Programación estructurada

Ricardo Pérez López

IES Doñana, curso 2023/2024

Generado el 2024/01/13 a las 15:54:00

Índice

1. Aspectos teóricos de la programación estructurada 2
1.1. Programación estructurada . . . 2
1.2. Programa restringido . . . 4
1.3. Programa propio . . . 5
1.4. Estructura . . . 6
1.5. Programa estructurado . . . 7
1.5.1. Ventajas de los programas estructurados . . . 10
1.6. Teorema de Böhm-Jacopini . . . 10

2. Estructuras básicas de control en Python 10
2.1. Secuencia . . . 10
2.2. Selección . . . 11
2.3. Iteración . . . 13
2.4. Otras sentencias de control . . . 16
2.4.1. `break` . . . 16
2.4.2. `continue` . . . 16
2.4.3. Excepciones . . . 17
2.4.4. Gestores de contexto . . . 19

3. Metodología de la programación estructurada 20
3.1. Diseño descendente por refinamiento sucesivo . . . 20
3.2. Recursos abstractos . . . 21
3.3. Ejemplo . . . 21

4. Programación procedimental 23
4.1. Procedimientos . . . 23
4.2. El paradigma de programación procedimental . . . 24
4.3. Procedimientos y refinamiento sucesivo . . . 24
4.4. Funciones imperativas . . . 27
4.4.1. Definición de funciones imperativas . . . 27
4.5. Llamadas a funciones imperativas . . . 29
4.6. Paso de argumentos . . . 31

4.7. La sentencia `return` . . . 32
4.8. Ámbito de variables . . . 34
4.8.1. Variables locales . . . 35
4.8.2. Variables globales . . . 36
4.9. Funciones locales a funciones . . . 41
4.9.1. `nonlocal` . . . 42

1. Aspectos teóricos de la programación estructurada

1.1. Programación estructurada

La **programación estructurada** es un paradigma de programación **imperativa** que se apoya en tres pilares fundamentales:

- **Estructuras básicas:** los programas se escriben usando sólo unos pocos componentes constructivos básicos que se combinan entre sí mediante composición.
- **Recursos abstractos:** los programas se escriben sin tener en cuenta inicialmente el ordenador que lo va a ejecutar ni las instrucciones de las que dispone el lenguaje de programación que se va a utilizar.
- **Diseño descendente por refinamiento sucesivo:** los programas se escriben de arriba abajo a través de una serie de niveles de abstracción de menor a mayor complejidad, pudiéndose verificar la corrección del programa en cada nivel.

Su **objetivo** es **conseguir programas fiables y fácilmente mantenibles**.

Su estudio puede dividirse en dos partes bien diferenciadas:

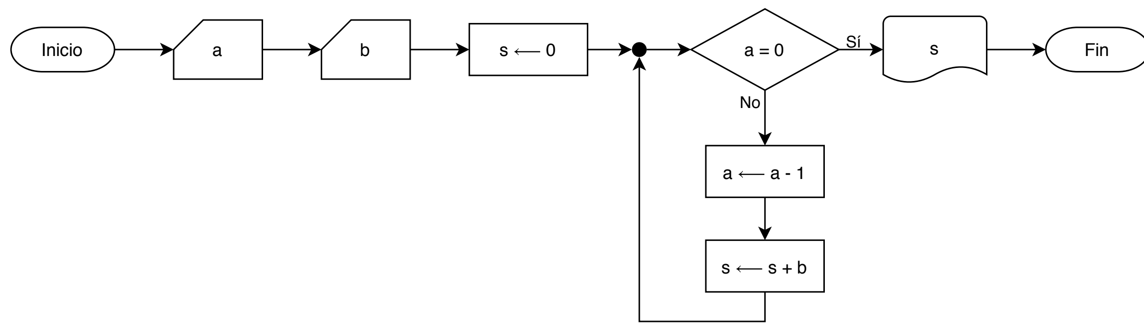
- Por una parte, el **estudio conceptual** se centra en ver qué se entiende por «programa estructurado» para estudiar con detalle sus características fundamentales.
- Por otra parte, dentro del **enfoque práctico** se presentará una metodología que permite construir programas estructurados paso a paso, detallando cada vez más las instrucciones que lo componen.

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.

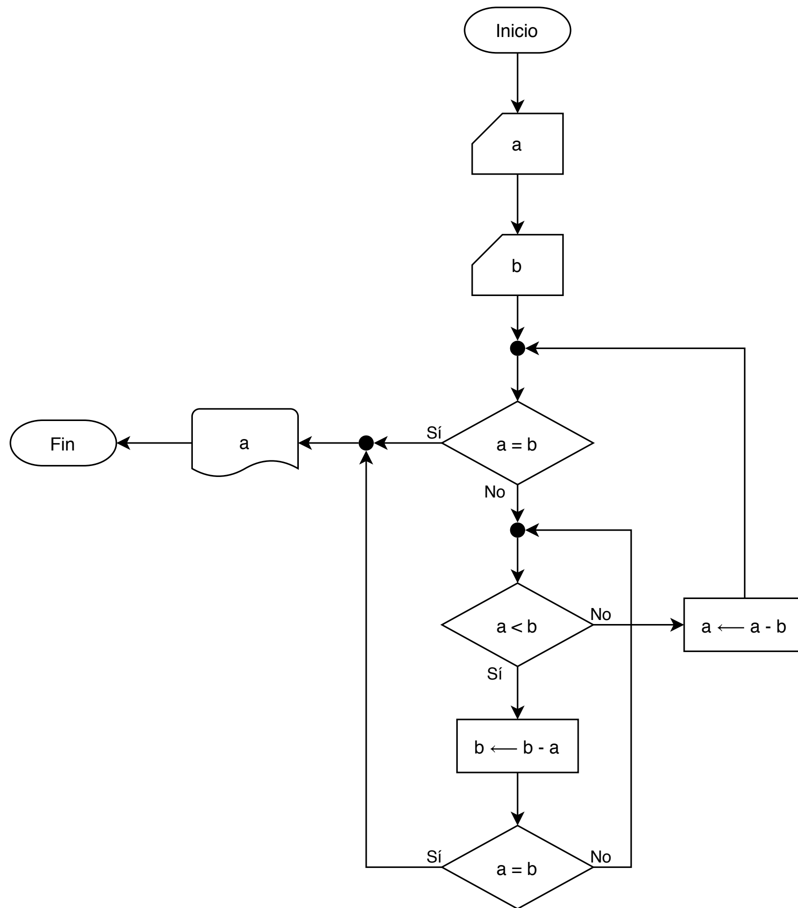
Un programador *disciplinado* crearía programas fáciles de leer en los que resulta relativamente fácil demostrar su corrección.

Por ejemplo, el siguiente programa que calcula el producto de dos números:



En cambio, un programador *indisciplinado* crearía programas más difíciles de leer y, por tanto, de demostrar que son correctos.

Este programa resuelve el mismo problema que el anterior, pero mediante saltos contínuos y líneas que se cruzan, lo que resulta en un programa más complicado de seguir.



Esos dos programas son **equivalentes**, lo que significa que producen el mismo resultado y los mismos efectos ante los mismos datos de entrada, aunque lo hacen de distinta forma.

Pero el primer programa es **mucho más fácil de leer y modificar** que el segundo, aunque los dos

resuelvan el mismo problema.

Si un programa se escribe de cualquier manera, aunque funcione correctamente, puede resultar engorroso, críptico, ilegible, casi imposible de modificar y de comprobar su corrección.

Por tanto, lo que hay que hacer 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 programa resultante sea fácil de leer, entender, mantener y verificar.

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

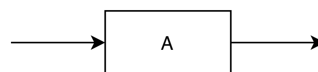
1.2. Programa restringido

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

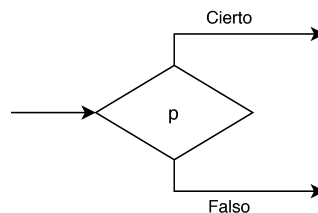
Sentencia, que sirve para representar una instrucción (por ejemplo: de lectura, escritura, asignación…).

Condición, que sirve para bifurcar el flujo del programa hacia un camino u otro dependiendo del valor de una expresión lógica.

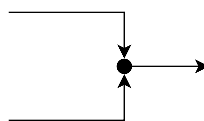
Agrupamiento, que sirve para agrupar líneas de flujo que procedan de distintos caminos.



