutables

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

Si tenemos:

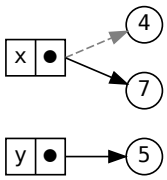
```
x = 4
y = 5
```



y hacemos:

```
x = 7
```

quedaría:

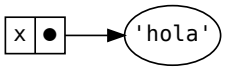


Lo que hace la asignación `x = 7` no es cambiar el contenido del valor `4`, sino hacer que la variable `x` contenga otro valor distinto (el valor `4` en sí mismo no se cambia internamente en ningún momento).

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

Si tenemos:

```
x = 'hola'
```

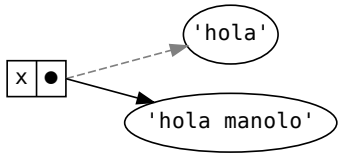


y luego hacemos:

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

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

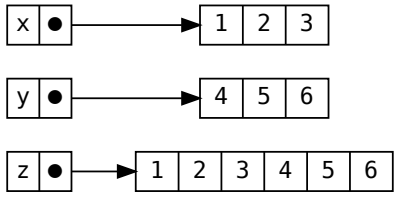
Es decir: la cadena `'hola'` original **no se cambia** (no se le añade `' manolo'` detrás), sino que la nueva **sustituye** a la anterior en la variable:



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

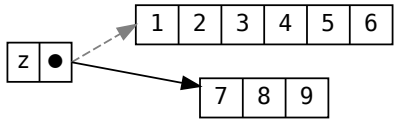
Por ejemplo, si tenemos una tupla `(1, 2, 3)` y le concatenamos otra tupla `(4, 5, 6)`, el resultado es una nueva tupla creada a partir de las otras dos (que permanecen inalteradas):

```
>>> 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)
```



Las **cadenas**, las **tuplas** y los **rangos** 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 individualmente y con los que podemos operar, aunque no podamos cambiarlos, ya que están contenidos en datos compuestos inmutables.

De hecho, las cadenas, las tuplas y los rangos 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, las tuplas y los rangos 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
x **`in`** *s*	`True` si algún elemento de s es igual a x
x **`not in`** *s*	`False` si algún elemento de s es igual a x

Operación	Resultado
`s + t`	La concatenación de s y t (no va con rangos)
`s * n` `n * s`	Equivale a concatenar s consigo misma n veces (no va con rangos)
`s[i]`	El i-ésimo elemento de s, empezando por 0
`s[i:j]`	Rodaja de s desde i hasta j
`s[i:j:k]`	Rodaja de s desde i hasta j con paso k
`len(s)`	Longitud de s
`min(s)`	El elemento más pequeño de s
`max(s)`	El elemento más grande de s
`s.index(x [, i [, j ]])`	El índice de la primera aparición de x en s (desde el índice i inclusive y antes del j)
`s.count(x)`	Número total de apariciones de x en s

El *operador* de **indexación** consiste en acceder al elemento situado en la posición 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.

Los índices negativos (del `-1` hacia atrás) empieza a contar desde el final de la secuencia.

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 *paso* < 0, 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 primer o el último elemento de la secuencia, dependiendo de si *k* es positivo o negativo.

Si se omite *j*, se entiende que queremos la rodaja hasta el último o el primero elemento de la secuencia, dependiendo de si *k* es positivo o negativo.

Si $i > j$, *k* debería ser positivo (de lo contrario, devolvería la secuencia vacía).

Si $i < j$, *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 hasta la posición *n*.
- `s[n:]` es la rodaja desde el elemento *n* hasta el final.

  Siempre se cumple que `s == s[:n] + s[n:]`.
- `s[n::-1]` es la rodaja invertida desde el elemento *n* hasta el principio, con los elementos al revés.
- `s[:]` devuelve una *copia* de `s`.
- `s[::-1]` devuelve una copia invertida de `s`.

