Programación imperativa

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Modelo de ejecución

1.1. Máquina de estados

La **programación imperativa** es un paradigma de programación basado en el concepto de **sentencia**.

Un programa imperativo está formado por una sucesión de sentencias que se ejecutan **en orden**.

Una sentencia es una instrucción del programa que lleva a cabo una de estas acciones:

- **Cambiar el estado interno** del 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.

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

En programación imperativa, el concepto de **tiempo** cobra mucha importancia: el programa actuará de una forma un otra según el estado en el que se encuentre, es decir, según el momento en el que estemos observando al programa.

Eso significa que, ante los mismos datos de entrada, una función 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.

Eso significa que, para modelar el comportamiento de un programa imperativo, ya **no nos vale el modelo de sustitución**.

1.2. Secuencia de instrucciones

Un programa imperativo es una **secuencia de instrucciones**, y ejecutar un programa es provocar los **cambios de estado** que dictan las instrucciones en el **orden** definido por el programa.

Las instrucciones 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 instrucción por instrucción y llevando la cuenta de los valores ligados a sus identificadores.

1.3. Sentencias

A las instrucciones de un programa imperativo también se las denomina **sentencias**.

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 cambiar el estado de éste.

- Las *expresiones* se *evalúan*.
- Las *sentencias* se *ejecutan*.

En muchos lenguajes imperativos (como ocurre con Python y Java) es posible colocar una expresión donde se espera una sentencia (aunque no al revés), si bien no suele resultar útil ya que, en ese caso, la ejecución de tal «sentencia» consistiría en evaluar la expresión, pero el resultado de dicha evaluación se perdería.

Sólo resultaría útil en caso de que la evaluación de la expresión provocara *efectos laterales* (cosa que estudiaremos en breve).

2. Asignación destructiva

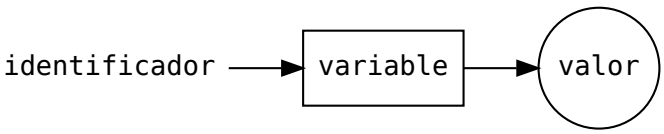
2.1. Variables

Una **variable** es un lugar en la **memoria** donde se puede **almacenar un valor**.

El valor de una variable **puede cambiar** 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 almacena el valor**.

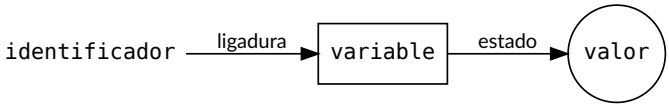


2.2. Estado

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

El **estado de una variable** es el valor que tiene una variable en un momento dado.

Por tanto, el estado es la asociación que se establece entre una variable y un valor.



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

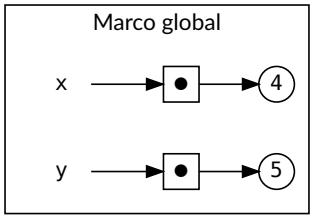
El **estado de un programa** es el conjunto de los estados de todas sus variables (más cierta información auxiliar gestionada por el intérprete).

2.3. Marcos en programación imperativa

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

A partir de ahora, un marco contendrá:

- El conjunto de las **ligaduras entre identificadores y variables**, y
- El **estado de cada variable**, es decir, el conjunto de las variables y el valor que contiene cada una en un momento dado.



2.4. 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
```

Significa que el identificador `x` está ligado a una variable cuyo valor pasa a ser `4`.

La asignación es **destructiva** porque al cambiar un valor a una variable se destruye su valor anterior. 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`, perdiéndose el valor `4` anterior.

Por abuso del lenguaje, se suele decir:

«se asigna el valor `9` a la variable `x`»

en lugar de:

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

Aunque esto simplifica las cosas a la hora de hablar, hay que tener cuidado, porque llegará el momento en el que podamos tener:

- Varios identificadores distintos ligados a la misma variable.
- Un mismo identificador ligado a distintas variables en diferentes puntos del programa.

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. Evaluación de expresiones con variables

Al evaluar expresiones, las variables actúan de modo similar a las ligaduras de la programación funcional, con la única diferencia de que su valor puede cambiar a lo largo del tiempo, por lo que deberemos *seguirle la pista* a las asignaciones que sufra dicha variable.

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

2.6. 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.

En programación imperativa, los identificadores se ligan a variables, que son las que realmente contienen 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 un 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
```

2.7. 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 que contiene 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 construcciones sintácticas que permiten declarar de qué tipo serán los datos que puede contener una variable.