Sentencia



Condición



Agrupamiento

1.3. Programa propio

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

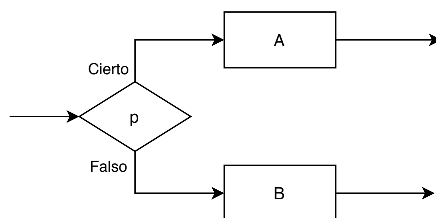
1. Posee un único punto de entrada y un único punto de salida.
2. Para cualquiera de sus componentes, existe al menos un camino desde la entrada hasta él y otro camino desde él hasta la salida.
3. No existen bucles infinitos.

Esto permite que **un programa propio pueda formar parte de otro programa mayor**, apareciendo allí donde pueda haber una sentencia.

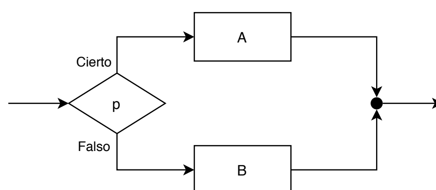
Cuando varios programas propios se combinan para formar uno solo, el resultado es también un programa propio.

Estas condiciones **restringen aún más el concepto de *programa***, de modo que sólo serán válidos aquellos que estén diseñados mediante el uso apropiado del agrupamiento (con una sola entrada y una sola salida) y sin componentes superfluos o formando bucles sin salida.

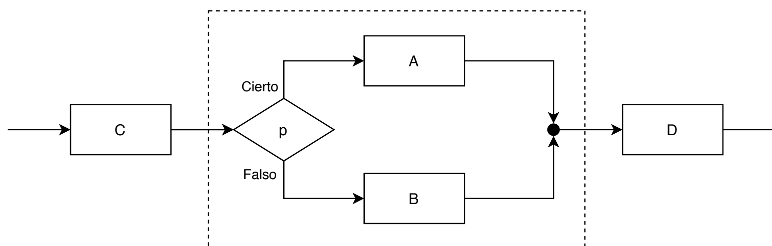
Este es un ejemplo de un programa que no es propio porque **no tiene una única salida**:



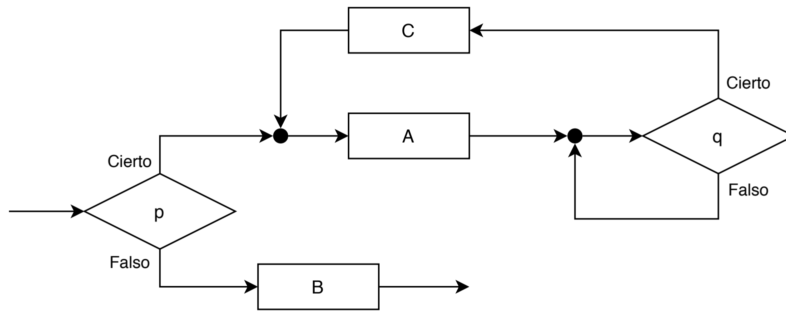
Agrupando las salidas se obtiene un programa propio:



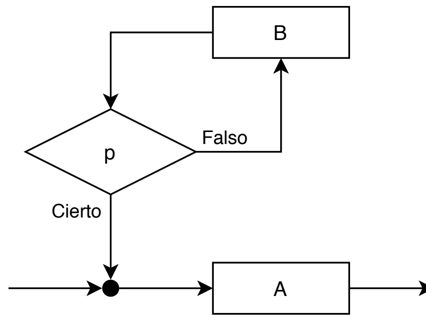
Ese programa propio ahora puede formar parte de otro programa mayor, ya que, al tener una sola entrada y una sola salida, puede actuar como una sentencia y aparecer allí donde pueda haber una sentencia:



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



Aquí aparece un programa que tampoco es propio porque contiene **componentes inaccesibles** desde la entrada del diagrama:

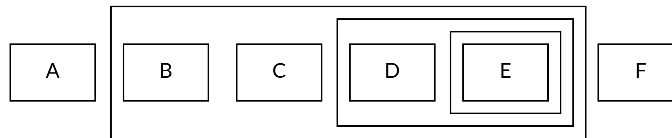


1.4. Estructura

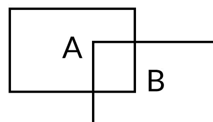
Las **estructuras** son construcciones sintácticas que pueden **anidarse completamente** unas dentro de otras.

Eso significa que, dadas dos estructuras cualesquiera, o una está incluida completamente dentro de la otra, o no se tocan en absoluto.

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



Estructuras



Estas no son estructuras

Las estructuras forman los componentes constructivos básicos de cualquier programa estructurado.

Por tanto, un programa estructurado se crea combinando entre sí varias estructuras.

Sintácticamente, una estructura es una unidad compuesta por varias sentencias que actúan como una sola.

Toda estructura, por tanto, representa una **sentencia compuesta** que actúa como un miniprograma propio y, por tanto, con un único punto de entrada y un único punto de salida.

1.5. Programa estructurado

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

1. La **estructura secuencial**, ***secuencia*** o **bloque** de una, dos o más sentencias *A*, *B*, *C*, etcétera.

   Los lenguajes que permiten la creación de bloques, incluyendo bloques dentro de otros bloques, se denominan **lenguajes estructurados en bloques**.

2. La **estructura alternativa** o ***selección*** entre dos sentencias *A* y *B* dependiendo de un predicado *p*.
3. La **estructura repetitiva** o ***iteración***, que repite una sentencia *A* dependiendo del valor lógico de un predicado *p*.

Las estructuras de control son **sentencias compuestas** que contienen, a su vez, otras sentencias.

Secuencia

Selección

Iteración

En pseudocódigo:

- Secuencia:

```
A
B
```

- Selección:

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

- Iteración:

```
mientras p hacer
    A
```

Cada una de las sentencias que aparecen en una estructura (las indicadas anteriormente como *A* y *B*) pueden ser, a su vez, estructuras.

- Esto es así porque una estructura también es una sentencia (que en este caso sería una sentencia *compuesta* en lugar de una sentencia *simple*).
- Por tanto, una estructura puede aparecer en cualquier lugar donde se espere una sentencia.

Resumiendo, en un programa podemos tener **dos tipos de sentencias**:

- **Sentencias *simples***.
- **Estructuras de control**, que son sentencias ***compuestas*** formadas a su vez por otras sentencias (que podrán ser, a su vez, simples o compuestas, recursivamente).

Por consiguiente, **todo programa puede verse como una única sentencia**, simple o compuesta por otras.

Esto tiene una consecuencia más profunda: si un programa es una sentencia, también puede decirse que cada sentencia es como un programa en sí mismo.

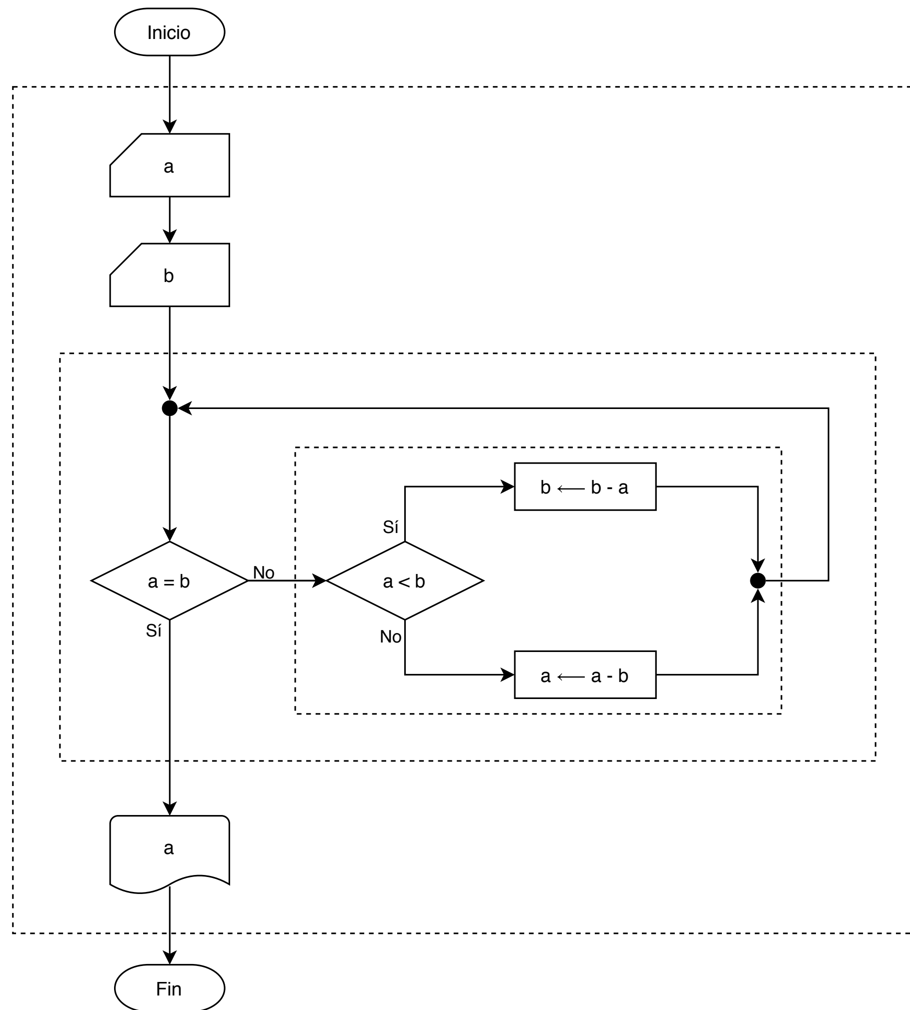
Como las estructuras de control también son sentencias, **cada estructura de control es como un *miniprograma* dentro del programa**.