Ejercicio

1. Dada la siguiente lista:

```
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]`

3.2.2. Mutables

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

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]
```

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

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

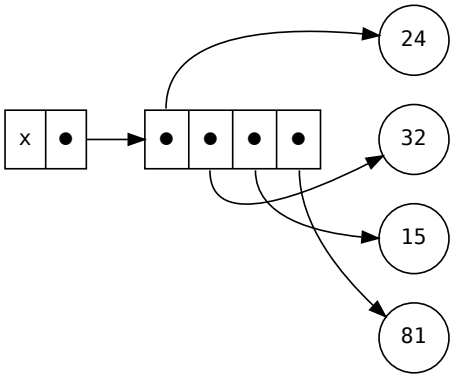
```
  +-----+-----+-----+-----+-----+-----+
l | 124 | 333 | 'a' | 3.2 |  9  |  53 |
  +-----+-----+-----+-----+-----+-----+
  0     1     2     3     4     5     6
 -6    -5    -4    -3    -2    -1
```

```
>>> 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]
```

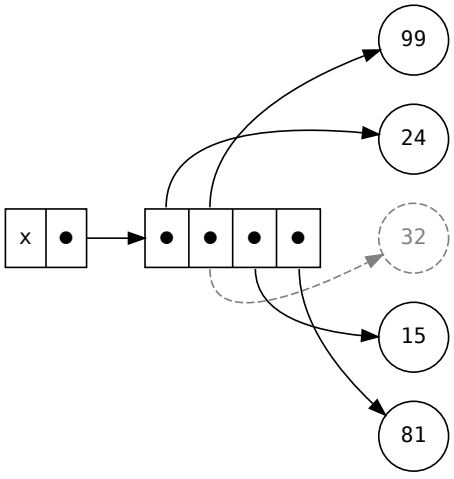
- *Métodos* como `append`, `clear`, `insert`, `remove`, `reverse` o `sort`.

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
`s[i] = x`	El elemento *i*-ésimo de *s* se sustituye por *x*
`s[i:j] = t`	La rodaja de *s* desde *i* hasta *j* se sustituye por *t*
`s[i:j:k] = t`	Los elementos de `s[i:j:k]` se sustituyen por *t*
`del s[i:j]`	Elimina los elementos de `s[i:j]` Equivale a hacer `s[i:j] = []`
`del s[i:j:k]`	Elimina los elementos de `s[i:j:k]`

Operación	Resultado
`s.append(x)`	Añade *x* al final de *s*; es igual que `s[len(s):len(s)] = [x]`
`s.clear()`	Elimina todos los elementos de *s*; es igual que `del s[:]`
`s.copy()`	Crea una copia *superficial* de *s*; es igual que `s[:]`
`s.extend(t)` `s += t`	Extiende *s* con el contenido de *t*; es igual que `s[len(s):len(s)] = t`
`s *= n`	Modifica *s* repitiendo su contenido *n* veces
`s.insert(i, x)`	Inserta *x* en *s* en el índice *i*; es igual que `s[i:i] = [x]`
`s.pop([i])`	Extrae el elemento *i* de *s* y lo devuelve (por defecto, *i* vale `-1`)
`s.remove(x)`	Quita el primer elemento de *s* que sea igual a *x*
`s.reverse()`	Invierte los elementos de *s*
`s.sort()`	Ordena los elementos de *s*

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

Ejemplo	Valor de `x` después
`x.append(14)`	`[8, 10, 7, 9, 14]`
`x.clear()`	`[]`
`x.insert(3, 66)`	`[8, 10, 7, 66, 9]`
`x.remove(7)`	`[8, 10, 9]`

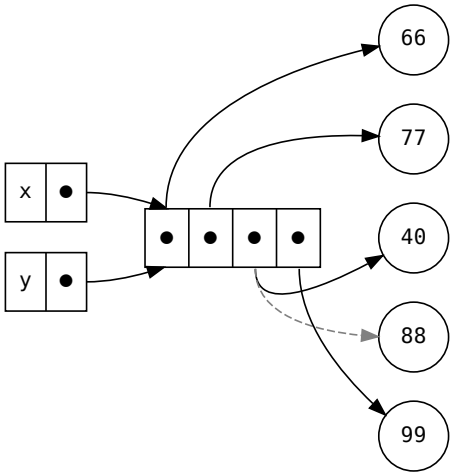
Ejemplo	Valor de x después
`x.reverse()`	`[9, 7, 10, 8]`

3.3. Alias de variables

Cuando una variable que tiene un valor se asigna a otra, ambas variables pasan a tener **el mismo valor (lo *comparten*)**, produciéndose un 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).



