Programación estructurada

Ricardo Pérez López

IES Doñana, curso 2019/2020

Índice general

1. Aspectos teóricos de la programación estructurada

1.1. Programación estructurada

La programación estructurada es una técnica de programación cuyo objetivo es, esencialmente, la obtención de programas fiables y fácilmente mantenibles.

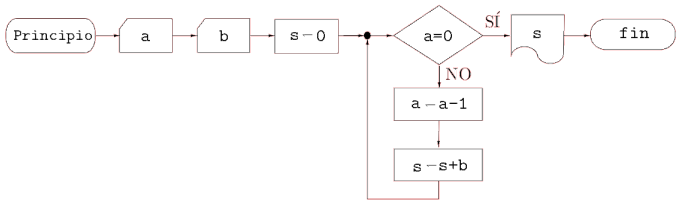
Su estudio puede dividirse en dos partes bien diferenciadas:

- Por una parte, el estudio conceptual se centra en ver qué se entiende por programa estructurado para estudiar con detalle sus características fundamentales.
- Por otra parte, dentro del enfoque práctico se presentará la metodología de refinamientos sucesivos que permite construir programas estructurados paso a paso, detallando cada vez más sus acciones componentes.

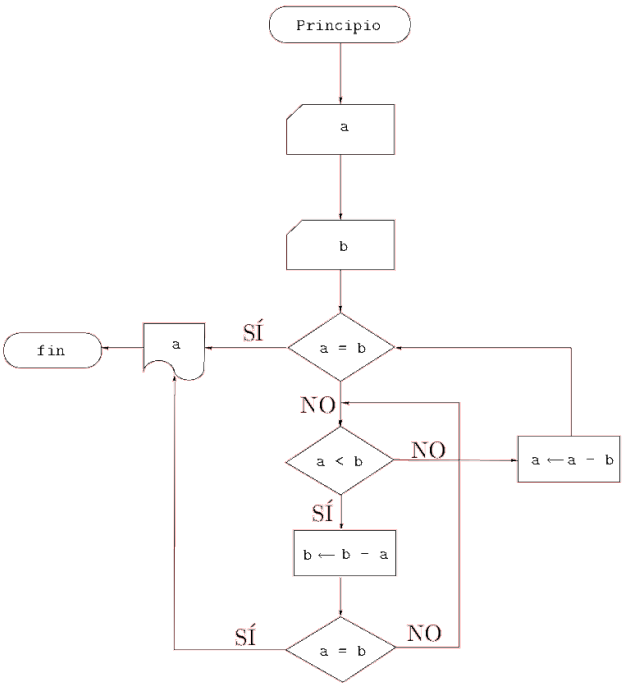
Las ideas que dieron lugar a la programación estructurada ya fueron expuestas por E. W. Dijkstra en 1965, aunque el fundamento teórico está basado en los trabajos de Böhm y Jacopini publicados en 1966.

La programación estructurada surge como respuesta a los problemas que aparecen cuando se programa sin una disciplina y unos límites que marquen la creación de programas claros y correctos.

Un programador disciplinado crearía programas fáciles de leer. Por ejemplo, el siguiente programa que calcula el producto de dos números:



En cambio, un programador indisciplinado crearía programas más difíciles de leer:



Si un programa se escribe de cualquier manera, aun siendo correcto desde el punto de vista de su funcionamiento, puede resultar engorroso, críptico, ilegible y casi imposible de modificar.

Lo que hay que hacer, en primer lugar, es impedir que el programador pueda escribir programas de cualquier manera, y para ello hay que restringir sus opciones a la hora de construir programas de forma que el diagrama resultante sea fácil de leer, entender y mantener.

Ese diagrama, una vez terminado, debe estar construido combinando sólo unos pocos tipos de bloques y cumpliendo una serie de restricciones.

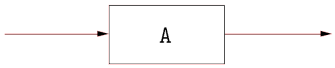
1.2. Programa restringido

Un **programa restringido** es aquel que se construye combinando únicamente los tres siguientes bloques:

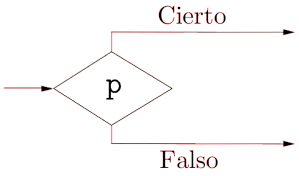
Acción, que sirve para representar una instrucción (por ejemplo: de lectura, escritura, asignación...).

Condición, que sirve para bifurcar el flujo del programa dependiendo del valor (verdadero o falso) de una expresión lógica.

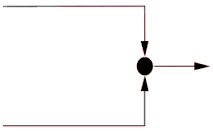
Agrupamiento, que sirve, como su nombre indica, para agrupar líneas de flujo con distintas procedencias.