Ese miniprograma debe cumplir las propiedades de los programas propios (los programas que no son propios no nos interesan).

Por eso, las estructuras:

1. Siempre tienen un único punto de entrada y un único punto de salida.
2. Tienen un camino desde la entrada a cada sentencia de la estructura, y un camino desde cada una de ellas hasta la salida.
3. No debe tener bucles infinitos.

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



Los cuadrados de trazo discontinuo representan las estructuras que forman el programa.

leer a
leer b
mientras $a \neq b$ **hacer**
  si $a < b$ **entonces**
    $b \longleftarrow b - a$
  sino
    $a \longleftarrow a - b$
escribir a

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, claros y fáciles de entender, mantener y verificar.

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

2. Estructuras básicas de control en Python

2.1. Secuencia

La **secuencia** (o *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 una secuencia de sentencias (que actuarían como una sola formando un **bloque**).

Esto es así porque, como vimos, toda sentencia puede ser simple o compuesta (una estructura) y, por tanto, **toda estructura es también una sentencia** (actúa como si fuera una única sentencia pero compuesta por otras de forma recursiva).

Por tanto, en cualquier lugar donde se pueda poner una sentencia, se puede poner una estructura.

La sintaxis es, sencillamente:

```
⟨secuencia⟩ ::=
⟨sentencia⟩
⟨sentencia⟩*
```

Las sentencias deben empezar todas en el mismo nivel de indentación (misma posición horizontal o *columna*).

Puede haber líneas en blanco entre las sentencias del mismo bloque.

> **Concepto fundamental:**
>
> En Python, la **estructura** del programa viene definida por la **indentación** del código.
>
> Por tanto, las instrucciones que aparecen consecutivamente una tras otra en el mismo nivel de indentación (es decir, las que empiezan en la misma columna en el archivo fuente) pertenecen a la misma estructura.

Ejemplo:

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

Estas cuatro sentencias, al estar todas consecutivas en el mismo nivel de indentación, actúan como una sola sentencia en bloque (forman una estructura *secuencial*) y se ejecutan en orden de arriba abajo.

A partir de ahora, tenemos que una sentencia puede ser simple o compuesta (es decir, una estructura), y esa sentencia compuesta puede ser una secuencia:

```
⟨sentencia⟩ ::= ⟨sentencia_simple⟩ | ⟨estructura⟩
⟨estructura⟩ ::= ⟨secuencia⟩
⟨secuencia⟩ ::=
⟨sentencia⟩
⟨sentencia⟩*
```

2.2. Selección

La **selección** (o *estructura alternativa*) en Python tiene la siguiente sintaxis:

```
⟨selección⟩ ::=
if ⟨condición⟩:
    ⟨sentencia⟩
[elif ⟨condición⟩:
    ⟨sentencia⟩]*
[else:
    ⟨sentencia⟩]
```

© Ricardo Pérez López (IES Doñana, curso 2023/2024)

También se la llama «**sentencia `if`**».

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')
```

La estructura alternativa está formada por una sucesión de cláusulas que asocian una condición con una sentencia.

Las cláusulas van marcadas con `if`, `elif` o `else`.

Las condiciones se van comprobando de arriba abajo, en el orden en que aparecen, de forma que primero se comprueba la condición del `if` y después las diferentes `elif`, si las hay.

En el momento en que se cumple una de las condiciones, se ejecuta su sentencia correspondiente y se sale de la estructura alternativa (no se sigue comprobando más).

Si no se cumple ninguna condición y hay una cláusula `else`, se ejecutará la sentencia de ésta.

Si no se cumple ninguna condición y no hay cláusula `else`, no se ejecuta ninguna sentencia.

Puede haber cláusulas `elif` y no haber `else`.

En el siguiente código:

```
1 a = 4
2 b = 3
3 if a != b:
4     print('Son distintos')
5     x = 5
6 else:
7     print('Son iguales')
```

tenemos las siguientes estructuras, anidadas una dentro de la otra:

1. Una *secuencia* formada por un bloque de tres sentencias: las asignaciones `a = 4` y `b = 3` y la sentencia `if ... else` que va desde la línea 3 hasta la 7.

2. La *selección* **`if ... else`**.
3. Una *secuencia* formada por las sentencias de las líneas 4–5.

Recordemos: cada estructura es una sentencia en sí misma, y contiene a otras sentencias (que pueden ser simples u otras estructuras).

Aquí se ven representadas visualmente las estructuras que forman el código fuente del programa:

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

Representación de las distintas estructuras que forman el código

Se aprecia claramente que hay tres estructuras (dos secuenciales y una alternativa) y cinco sentencias simples (las asignaciones y los `print`), por lo que hay ocho sentencias en total.

Ahora nuestra gramática se amplía:

```
⟨sentencia⟩ ::= ⟨sentencia_simple⟩ | ⟨estructura⟩
⟨estructura⟩ ::= ⟨secuencia⟩ | ⟨selección⟩
⟨selección⟩ ::=
if ⟨condición⟩:
    ⟨sentencia⟩
[elif ⟨condición⟩:
    ⟨sentencia⟩]*
[else:
    ⟨sentencia⟩]
```

2.3. Iteración

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

```
⟨iteración⟩ ::=
while ⟨condición⟩:
    ⟨sentencia⟩
```

A esta estructura también se la llama «**sentencia `while`**», «**bucle `while`**» o, simplemente, «**bucle**».

También se dice que la *⟨sentencia⟩* es el «**cuerpo**» del bucle.

© Ricardo Pérez López (IES Doñana, curso 2023/2024)

La estructura repetitiva asocia una condición a una sentencia, de forma que lo primero que se hace nada más entrar en la estructura es comprobar la condición:

- **Si la condición no se cumple**, se salta la sentencia y se sale de la estructura, pasando a la siguiente sentencia que haya tras el bucle.
- **Si la condición se cumple**, se ejecuta la sentencia y se vuelve otra vez al principio de la estructura, donde se volverá a comprobar si la condición se cumple.

  Este ciclo ciclo de «*comprobación y ejecución*» se repite una y otra vez hasta que se compruebe que la condición ya no se cumple, momento en el que se saldrá del bucle.

Cada vez que se ejecuta el cuerpo del bucle decimos que se ha producido una **iteración** o **paso** del bucle.

Es importante observar que la comprobación de la condición se hace justo al principio de cada iteración, es decir, justo antes de ejecutar la sentencia en la iteración actual.

Ejemplos

El siguiente código:

```python
x = 0
while x < 5:
    print(x)
    x += 1
print('Fin')
```

genera la siguiente salida:

```
0
1
2
3
4
Fin
```

```python
x = 0
while x < 5:
    print(x)
    x += 1
print('Fin')
```

Estructuras en el código

```
salida = False
while not salida:
    x = input('Introduce un número: ')
    if x == '2':
        salida = True
    print(x)
```

```
salida = False
while not salida:
    x = input('Introduce un número: ')
    if x == '2':
        salida = True
    print(x)
```

Estructuras en el código

Si la condición se cumple siempre y nunca evalúa a `False`, la ejecución nunca saldrá del bucle y tendremos lo que se denomina un **bucle infinito**.

Generalmente, los bucles infinitos indican un fallo en el programa y, por tanto, hay que evitarlos en la medida de lo posible.

Sólo en casos muy particulares resulta lógico y conveniente tener un bucle infinito.

Por ejemplo, los siguientes bucles serían infinitos:

```
while True:
    print("Hola")
```

```
i = 0
while i >= 0:
    print("Hola")
    i += 1
```

Ahora ampliamos de nuevo nuestra gramática, esta vez con la estructura de iteración (o sentencia **`while`**):

```
⟨sentencia⟩ ::= ⟨sentencia_simple⟩ | ⟨estructura⟩
⟨estructura⟩ ::= ⟨secuencia⟩ | ⟨selección⟩ | ⟨iteración⟩
⟨iteración⟩ ::=
while ⟨condición⟩:
    ⟨sentencia⟩
```

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.