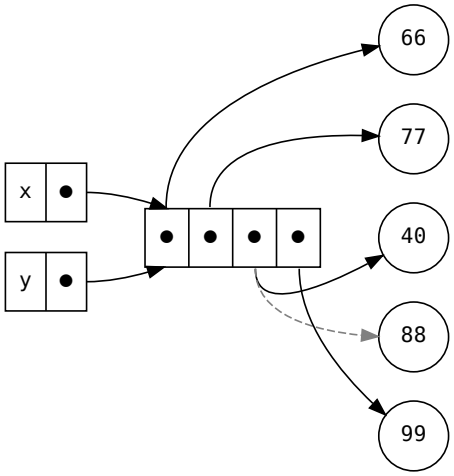
Si el valor es **mutable** y cambiamos su **contenido** desde x, también cambiará y, pues ambas variables **apuntan al mismo dato**:

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

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

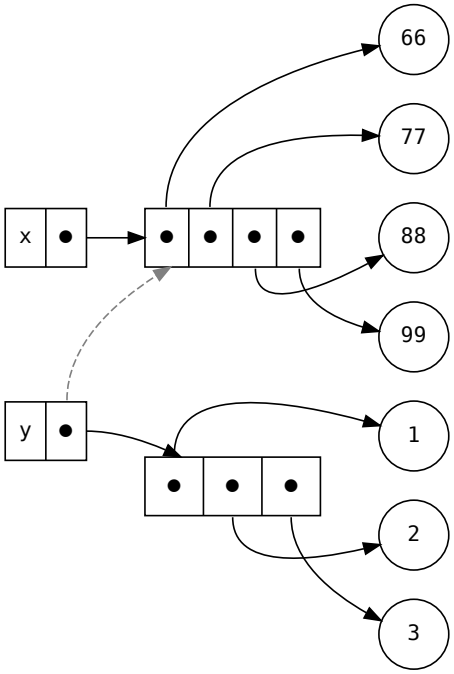
Cambiar el **contenido** es algo que sólo se puede hacer si el valor es **mutable** (por ejemplo, cambiando un elemento de una lista):

```
>>> 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]
```



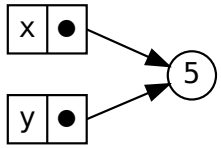
Cuando los valores son inmutables, no importa si se comparten o no, ya que no se pueden modificar.

De hecho, el intérprete a veces crea valores nuevos y a veces comparte los ya existentes.

Por ejemplo, el intérprete de Python crea internamente todos los números enteros comprendidos

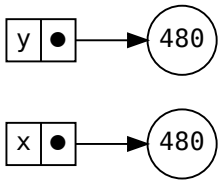
entre −5 y 256, por lo que todas las variables de nuestro programa que contengan el mismo número dentro de ese intervalo compartirán el mismo valor (serán *alias*):

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



Se comparte el valor

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

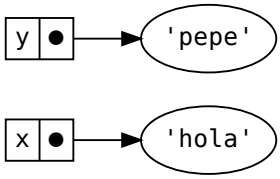


No se comparte el valor

También crea valores compartidos cuando contienen exactamente las mismas cadenas.

Si tenemos:

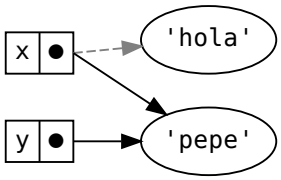
```
x = 'hola'
y = 'pepe'
```



y hacemos:

```
x = 'pepe'
```

quedaría:



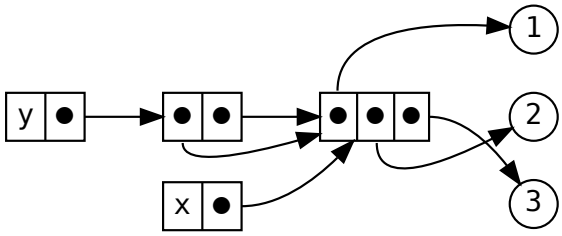
El intérprete aprovecharía la cadena ya creada y no crearía una nueva, para ahorrar memoria.