Acción



Condición



Agrupamiento

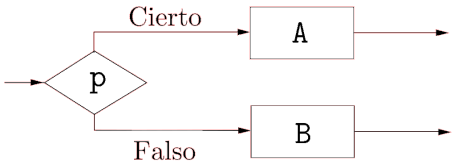
1.3. Programa propio

Se dice que un programa restringido es un **programa propio** (o ***limpio***) si reúne las tres condiciones siguientes:

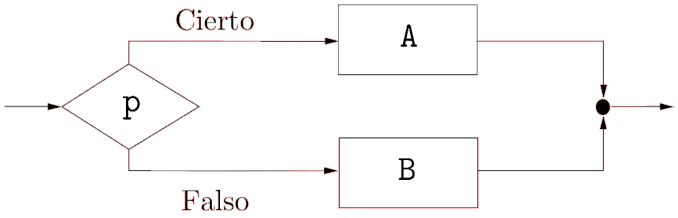
1. Todo bloque posee un único punto de entrada y otro único punto de salida.
2. Para cualquier bloque, existe al menos un camino desde la entrada hasta él y otro camino desde él hasta la salida.
3. No existen bucles infinitos.

Estas condiciones restringen el concepto de programa de modo que sólo se permite trabajar con aquéllos que están diseñados mediante el uso apropiado del agrupamiento y sin bloques superfluos o formando bucles sin salida.

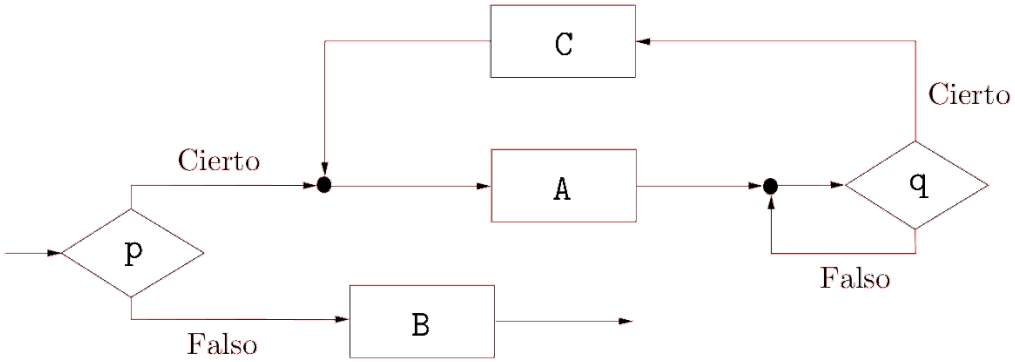
Este es un ejemplo de un programa que no es propio por no tener una única salida:



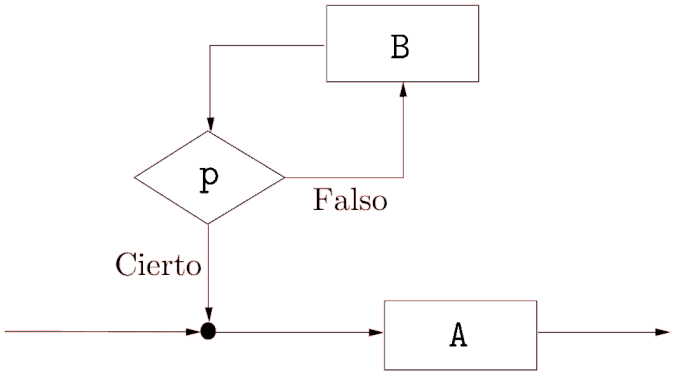
Agrupando las salidas se obtiene un programa propio:



Aquí se observa otro programa que no es propio, ya que existen bloques (los *A*, *C* y *q*) que no tienen un camino hasta la salida; si el programa llegara hasta esos bloques se bloquearía, pues no es posible terminar la ejecución:



Aquí aparece un programa que contiene bloques inaccesibles desde la entrada del diagrama:

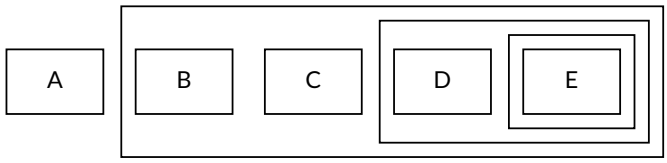


1.4. Estructura

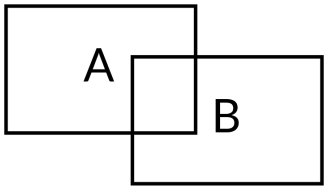
Una **estructura** es una construcción sintáctica (o un bloque constructivo) que se puede **anidar completamente** dentro de otra.

Eso significa que, dadas dos estructuras cualesquiera, o una está incluida completamente dentro de la otra, o son totalmente independientes.