```
j = 0
s = "string"
while j < len(s):
    val = s[j]
    j += 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.

```
j = 0
s = "string"
while j < len(s):
    val = s[j]
    j += 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 tienen por qué ser incondicionalmente fatales si se capturan y se gestionan adecuadamente.

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

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.

Para ello se utiliza una estructura de control llamada **`try ... except`**.

La sintaxis es:

```
⟨gestión_excepciones⟩ ::=
try:
    ⟨sentencia⟩
(except [⟨excepcion⟩ [as ⟨identificador⟩]]:
    ⟨sentencia⟩)+
[else:
    ⟨sentencia⟩]
[finally:
    ⟨sentencia⟩]
```

donde:

⟨excepcion⟩ ::= *⟨nombre_excepcion⟩*
    | **(***⟨nombre_excepcion⟩*(**,** *⟨nombre_excepcion⟩*)* **)**

Su funcionamiento es el siguiente:

- Se intenta ejecutar el bloque de sentencias del **`try`**.
- Si durante su ejecución no se levanta ninguna excepción, se saltan los **`except`** y se ejecutan las sentencias del **`else`**.
- Si se levanta alguna excepción, se busca (por orden de arriba abajo) algún **`except`** que cuadre con el tipo de excepción que se ha lanzado y, si se encuentra, se ejecutan sus sentencias asociadas.
- Finalmente, y en cualquier caso (se haya levantado alguna excepción o no), se ejecutan las sentencias del **`finally`**.

Por ejemplo, el siguiente programa pide al usuario que introduzca un número entero por la entrada. Si el dato introducido es correcto (es un número entero), lo muestra a la salida multiplicado por tres y dice que la cosa acabó bien. Si no, muestra un mensaje de advertencia:

```
try:
    x = int(input("Introduzca un número entero: "))
    print(x * 3)
except ValueError:
    print("¡Vaya! No ha introducido un número entero.")
else:
    print("La cosa ha acabado bien.")
finally:
    print("Fin")
```

En cualquiera de los dos casos, siempre acaba diciendo `Fin`.

Y volvemos a ampliar de nuevo nuestra gramática:

⟨sentencia⟩ ::= *⟨sentencia_simple⟩* | *⟨estructura⟩*
⟨estructura⟩ ::= *⟨secuencia⟩*
    | *⟨selección⟩*
    | *⟨iteración⟩*
    | *⟨gestión_excepciones⟩*
⟨gestión_excepciones⟩ ::=
try:
    ⟨sentencia⟩
(**except** [*⟨excepcion⟩* [**as** *⟨identificador⟩*]]**:**
    ⟨sentencia⟩)+
[**else:**
    ⟨sentencia⟩]
[**finally:**
    ⟨sentencia⟩]

2.4.4. Gestores de contexto

A veces un programa necesita trabajar con recursos externos:

- Archivos locales.
- Conexiones a bases de datos.
- Conexiones de red.

Trabajar con esos recursos siempre implica los siguientes pasos:

1. Abrir el recurso (solicitar la apertura o la conexión al sistema operativo).
2. Usar el recurso.
3. Cerrar el recurso (solicitar su cierre o su desconexión al sistema operativo).

Por ejemplo, al trabajar con archivos hay que:

1. Abrir el archivo con `f = open(...)`.
2. Usar el archivo con `f.read(...)`, `f.write(...)`, etc.
3. Cerrar el archivo con `f.close()`.

La **cantidad de recursos abiertos** al mismo tiempo está **limitada** por el sistema operativo o el intérprete.

Por ejemplo, si intentamos abrir demasiados archivos a la vez, el intérprete nos devolverá el error: `OSError: [Errno 24] Too many open files`.

Además, cada recurso abierto consume, a su vez, recursos del sistema operativo o del intérprete (memoria, descriptores internos, etcétera).

Por ello, es importante **acordarse de cerrar el recurso** una vez hayamos terminado de trabajar con él, para que el sistema operativo o el intérprete pueda liberar los recursos que está consumiendo y éstos se puedan reutilizar.

Para ello, se puede usar un **`try ... finally`**:

```
f = open('hola.txt', 'w')
try:
    f.write('¡Hola, mundo!')
finally:
    f.close()
```

Esto garantiza que el archivo se cerrará aunque el `f.write(...)` levante una excepción.

Los gestores de contexto son un mecanismo más cómodo y elegante para trabajar con recursos y asegurarse de que se cierran al final.

Para ello, se usa la sentencia **`with ... as`**, cuya sintaxis es:

```
⟨gestor_contexto⟩ ::=
with ⟨expresión⟩ [as ⟨identificador⟩]:
    ⟨sentencia⟩
```

El siguiente código es equivalente al anterior:

```
with open('hola.txt', 'w') as f:
    f.write('¡Hola, mundo!')
```

La sentencia `with ... as` es una estructura de control que hace lo siguiente:

1. Evalúa la ⟨*expresión*⟩, que deberá devolver un **gestor de recursos**.

   Los gestores de recursos son objetos que responden a los métodos `__enter__` y `__exit__`.

2. Llama al método `__enter__` sobre el objeto, el cual debe abrir y devolver el recurso.

3. Ese recurso se asigna a la variable del `identificador`.

4. Ejecuta la `sentencia` que, por supuesto, puede ser simple o compuesta.

5. Cuando termina de ejecutar la sentencia, llama al método `__exit__` sobre el objeto inicial, el cual se encargará de cerrar el recurso.

Por tanto, al salir de la estructura de control `with ... as`, se garantiza que el recurso asignado a `f` está cerrado.

Eso significa que, en el siguiente código, la última llamada al método `write` fallará al no estar abierto el recurso:

```
>>> with open('hola.txt', 'w') as f:
...     f.write('¡Hola, mundo!')
...
13
>>> f.write('Esto fallará')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

3. Metodología de la programación estructurada

3.1. Diseño descendente por refinamiento sucesivo

El diseño descendente es la técnica que consiste en descomponer un problema complejo en problemas más sencillos, realizándose esta operación de forma sucesiva hasta llegar al máximo nivel de detalle 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 más detalladas.

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 sentencia compleja en sentencias 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).

Al diseñar un programa estructurado, **se deben estructurar al mismo tiempo tanto el *programa* como los *datos*** que éste manipula.

Por tanto, el diseño descendente por refinamiento sucesivo se debe ir aplicando también a los datos además de a las instrucciones.

En cada paso del refinamiento, tanto las instrucciones como los datos se deben considerar recursos abstractos, de forma que, en un determinado nivel de abstracción, las instrucciones y los datos deberían estar refiniados con el mismo nivel de detalle.

Hay que evitar, por tanto, que las instrucciones estén poco detalladas y los datos muy detallados, o viceversa.

3.3. Ejemplo

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

Por ejemplo, para $n = 10$ tendríamos:

$$\begin{matrix} 1 & 2 & 3 & \cdots & 10 \\ 2 & 4 & 6 & \cdots & 20 \\ 3 & 6 & 9 & \cdots & 30 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 20 & 30 & \cdots & 100 \end{matrix}$$

Una primera versión (muy burda y poco refinada) del algoritmo escrito en pseudocódigo, que sería el paso previo al programa escrito en un lenguaje de programación, podría ser:

> **Algoritmo:** Tabla de multiplicar de $n \times n$
> **Entrada:** El tamaño n (por la entrada estándar)
> **Salida:** La tabla de multiplicar de $n \times n$ (por la salida estándar)
> **inicio**
>   **leer** n
>   **construir la tabla de** $n \times n$
> **fin**

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

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

Podríamos traducir ya la sentencia «**leer** n» al lenguaje de programación, o podríamos esperar a tener todas las sentencias refinadas y traducirlas todas a la vez. En este caso, lo haremos de la segunda forma.

