Programación procedimental (I)

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/07/15 a las 15:49:00

Índice

1. Conceptos básicos **2**
1.1. Procedimientos . . . 2
1.2. El paradigma de programación procedimental . . . 2
1.3. Procedimientos y refinamiento sucesivo . . . 3

2. Funciones imperativas **5**
2.1. Introducción . . . 5
2.2. Definición de funciones imperativas . . . 5
2.3. Llamadas a funciones imperativas . . . 8
2.4. Paso de argumentos . . . 10
2.5. La sentencia `return` . . . 11

3. Ámbitos en funciones imperativas **13**
3.1. Ámbito de variables . . . 13
3.1.1. Variables locales . . . 14
3.1.2. Variables globales . . . 15
3.2. Funciones locales a funciones . . . 20
3.2.1. `nonlocal` . . . 21

4. Funciones genéricas **22**
4.1. Definición y uso . . . 22

5. Calidad **23**
5.1. Documentación . . . 23
5.1.1. *Docstrings* . . . 23
5.1.2. `pydoc` . . . 25
5.2. Pruebas . . . 25
5.2.1. `doctest` . . . 25
5.2.2. `pytest` . . . 27

1. Conceptos básicos

1.1. 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**.

1.2. 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.

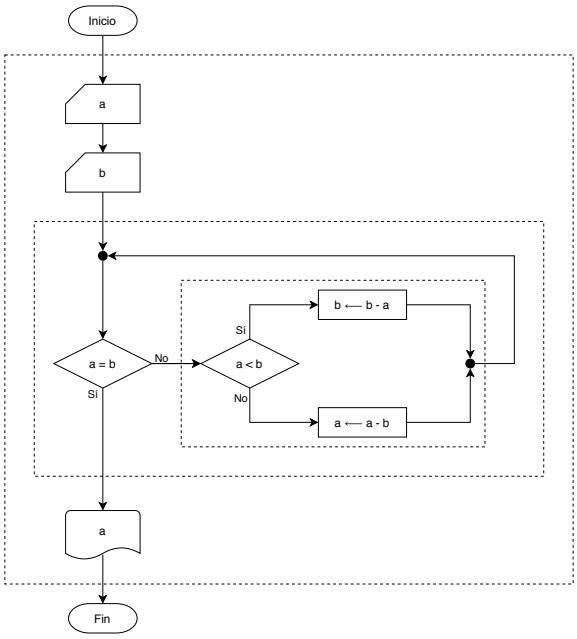
A su vez, los procedimientos pueden devolver un resultado, de ser necesario.

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.

1.3. 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.

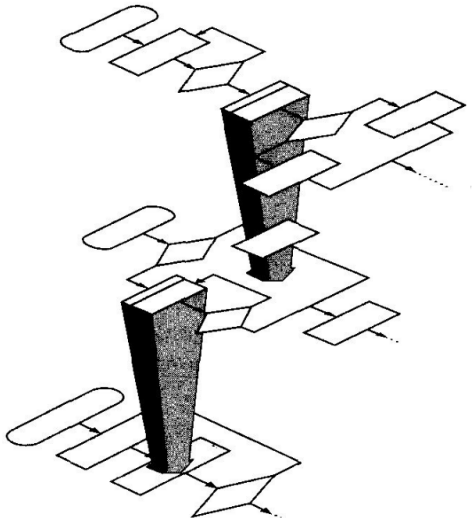
De igual forma, una llamada a un procedimiento es una sentencia simple que actúa como una sentencia compuesta, formada por varias instrucciones (el *cuerpo* del procedimiento) que actúan como una sola.

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.

2. Funciones imperativas

2.1. Introducción

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.