Por ejemplo, en Java podemos hacer:

```
int x;
```

con lo que declaramos que `x` sólo podrá contener valores de tipo `int` 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:

```
int x = 24;
```

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 = 24;
```

El compilador de Java deduce que la variable `x` debe ser de tipo `int` porque se le está asignando un valor entero (el `24`).

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.

2.8. 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`

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`
`<<=`	`x <<= 5`	`x = x << 5`

2.9. 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`.

3. Mutabilidad

3.1. 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 (su **contenido**) puede cambiar durante la ejecución del programa.

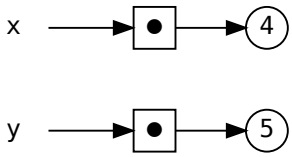
El principal tipo mutable en Python es la lista (`list`), pero también están los conjuntos (`set`) y los diccionarios (`dict`).

3.1.1. Inmutables

Un valor de un tipo inmutable no puede cambiar su contenido.

Por ejemplo, si tenemos:

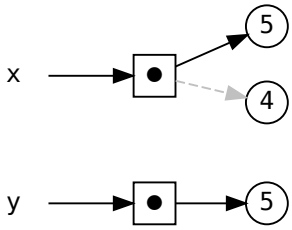
```
x = 4
y = 5
```



y hacemos:

```
x = 5
```

quedaría:

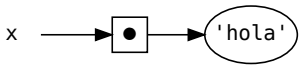


Lo que hace la asignación `x = 5` no es cambiar el contenido del valor `4`, sino hacer que la variable `x` contenga otro valor distinto (el contenido del valor `4` en sí mismo no se modifica internamente en ningún momento).

Con las cadenas sería exactamente igual.

Si tenemos:

```
x = 'hola'
```

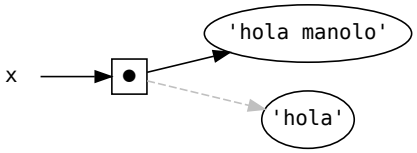


y luego hacemos:

```
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** (p. ej., no se le añade `' manolo'` al final), sino que desaparece y **se sustituye por una nueva**.



Aunque las cadenas son datos inmutables, también son datos compuestos y podemos acceder individualmente a sus elementos componentes y operar con ellos aunque no podamos cambiarlos.

Para ello podemos usar las operaciones comunes a toda secuencia de elementos (una cadena también es una **secuencia de caracteres**):

Operación	Resultado
x `in` *s*	`True` si *x* está en *s*
x `not in` *s*	`True` si *x* **no** está en *s*
s`[`*i*`]`	(*Indexación*) El *i*-ésimo elemento de *s*, empezando por 0
s`[`*i*`:`*j*`]`	(*Slicing*) Rodaja de *s* desde *i* hasta *j*
s`[`*i*`:`*j*`:`*k*`]`	Rodaja de *s* desde *i* hasta *j* con paso *k*
s`.index(`*x*`)`	Índice de la primera aparición de *x* en *s*
s`.count(`*x*`)`	Número de veces que aparece *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'
```

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:

```
     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'
```

3.1.2. Mutables

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

Hasta ahora, hemos visto un único tipo mutable: la **lista**.

Una lista puede cambiar el valor de sus elementos, aumentar o disminuir de tamaño.

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