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 sentencia abstracta «**construir la tabla de** $n \times n$», creando un nuevo nivel de refinamiento:

> **Algoritmo:** Tabla de multiplicar de $n \times n$
> **Entrada:** El tamaño n (por la entrada estándar)
> **Salida:** La tabla de multiplicar de $n \times n$ (por la salida estándar)
> **inicio**
>   **leer** n
> $i \longleftarrow 1$
>   **mientras** $i \leq n$ **hacer**
>   **escribir la fila de** i
> $i \longleftarrow i + 1$
> **fin**

donde ahora aparece la sentencia «**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 nivel refinamos la sentencia que nos falta, quedando:

> **Algoritmo:** Tabla de multiplicar de $n \times n$
> **Entrada:** El tamaño n (por la entrada estándar)
> **Salida:** La tabla de multiplicar de $n \times n$ (por la salida estándar)
> **inicio**
>   **leer** n
> $i \longleftarrow 1$
>   **mientras** $i \leq n$ **hacer**
> $j \longleftarrow 1$
>   **mientras** $j \leq n$ **hacer**

escribir $i \times j$ sin salto de línea
$j \longleftarrow j + 1$
escribir un salto de línea
$i \longleftarrow i + 1$
fin

Este es el último nivel de refinamiento, porque todas las instrucciones ya se pueden traducir directamente a un lenguaje de programación.

Por ejemplo, en Python se escribiría así:

```
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. Programación procedimental

4.1. Procedimientos

A un bloque de sentencias que realiza una tarea específica se le puede dar un **nombre**.

De esta forma se crearía una única **unidad de código empaquetado que actuaría bajo ese nombre como una caja negra**, de manera que, para poder usarla, bastaría con ***llamarla* invocando su nombre** sin tener que conocer sus detalles internos de funcionamiento.

A este tipo de «*bloques con nombre*» se les denomina **subrutinas**, **subprogramas** o **procedimientos**.

Es el equivalente imperativo al concepto de *función* en programación funcional, solo que en lugar de estar formado por una expresión, está formado por una sentencia o bloque de sentencias.

Los procedimientos nos ayudan a:

- Descomponer el problema principal en subproblemas más pequeños que se pueden resolver por separado de una forma más o menos independiente del resto.
- Ocultar la complejidad de partes concretas de un programa bajo capas de abstracción con diferentes niveles de detalle.
- Desarrollar el programa mediante sucesivos refinamientos de cada nivel de abstracción.

En definitiva, los procedimientos son **abstracciones**.

La ***llamada*** o ***invocación*** a un procedimiento es una sentencia simple pero que provoca la ejecución de un bloque de sentencias.

Por tanto, podría considerarse que **un procedimiento es una sentencia compuesta que actúa como una sentencia simple**.

4.2. El paradigma de programación procedimental

La **programación procedimental** (*procedural programming*) es un paradigma de programación imperativa basada en los conceptos de **procedimiento** y **llamada a procedimientos**.

En este paradigma, un programa imperativo está compuesto principalmente por procedimientos (bloques de sentencias con nombre) que se llaman entre sí.

Los procedimientos pueden tener parámetros a través de los cuales reciben sus datos de entrada, caso de necesitarlos.

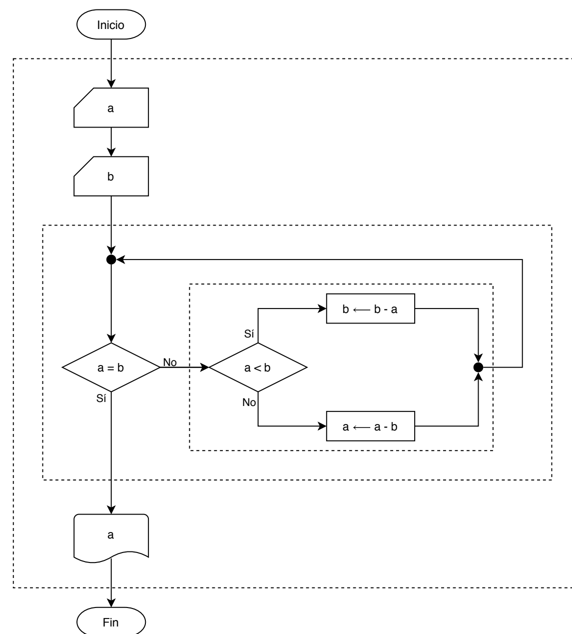
A su vez, los procedimientos pueden devolver un resultado, de ser necesario.

Los procedimientos, además, determinan su propio ámbito local y (dependiendo del lenguaje de programación usado) también podrían acceder a otros ámbitos no locales que contengan al suyo, como el ámbito global.

4.3. Procedimientos y refinamiento sucesivo

Durante el proceso de refinamiento sucesivo que acabamos de estudiar, se pueden ir creando procedimientos que representen **diferentes niveles de detalle** en el diseño descendente.

Recordemos que una estructura de control es una sentencia compuesta y, como tal, podemos estudiarla como si fuera una sola sentencia (con su entrada y su salida), sin tener que conocer el detalle de cómo funciona por dentro, es decir, sin tener que conocer qué sentencias más simples contiene.



Las cajas con trazo discontinuo, que representan los límites de cada estructura, nos hacen entender que podemos ver cada una de esas estructuras como una sola sentencia.

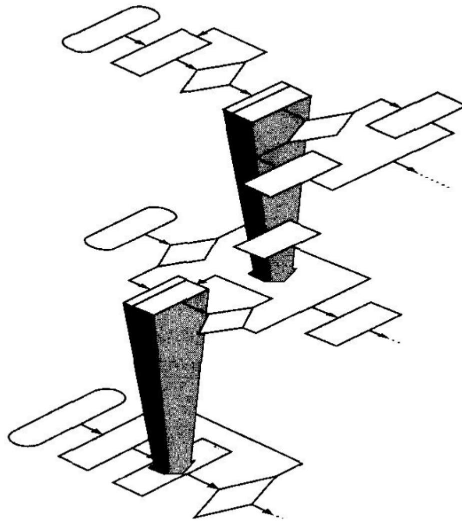
De igual forma, una llamada a un procedimiento es una sentencia simple que actúa como una sentencia compuesta, formada por varias instrucciones (el *cuerpo* del procedimiento) que actúan como una sola.

En ese caso, **el procedimiento actúa como un recurso abstracto en un determinado nivel** (en ese nivel, se invoca al procedimiento aunque aún no exista) que luego se implementa en un nivel de mayor refinamiento.

El uso de procedimientos para escribir programas siguiendo un diseño descendente nos lleva a un **código descompuesto en partes separadas** en lugar de tener un único código enorme con todo el texto del programa escrito directamente al mismo nivel.

Esta forma de refinamiento y de diseño descendente está ya más relacionado con el concepto de **programación modular**, que estudiaremos posteriormente.

En un ordinograma, una llamada a un procedimiento se representa como un rectángulo con doble trazo superior.



Diseño descendente por refinamiento sucesivo usando procedimientos

El ejemplo anterior descompuesto en procedimientos sería:

Algoritmo: Tabla de multiplicar de $n \times n$
Entrada: El tamaño n (por la entrada estándar)
Salida: La tabla de multiplicar de $n \times n$ (por la salida estándar)
inicio
  leer n
  construir_tabla(n)
fin

Procedimiento: construir_tabla(m)
Entrada: El tamaño m
Salida: La tabla de multiplicar de $m \times m$ (por la salida estándar)
inicio
  $i \longleftarrow 1$
  mientras $i \leq m$ **hacer**
    escribir_fila(i, m)
    $i \longleftarrow i + 1$
fin

Procedimiento: escribir_fila(f, t)
Entrada: El número (f) de la fila a escribir y el tamaño (t) de la tabla
Salida: La fila f de la tabla de multiplicar (por la salida estándar)
inicio
  $i \longleftarrow 1$
  mientras $i \leq t$ **hacer**
    escribir $f \times i$ sin salto de línea
    $i \longleftarrow i + 1$