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

Por ejemplo, si hacemos:

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

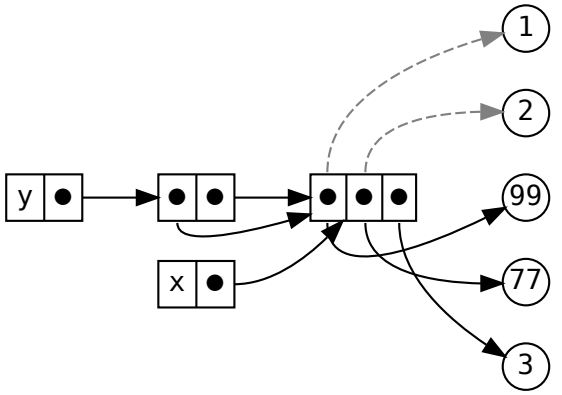
se compartiría la lista `x`, por lo que nos quedaría:



Y si ahora hacemos:

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

nos quedaría:



3.3.1. Recolección de basura

Un valor se vuelve **inaccesible** cuando no hay ninguna referencia que apunte a él.

Eso ocurre cuando no queda ninguna variable que contenga una referencia a ese dato.

En tal caso, el intérprete lo marca como *candidato para ser eliminado*.

Cada cierto tiempo, el intérprete activa el **recolector de basura**, que es un componente que se encarga de liberar de la memoria a los valores que están marcados como candidatos para ser eliminados.

Por tanto, el programador Python no tiene que preocuparse de gestionar manualmente la memoria ocupada por los datos que componen su programa.

Por ejemplo:

```
lista1 = [1, 2, 3]  # crea la lista y guarda una referencia a ella en lista1
lista2 = lista1     # almacena en lista2 la referencia que hay en lista1
```

A partir de ahora, ambas variables apuntan al mismo dato y, por tanto, decimos que el dato tiene dos referencias o que hay dos referencias apuntándole.

```
del lista1          # elimina una referencia pero el dato aún tiene otra
del lista2          # elimina la otra referencia y ahora el dato es inaccesible
```

Como ya no hay ninguna referencia apuntándole, se marca como *candidato a ser eliminado* y, por tanto, la próxima vez que se active el recolector de basura, se eliminará la lista del montículo.

3.3.2. `id`

Para saber si dos variables comparten **el mismo dato**, se puede usar la función `id`.

La función `id` aplicada a un dato devuelve la **identidad** del dato, es decir, el **identificador único** que se utiliza para localizar al dato dentro del montículo.

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

En consecuencia, si dos variables tienen el mismo `id`, significa que ambas apuntan al mismo dato en la memoria y, por tanto, son **referencias al 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
```

3.3.3. `is`

Otra forma 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.

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

No tiene sentido usarlo con literales (y el intérprete lo advierte).

En la práctica, equivale a hacer `id(`⟨*valor1*⟩`) == id(`⟨*valor2*⟩`)`.

```
>>> x = 500
>>> y = 500
>>> x is y
False
>>> 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
```

4. Cambios de estado ocultos

4.1. 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.

4.2. Funciones impuras

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

- si su valor de retorno o su comportamiento dependen de algo más que de sus argumentos, o

- si 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).

4.3. Efectos laterales

Un **efecto lateral** es cualquier cambio de estado llevado a cabo por una parte del programa (normalmente, una función) que puede observarse desde otras partes del mismo, las cuales podrían verse afectadas por él de una manera poco evidente o impredecible.

> Una función puede provocar efectos laterales, o bien 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.

4.4. Transparencia referencial

En un lenguaje imperativo **se pierde la transparencia referencial**, ya que ahora el valor de una función puede depender no sólo de los valores de sus argumentos, sino también además de los valores de las variables libres que ahora 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 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, en ella no se 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 muy 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 ocultos** 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.

4.5. Entrada y salida por consola

Nuestro programa puede comunicarse con el exterior realizando **operaciones de entrada/salida**.

Interpretamos la palabra *exterior* en un sentido amplio; por ejemplo:

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

La entrada/salida por consola se refiere a las operaciones de lectura de datos por el teclado y escritura por la pantalla.

Las operaciones de entrada/salida se consideran **efectos laterales** porque producen cambios en el exterior o pueden hacer que el resultado de una función dependa de los datos leídos y, por tanto, no depender sólo de sus argumentos.

4.5.1. `print`

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

Su signatura 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
```