```
>>> x = [24, 32, 15, 81]
>>> x[0] = 99
>>> x
[99, 32, 15, 81]
```

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`.

Las siguientes tablas muestran todas las **operaciones** que nos permiten **modificar secuencias mutables**.

(*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* Equivale a hacer `s[len(s):len(s)] = [x]`
`s.clear()`	Elimina todos los elementos de *s* Equivale a hacer `del s[:]`
`s.extend(t)` ó `s += t`	Amplía *s* con el contenido de *t* Equivale a hacer `s[len(s):len(s)] = t`
`s.insert(i, x)`	Inserta *x* en *s* en el índice *i* Equivale a hacer `s[i:i] = [x]`
`s.pop([i = -1])`	Devuelve el elemento *i*-ésimo y lo elimina de *s*
`s.remove(x)`	Elimina el primer elemento de *s* que sea igual a *x*

Operación	Resultado
`s.reverse()`	Invierte los elementos de *s*

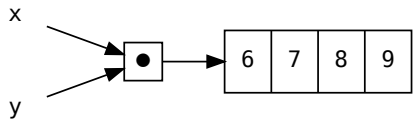
Partiendo de `x = [8, 10, 7, 9]`:

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]`
`x.reverse()`	`[9, 7, 10, 8]`

3.2. Alias de variables

Cuando una variable que contiene un valor se asigna a otra, se produce un fenómeno conocido como **alias de variables**, según el cual los dos identificadores se ligan a **la misma variable** (la comparten):

```
x = [6, 7, 8, 9]
y = x  # x se asigna a y
```



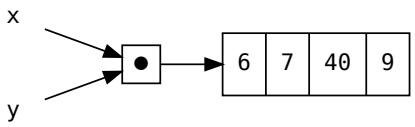
Ahora, si el valor es **mutable** y cambiamos su **contenido** desde `x`, también cambiará `y` (y viceversa), pues ambas son **la misma variable**:

```
>>> y[2] = 40
>>> x
[6, 7, 40, 9]
```

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

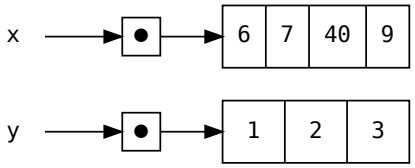
Cambiar el **contenido** es algo que sólo se puede hacer si el valor es **mutable** (por ejemplo, cambiando un elemento de una lista):

```
>>> x = [6, 7, 8, 9]
>>> y = x
>>> y[2] = 40
```



Cambiar el **valor** es algo que **siempre** se puede hacer simplemente **asignando** a la variable **un nuevo valor**:

```
>>> x = [6, 7, 8, 9]
>>> y = x
>>> y = [1, 2, 3]
```

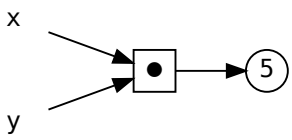


El intérprete puede crear alias de variables **implícitamente** para ahorrar memoria y sin que seamos conscientes de ello.

No tiene mucha importancia práctica, aunque es interesante saberlo en ciertos casos.

Por ejemplo, el intérprete de Python crea internamente una variable para cada número entero comprendido entre -5 y 256, por lo que todas las variables de nuestro programa que contengan el mismo valor dentro de ese intervalo compartirán el mismo espacio en memoria (serán alias):

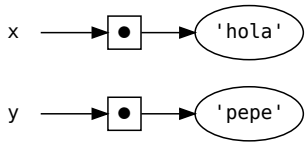
```
x = 5
y = 5
```



También crea variables compartidas cuando contienen exactamente las mismas cadenas.

Si tenemos:

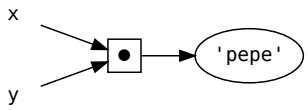
```
x = 'hola'
y = 'pepe'
```



y hacemos:

```
x = 'pepe'
```

quedaría:



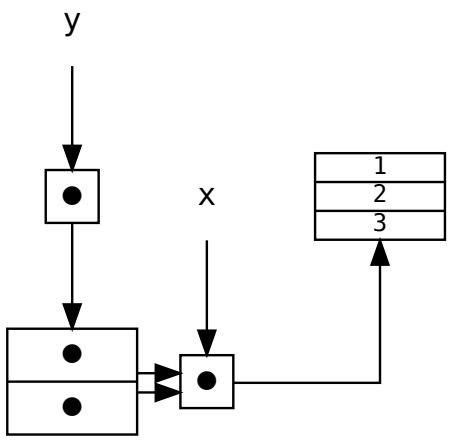
El intérprete aprovecharía la variable ya creada y no crearía una nueva, para ahorrar memoria.

También se comparten variables si se usa el mismo dato varias veces.

Por ejemplo, si hacemos:

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

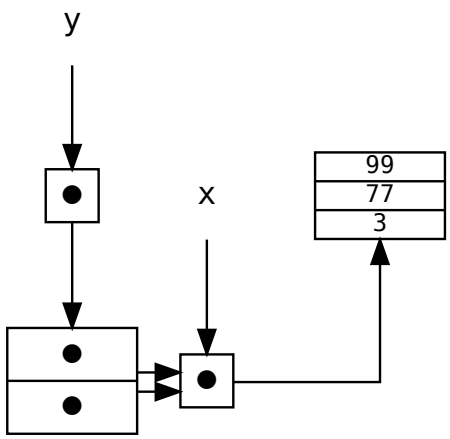
nos quedaría:



Y si ahora hacemos:

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

nos quedaría:



3.2.1. Identidad e igualdad

En el momento en que introducimos mutabilidad en nuestro modelo computacional, muchos conceptos que antes eran sencillos se vuelven problemáticos.

Consideremos el concepto de que dos cosas sean «la misma cosa».

Supongamos que hacemos:

```
restador = lambda cantidad: lambda x: x - cantidad
res1 = restador(25)
res2 = restador(25)
```

¿Son `res1` y `res2` la misma cosa? Una respuesta aceptable podría ser que sí, ya que tanto `res1` como `res2` se comportan de la misma forma (los dos son funciones que restan 25 a su argumento). De hecho, `res1` puede sustituirse por `res2` en cualquier lugar de un programa sin que cambie el resultado.

En cambio, supongamos que tenemos dos listas:

```
l1 = [1, 2, 3]
l2 = [1, 2, 3]
```

¿Son `l1` y `l2` la misma cosa? Evidentemente no, ya que al cambiar una no se observa el mismo cambio en la otra:

```
>>> l1.append(4)
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3]
```

Incluso aunque podamos pensar que `l1` y `l2` son «iguales» en el sentido de que ambos han sido creados evaluando la misma expresión (`[1, 2, 3]`), no es verdad que podamos sustituir `l1` por `l2` en cualquier expresión sin cambiar el resultado de evaluar dicha expresión.

Es otra forma de decir que con los datos mutables no hay transparencia referencial, ya que se pierde en el momento en que incorporamos estado y mutabilidad en nuestro modelo computacional.

Pero al perder la transparencia referencial, la noción de lo que significa que dos datos sean «el mismo datos» se convierte en algo difícil de capturar de una manera formal. De hecho, el significado de «el mismo» en el mundo real que estamos modelando con nuetro programa es ya difícil de entender.

En general, sólo podemos determinar que dos datos aparentemente idénticos son realmente «el mismo dato» modificando uno de ellos y observando a continuación si el otro se ha cambiado de la misma forma.

Pero, ¿cómo podemos decir si un dato ha «cambiado» si no es observando el «mismo» dato dos veces y comprobando si ha cambiado alguna propiedad del dato de la primera observación a la siguiente?

Por tanto, no podemos decir que ha habido un «cambio» sin alguna noción previa de «igualdad», y no podemos determinar la igualdad sin observar los efectos del cambio.

Un ejemplo de cómo puede afectar este problema en la programación, consideremos el caso en que Pedro y Pablo tienen un depósito con 100 € cada uno. Hay una enorme diferencia entre definirlo así:

```
dep_Pedro = Deposito(100)
dep_Pablo = Deposito(100)
```

y definirlo así:

```
dep_Pedro = Deposito(100)
dep_Pablo = dep_Pedro
```

En el primer caso, los dos depósitos son distintos. Las operaciones realizadas por Pedro no afectarán a la cuenta de Pablo, y viceversa. En el segundo caso, en cambio, hemos definido a `dep_Pablo` para que sea exactamente la misma cosa que `dep_Pedro`. Por tanto, ahora Pedro y Pablo son cotitulares de un depósito compartido, y si Pedro hace una retirada de efectivo a través de `dep_Pedro`, Pablo observará que hay menos dinero en `dep_Pablo`.

Estas dos situaciones, similares pero distintas, pueden provocar confusión al crear modelos computacionales. Con el depósito compartido, en particular, puede ser especialmente confuso el hecho de que haya un objeto (el depósito) con dos nombres distintos (`dep_Pedro` y `dep_Pablo`). Si estamos buscando todos los sitios de nuestro programa donde pueda cambiarse el depósito de `dep_Pedro`, tendremos que recordar buscar también los sitios donde se cambie a `dep_Pablo`.

Con respecto a los anteriores comentarios sobre «igualdad» y «cambio», obsérvese que si Pedro y Pablo sólo pudieran comprobar sus saldos y no pudieran realizar operaciones que cambiaran los fondos del depósito, entonces no existiría el problema de comprobar si los dos depósitos son distintos. En general, siempre que no se puedan modificar los objetos, podemos suponer que un objeto compuesto se corresponde con la totalidad de sus partes.

Por ejemplo, un número racional está determinado por su numerador y su denominador. Pero este punto de vista deja de ser válido cuando incorporamos mutabilidad, donde un objeto compuesto tiene una «identidad» que es algo distinto de las partes que lo componen. Un depósito sigue siendo «el mismo» depósito aunque cambiemos sus fondos haciendo una retirada de efectivo. Igualmente, podemos tener dos depósitos distintos con el mismo estado interno.

Esta complicación es consecuencia, no de nuestro lenguaje de programación, sino de nuestra percepción del depósito bancario como un objeto. Por ejemplo, no tendría sentido para nosotros consi-

derar que un número racional es un objeto mutable con identidad ya que al cambiar su numerador no tendríamos «el mismo» número racional.

3.2.2. `id`

Para saber si dos variables son realmente **la misma variable**, se puede usar la función `id`.

La **función `id`** devuelve un identificador único para cada dato en memoria.

Por tanto, si dos variables tienen el mismo `id`, significa que son realmente la misma variable (están situadas en la misma celda de la memoria).

```
>>> 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
```

3.2.3. `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 la **identidad** de un dato:

Su sintaxis es:

⟨*is*⟩ ::= ⟨*valor1*⟩ **`is`** ⟨*valor2*⟩

Es un operador relacional que devuelve `True` si ⟨*valor1*⟩ y ⟨*valor2*⟩ son **el mismo dato en memoria** (es decir, si se encuentran almacenados en la misma celda de la memoria y, por tanto, son **idénticos**) y `False` en caso contrario.

Cuando se usa con variables, devuelve `True` si son la misma variable, y `False` en caso contrario.

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

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.

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

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 es considerado un **efecto lateral**, ya que puede alterar el comportamiento de otras partes del programa de formas a menudo impredecibles.

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 que es observable fuera de la función.

Por ejemplo, una función que imprime algo por la pantalla o escribe algo en un archivo del disco 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, puesto que 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 sintaxis es:

```
⟨print⟩ ::= 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'` (el carácter de nueva línea).

Las expresiones se convierten en cadenas antes de imprimirse.

Por ejemplo:

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

4.5.1.1. 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 desde la entrada (normalmente el teclado) y devuelve el valor del dato introducido.

Siempre devuelve una **cadena**.

Su sintaxis es:

⟨*input*⟩ ::= **input(**[⟨*prompt*⟩]**)**

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).

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.