```
    escribir un salto de línea
fin
```

El código escrito mediante descomposición en procedimientos tiene dos grandes ventajas:

- Es más fácil de entender un código basado en abstracciones independientes y separadas que se llaman entre sí, antes que un código donde todo está en el mismo nivel de refinamiento formando un texto monolítico de principio a fin.
- Es probable que los procedimientos así obtenidos puedan *reutilizarse* en otros programas con poca o ninguna variación, siempre y cuando sean lo suficientemente genéricos e independientes del resto del programa que los utiliza.

Estas ventajas nos están ya haciendo entender que puede resultar interesante diseñar un programa descomponiéndolo en partes separadas, asunto que veremos con más detalle al estudiar la *programación modular*.

Pero no debemos confundir la programación procedimental con la programación modular, que son términos relacionados pero diferentes.

Asimismo, la mayoría de los lenguajes estructurados permiten la creación de procedimientos, lo que a veces lleva a la confusión de creer que la programación estructurada y la procedimental son el mismo paradigma.

4.4. Funciones imperativas

Cada lenguaje de programación procedimental establece sus propios mecanismos de creación de procedimientos.

En Python, los procedimientos son las denominadas **funciones imperativas**.

En los lenguajes orientados a objetos, los procedimientos serían los **métodos**, que son funciones imperativas que se ejecutan sobre objetos.

Estudiaremos ahora cómo crear y usar funciones imperativas en Python.

Al ser Python un lenguaje orientado a objetos además de procedimental, en su momento veremos también cómo crear métodos haciendo uso de funciones imperativas.

4.4.1. Definición de funciones imperativas

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

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

Las funciones imperativas en Python conforman los bloques básicos que nos permiten **descomponer un programa en partes** que se combinan entre sí, lo que resulta el complemento perfecto para la programación estructurada.

Todavía podemos construir funciones mediante expresiones lambda, pero las funciones imperativas tienen ventajas:

- Podemos escribir **sentencias** dentro de las funciones imperativas.

- Podemos escribir funciones que no devuelvan ningún resultado porque su cometido sea provocar algún efecto lateral.

La **definición** de una función imperativa tiene la siguiente sintaxis:

> *⟨definición_función⟩* ::=
> **def** *⟨nombre⟩***(**[*⟨lista_parámetros⟩*]**):**
>   *⟨cuerpo⟩*

donde:

> *⟨lista_parámetros⟩* ::= **identificador** [**,** **identificador**]*
> *⟨cuerpo⟩* ::= *⟨sentencia⟩*

Por ejemplo:

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

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

La definición de una función imperativa es una **sentencia compuesta**, es decir, una **estructura** (como las estructuras de control `if`, `while`, etcétera).

Por tanto, puede aparecer en cualquier lugar del programa donde pueda haber una sentencia.

Como en cualquier otra estructura, las sentencias que contiene (las que van en el **cuerpo** de la función) van **indentadas** (o *sangradas*) dentro de la definición de la función.

Por tanto (de nuevo, como en cualquier otra estructura), el final de la función se deduce al encontrarse una sentencia **menos indentada** que el cuerpo, o bien el final del *script*.

La definición de una función es una sentencia ejecutable que, como cualquier otra definición, **crea una ligadura** entre un identificador (el nombre de la función) y **una variable que almacenará una referencia a la función** dentro del montículo.

La definición de una función **no ejecuta el cuerpo de la función**. El cuerpo se ejecutará únicamente cuando se llame a la función, al igual que ocurría con las expresiones lambda.

Esa definición se ejecuta en un determinado ámbito (normalmente, el ámbito global) y, por tanto, su ligadura y su variable **se almacenarán en el marco del ámbito donde se ha definido la función** (normalmente, el marco global).

Asimismo, **el cuerpo de una función imperativa determina un ámbito**, al igual que ocurría con las expresiones lambda.

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

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

Ámbito global / Ámbito de saluda / Ámbito de despide

El cuerpo de una función determina un ámbito

Nuestra gramática se vuelve a ampliar para incluir las definiciones de funciones imperativas como un caso más de sentencia compuesta:

```
⟨sentencia⟩ ::= ⟨sentencia_simple⟩ | ⟨estructura⟩
⟨estructura⟩ ::= ⟨secuencia⟩
               | ⟨selección⟩
               | ⟨iteración⟩
               | ⟨gestión_excepciones⟩
               | ⟨definición_función⟩
⟨definición_función⟩ ::=
def ⟨nombre⟩([⟨lista_parámetros⟩]):
    ⟨sentencia⟩
```

4.5. Llamadas a funciones imperativas

Cuando se llama a una función imperativa, ocurre lo siguiente (en este orden):

1. Como siempre que se llama a una función, se crea un nuevo marco en el entorno (que contiene las ligaduras y variables locales a su ámbito, incluyendo sus parámetros) y se almacena en la pila de control.

2. Se pasan los argumentos de la llamada a los parámetros de la función, de forma que los parámetros toman los valores de los argumentos correspondientes.

   Recordemos que en Python se sigue el orden aplicativo (o evaluación estricta): primero se evalúan los argumentos y después se pasan a los parámetros correspondientes.

3. El flujo de control del programa se transfiere al bloque de sentencias que forman el cuerpo de la función y se empieza a ejecutar éste.

Cuando se termina de ejecutar el cuerpo de la función (o, dicho de otra forma, cuando se *sale* de la función):

1. Se genera su valor de retorno (en breve veremos cómo).

2. Se saca su marco de la pila.

3. Se devuelve el control de la ejecución a la sentencia que llamó a la función.

4. Se sustituye, en dicha sentencia, la llamada a la función por su valor de retorno.

5. Se continúa la ejecución del programa desde ese punto.

En consecuencia, podemos considerar que la llamada a una función es una sentencia simple que, en realidad, actúa como una sentencia compuesta, una estructura secuencial (o bloque), que es el cuerpo de la función.

Por ejemplo:

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

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

saluda('Pepe')
print('El gusto es mío')
saluda('Juan')
despide()
print('Sayonara, baby')
```

Produce la siguiente salida:

```
Hola Pepe
Encantado de saludarte
El gusto es mío
Hola Juan
Encantado de saludarte
Hasta luego, Lucas
Sayonara, baby
```

Ver ejecución paso a paso en Pythontutor

Una función puede llamar a otra.

Por ejemplo, este programa:

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

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

def quiensoy():
    print('Me llamo Ricardo')

saluda('Pepe')
print('El gusto es mío')
saluda('Juan')
despide()
print('Sayonara, baby')
```

Produce la siguiente salida:

```
Hola Pepe
Me llamo Ricardo
Encantado de saludarte
```

```
El gusto es mío
Hola Juan
Me llamo Ricardo
Encantado de saludarte
Hasta luego, Lucas
Sayonara, baby
```

Ver ejecución paso a paso en Pythontutor

La función debe estar definida antes de poder llamarla.

Eso significa que el intérprete de Python debe ejecutar el **`def`** de una función antes de que el programa pueda llamar a esa función.

Por ejemplo, el siguiente programa lanzaría el error «*NameError: name 'hola' is not defined*» en la línea 1:

```
1 hola()
2
3 def hola():
4     print('hola')
```

En cambio, este funcionaría perfectamente:

```
1 def hola():
2     print('hola')
3
4 hola()
```

4.6. Paso de argumentos

En el marco de la función llamada se almacenan, entre otras cosas, los parámetros de la función.

Al entrar en la función, los parámetros contendrán los valores de los argumentos que se hayan pasado a la función al llamar a la misma.

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***, que en la práctica resulta bastante sencillo:

Lo que hace el intérprete es ***asignar* el argumento al parámetro**, como si hiciera internamente ⟨*parámetro*⟩ = ⟨*argumento*⟩, por lo que se aplica todo lo relacionado con los *alias* de variables, mutabilidad, etc.

Por ejemplo:

```
1 def saluda(persona):
2     print('Hola', persona)
3     print('Encantado de saludarte')
4
```

```
saluda('Manolo')  # Saluda a Manolo
x = 'Juan'
saluda(x)         # Saluda a Juan
```

En la línea 5 se llama a `saluda` asignándole al parámetro `persona` el valor `'Manolo'`.

En la línea 7 se llama a `saluda` asignándole al parámetro `persona` el valor de `x`, como si se hiciera `persona = x`, lo que sabemos que crea un *alias*.

En este caso, la creación del alias no nos afectaría, ya que el valor pasado como argumento es una cadena y, por tanto, inmutable.

En caso de pasar un argumento mutable:

```
def cambia(l):
    print(l)
    l.append(99)

lista = [1, 2, 3]
cambia(lista)     # Imprime [1, 2, 3]
print(lista)      # Imprime [1, 2, 3, 99]
```

La función es capaz de **cambiar el estado interno de la lista que se ha pasado como argumento** porque:

- Al llamar a la función, el argumento `lista` se pasa a la función **asignándola** al parámetro `l` como si hubiera hecho `l = lista`.
- Eso hace que ambas variables sean *alias* una de la otra (se refieren al mismo objeto lista).
- Por tanto, la función está modificando el valor de la variable `lista` que se ha pasado como argumento.

4.7. 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, ocurre lo siguiente (en este orden):

1. Se genera el valor de retorno de la función, que será el valor de la expresión que aparece en la sentencia **`return`**.
2. Se finaliza la ejecución de la función, sacando su marco de la pila.
3. Se devuelve el control a la sentencia que llamó a la función.
4. En esa sentencia, se sustituye la llamada a la función por su valor de retorno (el calculado en el paso 1 anterior).
5. Se continúa la ejecución del programa desde ese punto.

Por ejemplo:

```
1 def suma(x, y):
2     return x + y
3
4 a = int(input('Introduce el primer número: '))
5 b = int(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 las sentencias de su cuerpo 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`, asignándolos a `x` e `y`, respectivamente.