Por tanto, los bordes de dos estructuras nunca pueden cruzarse:



Estructuras



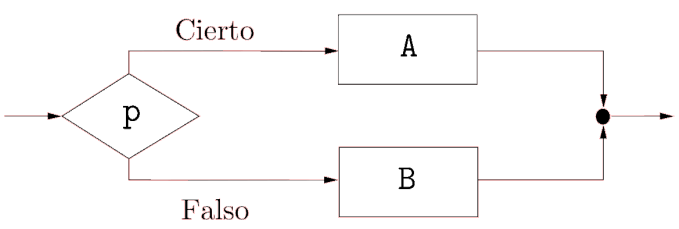
Estas no son estructuras

1.5. Programa estructurado

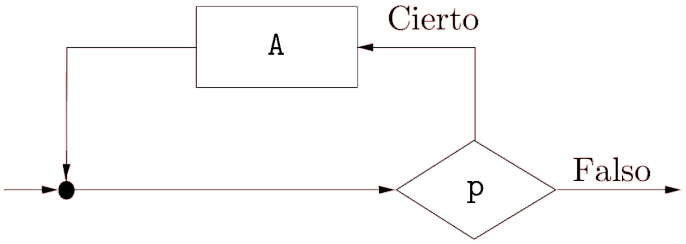
Un **programa estructurado** es un programa construido combinando las siguientes estructuras de control:

- La **secuencia** de dos acciones *A* y *B*, ya sean simples o compuestas.
- La **selección** entre dos acciones *A* y *B* dependiendo de un predicado *p*.
- La **iteración**, que repite una acción *A* dependiendo del valor de verdad de un predicado de control *p*.

Secuencia



Selección



Iteración

En pseudocódigo:

- Secuencia:

```
A
B
```

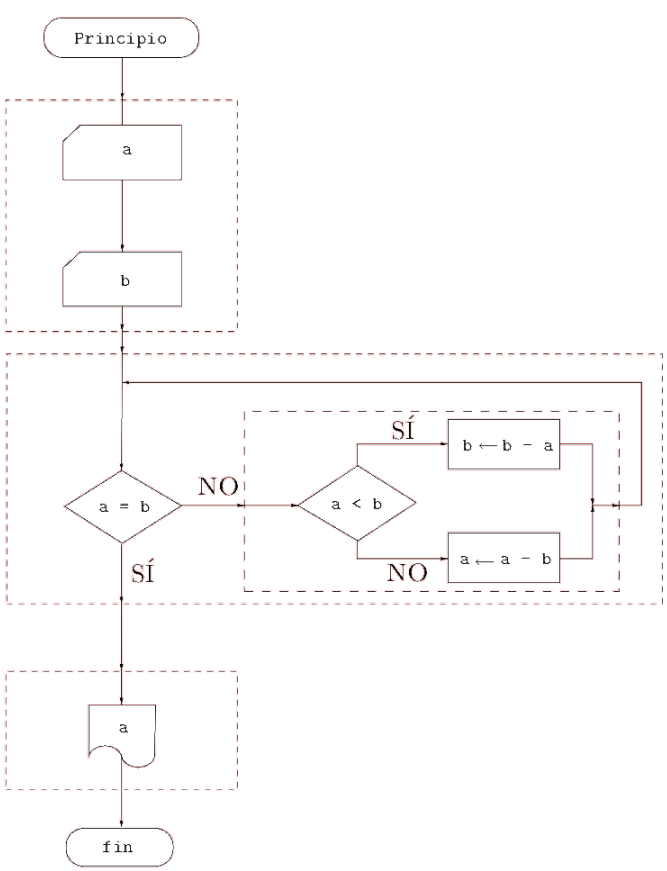
- Selección:

```
si p entonces
    A
sino
    B
```

- Iteración:

```
mientras p hacer
    A
```

Un programa estructurado equivalente al del ejemplo anterior, pero mucho más claro, sería:



inicio
leer a
leer b
mientras $a \neq b$ **hacer**
    si $a < b$ **entonces**
        $b \leftarrow b - a$
    sino
        $a \leftarrow a - b$
escribir a
fin

1.5.1. Ventajas de los programas estructurados

Las principales **ventajas de los programas estructurados** frente a los no estructurados son:

- Son más fáciles de entender, ya que básicamente se pueden leer de arriba abajo de estructura en estructura como cualquier otro texto sin tener que estar continuamente saltando de un punto a otro del programa.
- Es más fácil demostrar que son correctos, ya que las estructuras anidadas pueden verse como cajas negras, lo que facilita trabajar a diferentes niveles de abstracción.

- Se reducen los costes de mantenimiento.
- Aumenta la productividad del programador.
- Los programas quedan mejor documentados internamente.

1.6. Teorema de Böhm-Jacopini

El **teorema de Böhm-Jacopini**, también llamado **teorema de la estructura**, garantiza que todo programa propio se puede estructurar.

Se enuncia formalmente así:

> **Teorema de la estructura:**
>
> Todo programa propio es equivalente a un programa estructurado.

Por tanto, los programas estructurados son suficientemente expresivos como para expresar cualquier programa razonable.

Y además, por su naturaleza estructurada resultan programas más sencillos y claros.

En consecuencia, no hay excusa para no estructurar nuestros programas.

2. Estructuras básicas de control

2.1. Secuencia

La estructura secuencial en Python consiste sencillamente en poner cada sentencia una tras otra al mismo nivel de indentación.

No requiere de ninguna otra sintaxis particular ni palabras clave.