4.5.1.1. 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.

Por ejemplo:

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

4.5.1.2. 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).

La función `print` como tal no devuelve ningún valor, pero como en Python todas las funciones devuelven *algún* valor, en realidad lo que ocurre es que **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`.

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
```

4.5.2. `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 signatura es:

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

Por ejemplo:

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

Provoca el *efecto lateral* de alterar el estado de la consola imprimiendo el *prompt* y esperando a que desde el exterior se introduzca el dato solicitado (que en cada ejecución podrá tener un valor distinto).

Eso hace que sea *impura* por partida doble: provoca un efecto lateral y puede devolver un resultado distinto cada vez que se la llama.

4.6. Entrada y salida por archivos

Para leer y/o escribir datos en un archivo, los pasos a seguir son (en este orden):

1. Abrir el archivo en el modo adecuado con `open`.
2. Realizar las operaciones deseadas sobre el archivo.
3. Cerrar el archivo con `close`.

4.6.1. `open`

La función `open` abre un archivo y devuelve un objeto que lo representa. Su signatura es:

```
open(nombre: str [, modo: str])
```

El *nombre* es una cadena que contiene el nombre del archivo a abrir.

El *modo* es otra cadena que contiene caracteres que describen de qué forma se va a usar el archivo.

El valor devuelto es un objeto cuyo tipo depende del modo en el que se ha abierto el archivo.

Los valores posibles de *modo* son:

Carácter	Significado
`'r'`	Abre el archivo para lectura (valor predeterminado)

Carácter	Significado
`'w'`	Abre para escritura (si el archivo ya existe lo borrará)
`'x'`	Abre para creación exclusiva (falla si el archivo ya existe)
`'a'`	Abre para escritura, añadiendo al final del archivo si ya existe
`'b'`	Modo binario
`'t'`	Modo texto (valor predeterminado)
`'+'`	Abre para lectura y escritura

El modo predeterminado es `'r'`.

Los modos `'w+'` y `'w+b'` abren el archivo y lo borra si ya existe.

Los modos `'r+'` y `'r+b'` abren el archivo sin borrarlo.

Normalmente, los archivos se abren en **modo texto**, lo que significa que se leen y se escriben cadenas (valores de tipo `str`) desde y hacia el archivo, las cuales se codifican según una codificación específica que depende de la plataforma.

Por ejemplo, los saltos de línea se escriben como `\n` en Unix o `\r\n` en Windows, y se leen siempre como `\n`.

Al añadir una `b` en el modo se abre el archivo en **modo binario**. En tal caso, los datos se leen y se escriben en forma de objetos de tipo `bytes`.

El modo binario es el que debe usarse cuando se trabaje con archivos que no contengan texto (datos binarios *crudos*).

Ejemplo:

```
f = open('salida.txt', 'w')
```

El tipo de dato que devuelve `open` depende de cómo se ha abierto el archivo:

- Si se ha abierto en **modo texto**, devuelve un `TextIOWrapper`.
- Si se ha abierto en **modo binario**, entonces depende:
  - En modo **sólo lectura**, devuelve un `BufferedReader`.
  - En modo **sólo escritura o añadiendo al final**, devuelve un `BufferedWriter`.
  - En modo **lectura/escritura**, devuelve un `BufferedRandom`.

4.6.2. `read`

Para leer de un archivo, se puede usar el método `read` sobre el objeto que devuelve la función `open`. Su signatura es:

```
⟨archivo⟩.read([tamaño: str])
```

El método devuelve una cadena (tipo `str`) si el archivo se abrió en modo texto, o un objeto de tipo `bytes` si se abrió en modo binario.

El archivo contiene un **puntero interno** que indica hasta dónde se ha leído en el mismo. Cada vez que se llama al método `read`, se mueve ese puntero para que en posteriores llamadas se continúe leyendo desde ese punto.

Si se alcanza el final del archivo, se devuelve la cadena vacía (`''`).

El parámetro *tamaño* es opcional:

- Si se omite o es negativo, se devuelve todo lo que hay desde la posición actual del puntero hasta el final del archivo.
- En caso contrario, se leerán y devolverán *al menos* tantos caracteres (en modo texto) o bytes (en modo binario) como se haya indicado.

Ejemplos de lectura de todo el archivo:

```
>>> f = open('entrada.txt', 'r')
>>> f.read()
'Este es el contenido del archivo.\n'
>>> f.read()
''
```

Ejemplos de lectura del archivo en varios trozos:

```
>>> f = open('entrada.txt', 'r')
>>> f.read(4)
'Este'
>>> f.read(4)
' es '
>>> f.read(4)
'el c'
>>> f.read()
'ontenido del archivo\n'
>>> f.read()
''
```

4.6.3. `readline`

El método `readline` también sirve para leer de un archivo y también se ejecuta sobre el objeto que devuelve `open`.

Su signatura es:

```
⟨archivo⟩.readline([tamaño: str])
```

`readline` devuelve una línea del archivo, dejando el carácter de salto de línea (`\n`) al final.

El salto de línea sólo se omite cuando es la última línea del archivo y éste no acaba en salto de línea.

Si devuelve una cadena vacía (`''`), significa que se ha alcanzado el final del archivo.

Si se devuelve una cadena formada sólo por `\n`, significa que es una línea en blanco (una línea que sólo contiene un salto de línea).

El parámetro *tamaño* es opcional:

- Si se omite o es negativo, se devuelve todo desde la posición actual del puntero hasta el final de la línea.
- En caso contrario, se leerán y devolverán *al menos* tantos caracteres (en modo texto) o bytes (en modo binario) como se haya indicado.

Ejemplos:

```
>>> f = open('entrada.txt', 'r')
>>> f.readline()
'Esta es la primera línea.\n'
>>> f.readline()
'Esta es la segunda.\n'
>>> f.readline()
'Y esta es la tercera.\n'
>>> f.readline()
''
```

```
>>> f = open('entrada.txt', 'r')
>>> f.readline(4)
'Esta'
>>> f.readline(4)
' es '
>>> f.readline()
'la primera línea.\n'
>>> f.readline()
'Esta es la segunda.\n'
>>> f.readline()
'Y esta es la tercera.\n'
>>> f.readline()
''
```

4.6.4. `write`

El método `write` sirve para escribir en un archivo y se ejecuta sobre el objeto que devuelve `open`.

Su signatura es:

```
⟨archivo⟩.write(contenido)
```

El método escribe el *contenido* en el ⟨*archivo*⟩. Ese contenido debe ser una *cadena* si el archivo se abrió en **modo texto**, o un *valor de tipo* `bytes` si se abrió en **modo binario**.

Al escribir, modifica el puntero interno del archivo.

Devuelve el número de caracteres o de bytes que se han escrito, dependiendo de si se abrió en modo texto o en modo binario.

También se puede usar `print` para escribir en un archivo.

En la práctica, no hay mucha diferencia entre usar `print` y usar `write`.

Hacer:

```
>>> f = open('archivo.txt', 'r+')
>>> f.write('Hola Manolo\n')
```

equivale a hacer:

```
>>> f = open('archivo.txt', 'r+')
>>> print('Hola', 'Manolo', file=f)
```

Hay que tener en cuenta los separadores y los saltos de línea que introduce `print`.

`print` escribe en el archivo `sys.stdout` mientras no se diga lo contrario.

4.6.5. `seek` y `tell`

El método `seek` situa el puntero interno del archivo en una determinada posición.

El método `tell` devuelve la posición actual del puntero interno.

Sus signaturas son:

```
⟨archivo⟩.seek(offset: int) -> int

⟨archivo⟩.tell() -> int
```

El *offset* es la posición a la que se desea mover el puntero, empezando por 0 desde el comienzo del archivo.

Además de mover el puntero, el método `seek` devuelve la nueva posición del puntero.

Por ejemplo:

```
>>> f = open('archivo.txt', 'r+')       # abre en modo lectura/escritura
>>> f.tell()
0
>>> f.readline()
'Esta es la primera línea.\n'
>>> f.tell()
27
>>> f.seek(0)
0
>>> f.readline()
'Esta es la primera línea.\n'
>>> f.seek(0)
0
>>> f.write('Cambiar')
7
>>> f.tell()
7
>>> f.seek(0)
0
```

```
>>> f.readline()
'Cambiar la primera línea.\n'
```

4.6.6. `close`

El método `close` cierra un archivo previamente abierto por `open`, finalizando la sesión de trabajo con el mismo.

Su signatura es:

```
⟨archivo⟩.close()
```

Siempre hay que cerrar un archivo previamente abierto, para así asegurarse de que los cambios realizados se vuelquen al archivo a través del sistema operativo y liberar inmediatamente los recursos del sistema que pudiera estar consumiendo.

Una vez que se ha cerrado el archivo ya no se podrá seguir usando:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Podemos comprobar si un archivo ya se ha cerrado consultando su atributo `closed`:

```
>>> f = open('archivo.txt', 'r')
>>> f.closed
False
>>> f.close()
>>> f.closed
True
```

Observa que no es un método (no lleva paréntesis), sino un atributo que contiene directamente un valor lógico que el propio objeto modifica al cambiar su estado de abierto a cerrado o viceversa.

5. Saltos

5.1. Incondicionales

Un **salto incondicional** es una ruptura abrupta del flujo de control del programa hacia otro punto del mismo.

Se llama *incondicional* porque no depende de ninguna condición, es decir, se lleva a cabo **siempre** que se alcanza el punto del salto.

Históricamente, a esa instrucción que realiza saltos incondicionales se la ha llamado **instrucción *GOTO***.

El uso de instrucciones *GOTO* es considerado, en general, una mala práctica de programación ya que favorece la creación del llamado **código *espagueti***: programas con una estructura de control tan

complicada que resultan casi imposibles de mantener.

En cambio, usados controladamente y de manera local, puede ayudar a escribir soluciones sencillas y claras.

Python no incluye la instrucción *GOTO* pero se puede simular usando el módulo `with_goto` del paquete llamado `goto-statement`:

```
$ sudo apt install python3-pip
$ python3 -m pip install goto-statement
```

Sintaxis:

⟨*goto*⟩ ::= **goto** ⟨*etiqueta*⟩
⟨*label*⟩ ::= **label** ⟨*etiqueta*⟩
⟨*etiqueta*⟩ ::= **.**⟨*identificador*⟩

Un ejemplo de uso:

```
from goto import with_goto

CODIGO = """
print('Esto se hace')
goto .fin
print('Esto se salta')
label .fin
print('Aquí se acaba')
"""

exec(with_goto(compile(CODIGO, '', 'exec')))
```

5.2. Condicionales

Un **salto condicional** es un salto que se lleva a cabo sólo si se cumple una determinada condición.

En Python, usando el módulo `with_goto`, podríamos implementarlo de la siguiente forma:

⟨*salto_condicional*⟩ ::= **if** ⟨*condición*⟩**: goto** ⟨*etiqueta*⟩

Ejemplo de uso:

```
from goto import with_goto

CODIGO = """
primero = 2
ultimo = 25

i = primero

label .inicio
if i == ultimo: goto .fin

print(i, end=' ')
```

```
i += 1
goto .inicio

label .fin
"""

exec(with_goto(compile(CODIGO, '', 'exec')))
```

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.