Dentro de la función, en la sentencia **`return`** se calcula la suma `x + y` y se 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 ejecutando la sentencia tal y como está ahora.

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`.

Por tanto, la sentencia **`return`** sin valor de retorno equivale a hacer **`return`** `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
```

Cuando se alcanza el final del cuerpo de una función sin haberse ejecutado antes ninguna sentencia **`return`**, es como si la última sentencia del cuerpo de la función fuese un **`return`** sin valor de retorno.

Por ejemplo:

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

equivale a:

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

Esa última sentencia **`return`** nunca es necesario ponerla, ya que la ejecución de una función termina automáticamente (y retorna al punto donde se la llamó) cuando ya no quedan más sentencias que ejecutar en su cuerpo.

4.8. 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
```

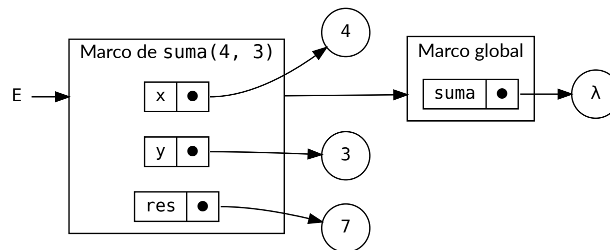
Fuera de la función, la variable `res` no está definida en el entorno (que está formado sólo por el marco global) y por eso da error en la línea 6.

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

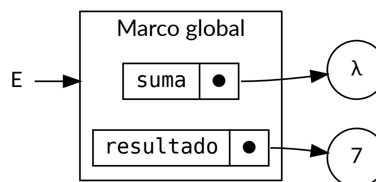
```python
def suma(x, y):
    res = x + y
    return res

resultado = suma(4, 3)
print(resultado)
```

Ese marco es, por tanto, el **espacio de nombres** donde se almacenará todo lo que sea local a la función.



Entorno dentro de la función `suma`



Entorno en la línea 6

4.8.1. Variables locales

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

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

Su **ámbito** es **el cuerpo de la función** a la que pertenecen.

Los **parámetros** se pueden usar libremente en cualquier parte del cuerpo de la función porque ya se les ha asignado un valor.

En cambio, se produce un error `UnboundLocalError` si se intenta usar una **variable local** antes de asignarle un valor:

```
>>> def hola():
...     print(x)  # x es una variable local pero aún no tiene valor asignado
...     x = 1     # aquí es donde empieza a tener un valor
...
>>> hola()
UnboundLocalError: local variable 'x' referenced before assignment
```

4.8.2. Variables globales

Desde dentro de una función es posible usar variables globales, ya que se encuentran en el **entorno** de la función.

Se puede **consultar** el valor de una variable global directamente:

```
x = 4           # esta variable es global

def prueba():
    print(x)    # accede a la variable 'x' global, que vale 4

prueba()        # imprime 4
```

Pero para poder **cambiar** una variable global 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           # esta variable es global

def prueba():
    x = 5       # crea una variable 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 en cualquier lugar **dentro** de una función, esa variable se considera **local** a la función.

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

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*».

4.8.2.1. `global`

Para informar al intérprete que una determinada variable es global, se usa la sentencia **`global`**:

```
x = 4

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

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

La sentencia «**`global`** `x`» es una **declaración** que informa al intérprete de que la variable `x` debe buscarla únicamente en el marco global y que, por tanto, debe saltarse los demás marcos que haya en el entorno.

Si la variable global no existe en el momento de realizar la asignación, se crea. Por tanto, una función puede crear nuevas variables globales usando **`global`**:

```
def prueba():
    global y  # informa que la variable 'y' (que aún no existe) es global
    y = 9     # se crea una nueva variable global 'y' que antes no existía

prueba()
print(y)  # imprime 9
```

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

- Cuando se crea una variable **dentro** de una función (asignándole un valor), por omisión es **local**.
- Cuando se crea una variable **fuera** de una función, por omisión es **global** (no hace falta usar la sentencia **`global`**).
- Se usa la sentencia **`global`** para cambiar el valor de una variable global *dentro* de una función (si la variable global no existía previamente, se crea durante la asignación).
- El uso de la sentencia **`global`** *fuera* de una función no tiene ningún efecto.
- La sentencia **`global`** debe aparecer *antes* de que se use la variable global correspondiente.

4.8.2.2. 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 desde el exterior de la función**, más allá de calcular y devolver su valor de retorno.

Por ejemplo:

- Cuando cambia el valor de una variable global.
- Cuando cambia un argumento mutable.
- Cuando realiza una operación de entrada/salida.
- Cuando llama a otras funciones que provocan efectos laterales.

Los efectos laterales hacen que el comportamiento de un programa sea más difícil de predecir.

La pureza o impureza de una función tienen mucho que ver con los efectos laterales.

Una función es **pura** si, desde el punto de vista de un observador externo, el único efecto que produce es calcular su valor de retorno, el cual sólo depende del valor de sus argumentos.

Por tanto, una función es **impura** si cumple al menos una de las siguientes condiciones:

- **Provoca efectos laterales**, porque está haciendo algo más que calcular su valor de retorno.
- Su valor de retorno depende de algo más que de sus argumentos (p. ej., de una variable global).

En una expresión, no podemos sustituir libremente una llamada a una función impura por su valor de retorno.

Por tanto, decimos que una función impura no cumple la **transparencia referencial**.

El siguiente es un ejemplo de **función impura**, ya que, además de calcular su valor de retorno, provoca el **efecto lateral** de ejecutar una **operación de entrada/salida** (la función `print`):

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

Cualquiera que desee usar la función `suma`, pero no sepa cómo está construida internamente, 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 que se obtiene al ejecutar el siguiente programa no es el que cabría esperar:

Programa:

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

Resultado:

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

```
20
```

No podemos sustituir libremente en una expresión las llamadas a la función `suma` por sus valores de retorno correspondientes.

Es decir, no es lo mismo hacer:

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

que hacer:

```
resultado = 7 + 13
```

porque en el primer caso se imprimen cosas por pantalla y en el segundo no.

Por tanto, la función `suma` es **impura** porque **no cumple la *transparencia referencial***, y no la cumple porque provoca un **efecto lateral**.

Si una función necesita **consultar el valor de una variable global**, también **pierde la transparencia referencial**, ya que la convierte en **impura** porque su valor de retorno puede depender de algo más que de sus argumentos (en este caso, del valor 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 (z)
    return res

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

En este caso, la función es **impura** porque, aunque no provoca efectos laterales, sí puede verse afectada por los efectos laterales que provocan otras partes del programa cuando modifican el valor de la variable global `z`.

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

En consecuencia, esa misma 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 += 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
```

O también podría afectar a otras funciones que dependan del valor de la variable global.

En ese caso, **ambas funciones serían impuras: la que provoca el efecto lateral y la que se ve afectada por ella**.

Por ejemplo, las siguientes dos funciones son **impuras**, cada una por un motivo:

```
def cambia(x):
    global z
    z += x             # efecto lateral: cambia una variable global

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

z = 0
print(suma(4, 3))      # imprime 7
cambia(2)              # provoca un efecto lateral
print(suma(4, 3))      # ahora imprime 9
```

`cambia` provoca un efecto lateral y `suma` depende de una variable global.

Aunque los efectos laterales resultan indeseables en general, a veces es precisamente el efecto que deseamos.