Una secuencia de sentencias actúa sintácticamente como si fuera una sola sentencia; por lo tanto, en cualquier lugar del programa donde se pueda poner una sentencia, se puede poner asimismo una secuencia de sentencias (que actuarían como una sola formando un bloque).

En las demás estructuras y definiciones que veremos a continuación, siempre que se espere una sentencia, se podrá poner una secuencia de sentencias.

> **Concepto fundamental:**
>
> En Python, la **estructura** del programa viene definida por la **indentación** del código.

Ejemplo:

```
x = 1
y = 2
f = lambda a, b: a + b
z = f(x + y)
```

Estas cuatro sentencias, al estar todas al mismo nivel de indentación, actúan como una sola sentencia en bloque y se ejecutan en orden de arriba abajo.

2.2. Selección

La selección (o estructura alternativa) tiene varias sintaxis en Python:

Selección simple:

```
if <condición>:
    <sentencia>
```

Selección doble:

```
if <condición>:
    <sentencia>
else:
    <sentencia>
```

Selección múltiple:

```
if <condición>:
    <sentencia>
[elif <condición>:
    <sentencia>]
[else:
    <sentencia>]
```

Ejemplos:

```
if 4 == 3:
    print('Son distintos')
    x = 5

if 4 == 3:
    print('Son distintos')
    x = 5
else:
    print('Son iguales')
    x = 9
```

```
if x < 2:
    print('Es menor que dos')
elif x <= 9:
    print('Está comprendido entre 2 y 9')
    x = 5
elif x < 12:
    print('Es mayor que 9 y menor que 12')
else:
    print('Es mayor o igual que 12')
```

2.3. Iteración

La iteración (estructura *iterativa* o *repetitiva*) en Python tiene la siguiente sintaxis:

Bucle **`while`**:

```
while <condición>:
    <sentencia>
```

A esta estructura se la llama **bucle *while*** o, simplemente, **bucle**.

La *<sentencia>* es el **cuerpo** del bucle.

Cada ejecución del cuerpo del bucle se denomina **iteración**.

2.4. Otras sentencias de control

2.4.1. `break`

La sentencia `break` finaliza el bucle que la contiene.

El flujo de control del programa pasa a la sentencia inmediatamente posterior al cuerpo del bucle.

Si la sentencia `break` se encuentra dentro de un bucle anidado (un bucle dentro de otro bucle), finalizará el bucle más interno.

```
i = 0
s = "string"
while i < len(s):
    val = s[i]
    i += 1
    if val == "i":
        break
    print(val)

print("Fin")
```

produce:

```
s
t
r
Fin
```

2.4.2. `continue`

La sentencia `continue` se usa para saltarse el resto del código dentro de un bucle en la iteración actual.

El bucle no finaliza sino que continúa con la siguiente iteración.

```
i = 0
s = "string"
while i < len(s):
    val = s[i]
    i += 1
    if val == "i":
        continue
    print(val)

print("Fin")
```

produce:

```
s
t
r
n
g
Fin
```

2.4.3. Excepciones

Incluso aunque una sentencia o expresión sea sintácticamente correcta, puede provocar un error cuando se intente ejecutar o evaluar.

Los errores detectados durante la ejecución del programa se denominan **excepciones** y no son incondicionalmente fatales si se capturan y se gestionan adecuadamente.

En cambio, la mayoría de las excepciones no son gestionadas por el programa y provocan mensajes de error.

Por ejemplo:

```
>>> 10 * (1 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam * 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

La última línea del mensaje de error indica qué ha ocurrido.

Hay distintos tipos de excepciones y ese tipo se muestra como parte del mensaje: los tipos del ejemplo anterior son `ZeroDivisionError`, `NameError` y `TypeError`.

El resto de la línea proporciona detalles sobre el tipo de excepción y qué lo causó.

2.4.3.1. Gestión de excepciones

Es posible escribir programas que gestionen excepciones concretas.

La sintaxis es:

```
try:
    <sentencia>
except [<excepcion> [as <identificador>]]:
    <sentencia>
[else:
    <sentencia>]
[finally:
    <sentencia>]
```

donde:

```
<excepcion> ::= <nombre_excepcion>
              | (<nombre_excepcion>(, <nombre_excepcion>)*)
```

Por ejemplo, el siguiente programa pide al usuario que introduzca un número entero por la entrada y luego lo muestra a la salida multiplicado por tres. Si el usuario no introduce un número entero, muestra un mensaje de advertencia:

```
try:
    x = int(input("Introduzca un número entero: "))
    print(x * 2)
except ValueError:
    print("¡Vaya! No ha introducido un número entero.")
else:
    print("La cosa ha acabado bien.")
finally:
    print("Fin")
```

Si no ha habido errores en el cuerpo del `try`, se ejecuta la parte del `else`.

En cualquier caso, siempre se ejecuta la parte del `finally`.

3. Metodología de la programación estructurada

3.1. Diseño descendente por refinamiento sucesivo

El diseño descendente es la técnica que permite descomponer un problema complejo en problemas más sencillos, realizándose esta operación de forma sucesiva hasta llegar al mínimo nivel de abstracción en el cual se pueden codificar directamente las operaciones en un lenguaje de programación estructurado.

Con esta técnica, los programas se crean en distintos niveles de refinamiento, de forma que cada nuevo nivel define la solución de forma más concreta y subdivide las operaciones en otras menos abstractas.

Los programas se diseñan de lo general a lo particular por medio de sucesivos refinamientos o descomposiciones que nos van acercando a las instrucciones finales del programa.

El último nivel permite la codificación directa en un lenguaje de programación.

3.2. Recursos abstractos

Descomponer un programa en términos de recursos abstractos consiste en descomponer una determinada acción compleja en un número de acciones mas simples, capaces de ser ejecutadas por un ordenador, y que constituirán sus instrucciones.

Es el complemento perfecto para el diseño descendente y el que nos proporciona el método a seguir para obtener un nuevo nivel de refinamiento a partir del anterior.

Se basa en suponer que, en cada nivel de refinamiento, todos los elementos (instrucciones, expresiones, funciones, etc.) que aparecen en la solución están ya disponibles directamente en el lenguaje de programación, aunque no sea verdad.

Esos elementos o recursos se denominan abstractos porque los podemos usar directamente en un determinado nivel de refinamiento sin tener que saber cómo funcionan realmente por dentro, o incluso si existen realmente. Nosotros suponemos que sí existen y que hacen lo que tienen que hacer sin preocuparnos del cómo.

En el siguiente refinamiento, aquellos elementos que no estén implementados ya directamente en el lenguaje se refinarán, bajando el nivel de abstracción y acercándonos cada vez más a una solución que sí se pueda implementar en el lenguaje.

El refinamiento acaba cuando la solución se encuentra completamente definida usando los elementos del lenguaje de programación (ya no hay recursos abstractos).

3.3. Ejemplo

Supongamos que queremos escribir un programa que muestre una tabla de multiplicar de tamaño $n \times n$.

Una primera versión (burda) del programa podría ser:

```
inicio
  leer n
  construir la tabla de n × n
fin
```

donde el programa se plantea como una secuencia de dos acciones: preguntar el tamaño de la tabla deseada y construir la tabla propiamente dicha.

La instrucción **leer** *n* ya está suficientemente refinada (se puede traducir a un lenguaje de programación) pero la segunda no (por tanto, es un recurso abstracto).

La construcción de la tabla se puede realizar fácilmente escribiendo en una fila los múltiplos de 1, en la fila inferior los múltiplos de 2, y así sucesivamente hasta que lleguemos a los múltiplos de *n*.

Por tanto, el siguiente paso es refinar la instrucción abstracta **construir la tabla de** $n \times n$, creando un nuevo nivel de refinamiento:

```
inicio
  leer n
  # construir la tabla de n × n:
  i ← 1
```

```
  mientras i ≤ n:
      escribir la fila de i
      i ← i + 1
fin
```

donde ahora aparece la acción **escribir la fila de** *i*, que escribe cada una de las filas de la tabla, y que habrá que refinar porque no se puede traducir directamente al lenguaje de programación.

En este (último) nivel refinamos la acción que nos falta, quedando:

```
inicio
  leer n
  { construir la tabla de n × n: }
  i ← 1
  mientras i ≤ n:
      { escribir la fila de i: }
      j ← 1
      mientras j ≤ n:
          escribir i × j
          j ← j + 1
      escribir un salto de línea
      i ← i + 1
fin
```

Ese programa es directamente traducible a Python:

```
n = int(input('Introduce el número: '))
i = 1
while i <= n:
    j = 1
    while j <= n:
        print(i * j, end=' ')
        j += 1
    print()
    i += 1
```

O mejor aún:

```
try:
    n = int(input('Introduce el número: '))
    i = 1
    while i <= n:
        j = 1
        while j <= n):
            print(f'{i * j:>3}', end=' ')
            j += 1
        print()
        i += 1
except ValueError:
    print('Número incorrecto')
```

4. Funciones con nombre

4.1. Definición de funciones con nombre

En programación imperativa también podemos definir funciones.

Al igual que ocurre en programación funcional, una función en programación imperativa es una construcción sintáctica que acepta argumentos y produce un resultado.

Pero a diferencia de lo que ocurre en programación funcional, una función en programación imperativa es una **secuencia de sentencias**.

Las funciones en programación imperativa conforman los bloques básicos que nos permiten **descomponer un programa en partes** que se combinan entre sí.

Todavía podemos construir funciones mediante expresiones lambda, pero Python nos proporciona otro mecanismo para definir funciones en estilo imperativo: las **funciones con nombre**.

La sintaxis para definir una función con nombre es:

```
def <nombre>([<parámetros>]):
    <cuerpo>
```

donde:

```
<cuerpo> ::= <sentencia>
```

Por ejemplo:

```
def saluda(persona):
    print('Hola', persona)
    print('Encantado de saludarte')

def despide():
    print('Hasta luego, Lucas')
```

Notas importantes:

- Tiene que haber, al menos, *una* sentencia (o secuencia, claro).
- Las sentencias van **indentadas** (o *sangradas*) dentro de la definición de la función, como una estructura.
- El final de la función se deduce al encontrarse una sentencia con un **nivel de indentación superior** (en el caso de arriba, otro `def`).

4.2. Paso de argumentos

Existen distintos mecanismos de paso de argumentos, dependiendo del lenguaje de programación utilizado.

Los más conocidos son los llamados **paso de argumentos *por valor*** y **paso de argumentos *por referencia***.

En Python existe un único mecanismo de paso de argumentos llamado **paso de argumentos *por asignación*** o también, a veces, **paso de argumentos *por nombre***.

En la práctica resulta bastante sencillo.

Consiste en suponer que **el argumento *se asigna* al parámetro** correspondiente, con toda la semántica relacionada con los *alias* de variables, inmutabilidad, mutabilidad, etcétera.

Por ejemplo:

```
1 def saluda(persona):
2     print('Hola', persona)
3     print('Encantado de saludarte')
4
5 saluda('Manolo')  # Saluda a Manolo
6 x = 'Juan'
7 saluda(x)         # Saluda a Juan
```

En la línea 5 se asigna a `persona` el valor `Manolo` (como si se hiciera `persona = Manolo`).

En la línea 7 se asigna a `persona` el valor de `x`, como si se hiciera `persona = x`, lo que sabemos que crea un *alias* (que no afectaría ya que el valor pasado es una cadena y, por tanto, inmutable).

En caso de pasar un argumento mutable:

```
1 def cambia(l):
2     print(l)
3     l.append(99)
4
5 lista = [1, 2, 3]
6 cambia(lista)     # Imprime [1, 2, 3]
7 print(lista)      # Imprime [1, 2, 3, 99]
```

La función es capaz de **cambiar el estado de la lista que se ha pasado como argumento** ya que, al llamar a la función, el argumento `lista` se pasa a la función **asignándola** al parámetro `l`, haciendo que ambas variables sean *alias* una de la otra (se refieren al mismo objeto) y, por tanto, la función está modificando la misma variable que se ha pasado como argumento (`lista`).

4.3. La sentencia `return`

Para devolver el resultado de la función al código que la llamó, hay que usar una sentencia `return`.

Cuando el intérprete encuentra una sentencia `return` dentro de una función:

- se finaliza la ejecución de la función,
- se devuelve el control al punto del programa en el que se llamó a la función y
- la función devuelve como resultado el valor de retorno definido en la sentencia `return`.

Por ejemplo:

```
1 def suma(x, y):
2     return x + y
3
4 a = input('Introduce el primer número: ')
5 b = input('Introduce el segundo número: ')
6 resultado = suma(a, b)
7 print('El resultado es:', resultado)
```

La función se define en las líneas 1–2. El intérprete lee la definición de la función pero no ejecuta sus sentencias en ese momento (lo hará cuando se *llame* a la función).

En la línea 6 se llama a la función `suma` pasándole como argumentos los valores de `a` y `b`, asignándose a `x` e `y`, respectivamente.

Dentro de la función, se calcula la suma `x + y` y la sentencia `return` finaliza la ejecución de la función, devolviendo el control al punto en el que se la llamó (la línea 6) y haciendo que su valor de retorno sea el valor calculado en la suma anterior (el valor de la expresión que acompaña al `return`).

El valor de retorno de la función sustituye a la llamada a la función en la expresión en la que aparece dicha llamada, al igual que ocurre con las expresiones lambda.

Por tanto, una vez finalizada la ejecución de la función, la línea 6 se reescribe sustituyendo la llamada a la función por su valor.

Si, por ejemplo, suponemos que el usuario ha introducido los valores `5` y `7` en las variables `a` y `b`, respectivamente, tras finalizar la ejecución de la función tendríamos que la línea 6 quedaría:

```
resultado = 12
```

y la ejecución del programa continuaría por ahí.

También es posible usar la sentencia `return` sin devolver ningún valor.

En ese caso, su utilidad es la de finalizar la ejecución de la función en algún punto intermedio de su código.

Pero en Python todas las funciones devuelven algún valor.

Lo que ocurre en este caso es que la función devuelve el valor `None`:

```
def hola():
  print('Hola')
  return
  print('Adiós')  # aquí no llega

hola()
```

imprime:

```
Hola
```

```
def hola():
  print('Hola')
  return
  print('Adiós')

x = hola()  # devuelve None
print(x)
```

imprime:

```
Hola
None
```

4.4. Ámbito de variables

La función `suma` se podría haber escrito así:

```
def suma(x, y):
    res = x + y
    return res
```

y el efecto final habría sido el mismo.

La variable `res` que aparece en el cuerpo de la función es una **variable local** y sólo existe dentro de la función. Por tanto, esto sería incorrecto:

```
1 def suma(x, y):
2     res = x + y
3     return res
4
5 resultado = suma(4, 3)
6 print(res)  # da error
```

Da error porque la variable `res` no está definida en el ámbito actual.

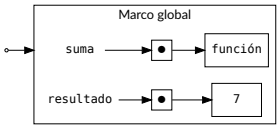
4.4.1. Variables locales

Al igual que pasa con las expresiones lambda, las definiciones de funciones generan un nuevo ámbito.

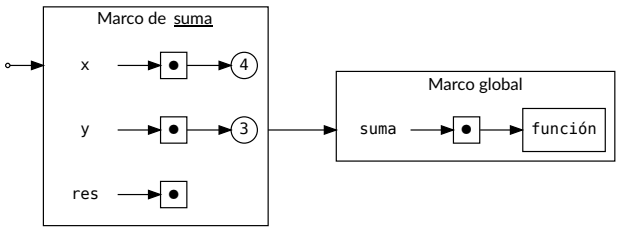
Tanto los parámetros como las variables que se definan en el cuerpo de la función son **locales** a ella, y por tanto sólo existen dentro de ella.

- El ámbito de un parámetro es el cuerpo de la función.
- El ámbito del resto de las variables locales es desde su definición hasta el final del cuerpo de la función.

Eso significa que se crea un nuevo marco en el entorno, que contendrá, en principio, los parámetros y las variables locales a la función.



Entorno en la línea 6



Entorno dentro de la función `suma`

4.4.2. Variables globales

Desde dentro de una función es posible usar variables globales.

Se puede **acceder** al valor de una variable global directamente:

```
x = 4

def prueba():
    print(x)

prueba()  # imprime 4
```

Pero para poder **modificar** una variable global ello es necesario que la función la declare previamente como *global*.

De no hacerlo así, el intérprete supondría que el programador quiere crear una variable local que tiene el mismo nombre que la global:

```
x = 4

def prueba():
    x = 5  # esta variable es local

prueba()
print(x)  # imprime 4
```

Como en Python no existen las declaraciones de variables, el intérprete tiene que *averiguar* por sí mismo qué ámbito tiene una variable.

Lo hace con una regla muy sencilla:

> Si hay una **asignación** a una variable **dentro** de una función, esa variable se considera **local**.

El siguiente código genera un error «*UnboundLocalError: local variable 'x' referenced before assignment*». ¿Por qué?

```
x = 4

def prueba():
```

```
    x = x + 4
    print(x)

prueba()
```

Como la función asigna un valor a `x`, Python considera que `x` es local.

Pero en la expresión `x + 4`, la variable `x` aún no tiene ningún valor asignado, por lo que genera un error «*variable local `x` referenciada antes de ser asignada*».

4.4.2.1. `global`

Para declarar una variable como global, se usa la sentencia `global`:

```
x = 4

def prueba():
    global x  # se declara que la variable x es global
    x = 5     # cambia el valor de la variable global x

prueba()
print(x)  # imprime 5
```

Las reglas básicas de uso de la sentencia `global` en Python son:

1. Cuando se crea una variable dentro de una función, por omisión es local.
2. Cuando se define una variable fuera de una función, por omisión es global (no hace falta usar la sentencia `global`).
3. Se usa la sentencia `global` para cambiar el valor de una variable global dentro de una función.
4. El uso de la sentencia `global` fuera de una función no tiene ningún efecto.
5. La sentencia `global` debe aparecer *antes* de que se use la variable global correspondiente.

4.4.2.2. Efectos laterales

Cambiar el estado de una variable global es uno de los ejemplos más claros y conocidos de los llamados **efectos laterales**.

Recordemos que una función tiene (o *provoca*) efectos laterales cuando provoca cambios de estado observables en el exterior de la función, más allá de devolver su valor de retorno. Típicamente:

- Cuando cambia el valor de una variable global
- Cuando cambia un argumento mutable
- Cuando realiza una operación de entrada/salida

Una función que provoca efectos laterales es una **función impura**, a diferencia de las **funciones puras**, que no tienen efectos laterales.

Una función también puede ser **impura** si su valor de retorno depende de algo más que de sus argumentos (p. ej., de una variable global).

Un ejemplo de **función impura** (con un efecto lateral provocado por una operación de entrada/salida) podría ser:

```
def suma(x, y):
    res = x + y
    print('La suma vale', res)
    return res

resultado = suma(4, 3) + suma(8, 5)
print(resultado)
```

Cualquiera que no sepa cómo está construida internamente la función `suma`, se podría pensar que lo único que hace es calcular la suma de dos números, pero resulta que también imprime un mensaje en la salida, por lo que el resultado final que se obtiene no es el que se esperaba:

```
La suma vale 7
La suma vale 13
20
```

Las llamadas a la función `suma` no se pueden sustituir por su valor de retorno correspondiente. Es decir, que no es lo mismo hacer:

```
resultado = suma(4, 3) + suma(8, 5)
```

que hacer:

```
resultado = 7 + 13
```

Por tanto, la función `suma` no cumple la **transparencia referencial**.

El que una función necesite **acceder al valor de una variable global** supone otra forma de **perder transparencia referencial**, ya que la convierte en **impura** porque su valor de retorno podría depender de algo más que de sus argumentos (en este caso, de la variable global).

En consecuencia, la función podría producir **resultados distintos en momentos diferentes** ante los mismos argumentos:

```
def suma(x, y):
    res = x + y + z  # impureza: depende del valor de una variable global
    return res

z = 5
print(suma(4, 3))  # imprime 12
z = 2
print(suma(4, 3))  # imprime 9
```

Igualmente, el **uso de la sentencia `global`** supone otra forma más de **perder transparencia referencial**, puesto que, gracias a ella, una función puede cambiar el valor de una variable global, lo que la convertiría en **impura** porque podría provocar un **efecto lateral** (la modificación de la variable global).

En consecuencia, la función podría producir **resultados distintos en momentos diferentes** ante los mismos argumentos:

```
def suma(x, y):
    global z
    res = x + y + z  # impureza: depende del valor de una variable global
    z = z + 1        # efecto lateral: cambia una variable global
    return res

z = 0
print(suma(4, 3))  # imprime 7
print(suma(4, 3))  # la misma llamada a función ahora imprime 8
```

4.5. Funciones locales a funciones

En Python es posible definir **funciones locales** a una función.

Las funciones locales también se denominan **funciones internas** o **funciones anidadas**.

Una función local se define **dentro** de otra función y, por tanto, sólo existe dentro de la función en la que se ha definido.

Su **ámbito** empieza en su definición y acaba al final del cuerpo de la función que la contiene.

Evita la superpoblación de funciones en el ámbito más externo cuando sólo tiene sentido su uso en un ámbito más interno.

Por ejemplo:

```
def fact(n):
    def fact_iter(n, acc):
        if n == 0:
            return acc
        else:
            return fact_iter(n - 1, acc * n)
    return fact_iter(n, 1)

print(fact(5))

# daría un error porque fact_iter no existe en el ámbito global:
print(fact_iter(5, 1))
```

La función `fact_iter` es local a la función `fact`. No se puede usar desde fuera de `fact`.

Tampoco se puede usar dentro de `fact` *antes* de haberse definido.

Lo siguiente daría un error porque intentamos usar `fact_iter` antes de haber definido:

```
def fact(n):
    print(fact_iter(1, 1))  # error: se usa antes de definirse
    def fact_iter(n, acc):  # aquí es donde empieza su definición
        if n == 0:
            return acc
        else:
            return fact_iter(n - 1, acc * n)
    return fact_iter(n, 1)
```

Las funciones locales definen un nuevo ámbito.

Ese nuevo ámbito crea un nuevo marco en el entorno.

Y ese nuevo marco se conecta con el marco del ámbito que lo contiene, es decir, el marco de la función que contiene a la local.

4.5.1. `nonlocal`

Una función local puede **acceder** al valor de las variables locales a la función que la contiene.

En cambio, cuando una función local quiere **modificar** el valor de una variable local a la función que la contiene, debe declararla previamente como **no local** con la sentencia `nonlocal`.

Es algo similar a lo que ocurre con las variables globales.

```
def fact(n):
    def fact_iter(acc):
        nonlocal n
        n = n - 1
        if n == 0:
            return acc
        else:
            return fact_iter(acc * n)
    return fact_iter(n)

print(fact(5))
```

La función local `fact_iter` puede acceder a la variable `n`, que es local a la función `fact` (para ello no es necesario declararla previamente como **no local**).

Como la variable `n` está declarada **no local** en `fact_iter`, también puede modificarla.

De esta forma, ya no es necesario pasar el valor de `n` como argumento a la función `fact_iter` y puede modificarla directamente.

4.6. *Docstrings*

Python implementa un mecanismo muy sencillo y elegante para documentar partes del código basado en cadenas llamadas ***docstrings*** (*cadenas de documentación*).

En funciones, únicamente tenemos que insertar una cadena en la primera línea del cuerpo:

```
def hola(arg):
  """
  Este es el docstring de la función.
  """
  print("Hola", arg, "!")

hola("Héctor")
```

Las reglas de estilo dictan que esa cadena debe escribirse con triples comillas.

Para consultar la documentación se usa la función `help` pasándole como argumento la función a consultar:

```
>>> help(hola)
Help on function hola in module __main__:

hola()
  Este es el docstring de la función.
```

También se puede documentar un script añadiendo un *docstring* al principio del mismo (en la primera línea):

```
"""Este es el docstring del script"""

def despedir():
    """Este es el docstring de la función despedir"""
    print("¡Adiós! Me despido desde la función despedir()")

def saludar():
    """Este es el docstring de la función saludar"""
    print("¡Hola! Te saludo desde la función saludar()")
```

Bibliografía

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.