Por ejemplo, podemos diseñar una función que modifique los elementos de una lista en lugar de devolver una lista nueva:

```
def cambia(lista, indice, valor):
    lista[indice] = valor         # modifica el argumento recibido

l = [1, 2, 3, 4]
cambia(l, 2, 99)                  # cambia l
print(l)                          # imprime [1, 2, 99, 4]
```

Si la función no pudiera cambiar el interior de la lista que recibe como argumento, tendría que crear una lista nueva, lo que resultaría menos eficiente en tiempo y espacio:

```
def cambia(lista, indice, valor):
    nueva = lista[:]              # hace una copia de la lista
    nueva[indice] = valor         # modifica la copia
    return nueva                  # devuelve la copia

l = [1, 2, 3, 4]
print(cambia(l, 2, 99))           # imprime [1, 2, 99, 4]
```

4.9. Funciones locales a funciones

En Python también podemos definir funciones dentro de funciones:

```
def f(...):
   def g(...):
      ...
```

Cuando definimos una función `g` dentro de otra función `f`, decimos que:

- `g` es un **función local** o **interna** de `f`.
- `f` es la **función externa** de `g`.

También se dice que:

- `g` es una **función anidada** dentro de `f`.
- `f` **contiene** a `g`.

Como `g` se define **dentro** de `f`, sólo es visible dentro de `f`, ya que el **ámbito** de `g` es el cuerpo de `f`.

El uso de funciones locales evita la superpoblación de funciones en un espacio de nombres cuando esa función sólo tiene sentido usarla en un ámbito más local.

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`.

Por tanto, no se puede usar fuera de `fact`, ya que sólo existe en el ámbito de la función `fact` (es decir, en el cuerpo de la función `fact`).

Como `fact_iter` sólo existe para ser usada como función auxiliar de `fact`, tiene sentido definirla como una función local de `fact`.

De esta forma, no contaminaremos el espacio de nombres global con el nombre `fact_iter`, que es el nombre de una función que sólo debe ser usada y conocida por `fact`, y que queda oculta dentro de `fact`.

Tampoco se puede usar `fact_iter` dentro de `fact` *antes* de definirla:

```
1 def fact(n):
2     print(fact_iter(n, 1))  # UnboundLocalError: se usa antes de definirse
3     def fact_iter(n, acc):  # aquí es donde empieza su definición
4         if n == 0:
```

```
5            return acc
6        else:
7            return fact_iter(n - 1, acc * n)
```

Esto ocurre porque la sentencia **def** de la línea 3 crea una ligadura entre `fact_iter` y una variable que apunta a la función que se está definiendo, pero esa ligadura y esa variable sólo empiezan a existir cuando se ejecuta la sentencia **def** en la línea 3, y no antes.

Por tanto, en la línea 2 aún no existe la función `fact_iter` y, por tanto, no se puede usar ahí, dando un error `UnboundLocalError`.

Esto puede verse como una extensión a la regla que vimos anteriormente sobre cuándo considerar a una variable como local, cambiando «*asignación*» por «*definición*» y «*variable*» por «*función*».

Como ocurre con cualquier otra función, las funciones locales también determinan un ámbito.

Ese ámbito, como siempre ocurre, estará anidado dentro del ámbito en el que se define la función.

En este caso, el ámbito de `fact_iter` está anidado dentro del ámbito de `fact`.

Asimismo, como ocurre con cualquier otra función, cuando la ejecución del programa entre en el ámbito de `fact_iter` se creará un nuevo marco en el entorno.

Y, como siempre, ese nuevo marco apuntará al marco del ámbito que lo contiene, es decir, el marco de la función que contiene a la función local.

En este caso, el marco de `fact_iter` apuntará al marco de `fact`, el cual a su vez apuntará al marco global.

4.9.1. nonlocal

Una función local puede **acceder** al valor de las variables locales a la función que la contiene, ya que se encuentran dentro de su ámbito (aunque en otro marco).

En cambio, cuando una función local quiere **cambiar** mediante una asignación el valor de una variable local a la función que la contiene, deberá declararla previamente como **no local** con la sentencia **nonlocal**.

De lo contrario, al intentar cambiar el valor de la variable, el intérprete crearía una nueva variable local a la función actual, que haría sombra a la variable que queremos modificar y que pertenece a otra función.

Es algo similar a lo que ocurre con la sentencia **global** y las variables globales, pero en ámbitos intermedios.

La sentencia «**nonlocal** `n`» es una **declaración** que informa al intérprete de que la variable `n` debe buscarla en el entorno saltándose el marco de la función actual y el marco global.

```
1 def fact(n):
2     def fact_iter(acc):
3         nonlocal n
4         if n == 0:
5             return acc
6         else:
7             acc *= n
```

```
            n -= 1
            return fact_iter(acc)
    return fact_iter(1)

print(fact(5))
```

La función `fact_iter` puede consultar el valor de la variable `n`, ya que es una variable local a la función `fact` y, por tanto, está en el entorno de `fact_iter` (para eso no hace falta declararla como **no local**).

Como, además, `n` está declarada **no local** en `fact_iter` (en la línea 3), la función `fact_iter` también puede modificar esa variable y no hace falta que la reciba como argumento.

Esa instrucción le indica al intérprete que, a la hora de buscar `n` en el entorno de `fact_iter`, debe saltarse el marco de `fact_iter` y el marco global y, por tanto, debe empezar a buscar en el marco de `fact`.

Ejercicios

Ejercicios

1. Considérese la siguiente fórmula (debida a Herón de Alejandría), que expresa el valor de la superficie S de un triángulo cualquiera en función de sus lados, a, b y c:

$$S = \sqrt{\frac{a+b+c}{2}\left(\frac{a+b+c}{2}-a\right)\left(\frac{a+b+c}{2}-b\right)\left(\frac{a+b+c}{2}-c\right)}$$

   Escribir una función que obtenga el valor S a partir de a, b y c, evitando el cálculo repetido del semiperímetro, $sp = \frac{a+b+c}{2}$, y almacenando el resultado finalmente en la variable S.

2. Escribir tres funciones que impriman las siguientes salidas en función de la cantidad de líneas que se desean (· es un espacio en blanco):

```
*****     ·*         ····*····
*****     ··*        ···***···
*****     ···*       ··*****··
*****     ····*      ·*******·
*****     ·····*     ····*····
```

3. Convertir una cantidad de tiempo (en segundos, $\mathbb{Z}$) en la correspondiente en horas, minutos y segundos, con arreglo al siguiente formato:

   3817 segundos = 1 horas, 3 minutos y 37 segundos

4. Escribir un programa que, en primer lugar, lea los coeficientes a_2 , a_1 y a_0 de un polinomio de segundo grado

$$a_2x^2 + a_1x + a_0$$

   y escriba ese polinomio. Y, en segundo, lea el valor de x y escriba qué valor toma el polinomio para esa x.

   Para facilitar la salida, se supondrá que los coeficientes y x son enteros. Por ejemplo, si los coeficientes y x son 1, 2, 3 y 2, respectivamente, la salida puede ser:

```
1x^2 + 2x + 3
p(2) = 9
```

5. Escribir un programa apropiado para cada una de las siguientes tareas:
   a. Pedir los dos términos de una fracción y dar el valor de la división correspondiente, a no ser que sea nulo el hipotético denominador, en cuyo caso se avisará del error.
   b. Pedir los coeficientes de una ecuación de segundo grado y dar las dos soluciones correspondientes, comprobando previamente si el discriminante es positivo o no.
   c. Pedir los coeficientes de la recta $ax+by+c=0$ y dar su pendiente y su ordenada en el origen en caso de que existan, o el mensaje apropiado en otro caso.
   d. Pedir un número natural n y dar sus divisores.
6. Escribir un programa que lea un carácter, correspondiente a un dígito hexadecimal:

   `0`, `1`, ..., `9`, `A`, `B`, ..., `F`

   y lo convierta en el valor decimal correspondiente:

   `0`, `1`, ..., `9`, `10`, `11`, ..., `15`
7. Para hallar en qué fecha cae el Domingo de Pascua de un `anyo` cualquiera, basta con hallar las cantidades `a` y `b` siguientes:

```
a = (19 * (anyo % 19) + 24) % 30
b = (2 * (anyo % 4) + 4 * (anyo % 7) + 6 * a + 5) % 7
```

   y entonces, ese Domingo es el *22 de marzo +* `a` *+* `b` *días*, que podría caer en abril. Escriba un programa que realice estos cálculos, produciendo una entrada y salida claras.
8. Escribir una función para hallar $\binom{n}{k}$, donde n y k son datos enteros positivos,
   a. mediante la fórmula $\frac{n!}{(n-k)!k!}$
   b. mediante la fórmula $\frac{n(n-1)\cdots(k+1)}{(n-k)!}$

   ¿Qué ventajas presenta la segunda con respecto a la primera?

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.