2.2. 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:

```
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.

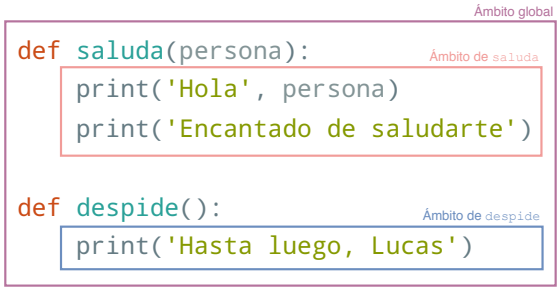
Por tanto (de nuevo, como en cualquier otra estructura), el final de la función se deduce al encontrarse una sentencia **menos indentada** que el cuerpo, o bien el final del *script*.

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.



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*⟩

En la definición de un función, podemos indicar los tipos tanto de los parámetros como del valor de retorno de la función.

Esos tipos son, efectivamente, **anotaciones de tipo** (o *type hints* en inglés), y ya los hemos visto anteriormente, por ejemplo al escribir la **signatura** de una función.

En realidad, la primera línea de la definición de una función (que técnicamente se denomina la **cabecera** de la función) podría entenderse que es la signatura de la función, precedida por la palabra clave **`def`**.

Por ejemplo, la siguiente definición de una función que saluda a una persona a partir de su nombre y sus apellidos:

```
def saluda(nombre, apellidos):
    print('Hola', nombre, apellidos)
```

se podría anotar de la siguiente forma:

```
def saluda(nombre: str, apellidos: str) -> None:
    print('Hola', nombre, apellidos)
```

lo que nos lleva a que la función tiene la siguiente signatura:

`saluda(`*nombre*`: str, `*apellidos*`: str) -> None`

La función anterior devuelve `None` porque realmente no devuelve ningún valor (su único cometido

es escribir en la consola usando `print`).

Por tanto, nuestra sintaxis se puede ampliar para incluir las anotaciones de tipo en las definiciones de las funciones:

⟨*definición_función*⟩ ::=
def ⟨*nombre*⟩**(**[⟨*lista_parámetros*⟩]**)**[**->** ⟨*tipo*⟩]**:**
    ⟨*cuerpo*⟩
⟨*lista_parámetros*⟩ ::= **identificador**[[**:** ⟨*tipo*⟩] [**,** **identificador**]*

Es importante recordar que el intérprete no comprueba en ningún momento las anotaciones de tipo para verificar que son correctas. Para ello se utilizan herramientas externas como `pylint` o `mypy`.

2.3. 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:

```
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:

```
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()
```

2.4. 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
5 saluda('Manolo')  # Saluda a Manolo
6 x = 'Juan'
7 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.

Recordemos de nuevo que el intérprete no comprueba si los tipos de los argumentos coinciden con los esperados en la lista de parámetros cuando se usan anotaciones de tipos.

Por tanto, el intérprete de Python ejecutará el siguiente programa sin dar ningún error:

```
def saluda(nombre: str) -> None:
    print('Hola', nombre)

saluda(2)
```

El intérprete no se quejará aunque en la llamada a la función le hayamos pasado un argumento de tipo `int` cuando deberíamos haberle pasado uno de tipo `str`, según se indica en su lista de parámetros.

En este caso, además, la función `print` tampoco dará ningún error ya que puede imprimir valores de tipo entero.

2.5. 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.

3. Ámbitos en funciones imperativas

3.1. 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:

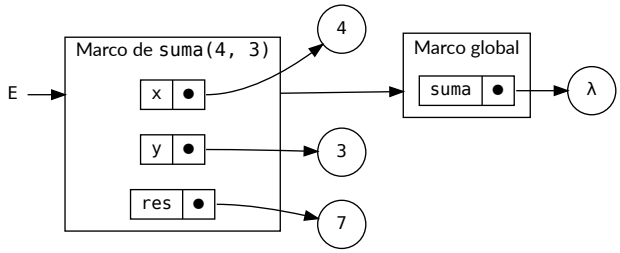
```
1 def suma(x, y):
2     res = x + y
3     return res
4
5 resultado = suma(4, 3)
6 print(res)  # da error
```

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.

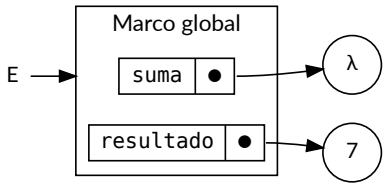
Eso significa que se crea un nuevo marco en el entorno que contendrá, al menos, los parámetros, las variables locales y las ligaduras locales a la función.

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

3.1.1. 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
```

3.1.2. 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*».

3.1.2.1. `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.

3.1.2.2. 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`):

```
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:

```
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]
```

3.2. 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.

3.2.1. `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
```

```
8               n -= 1
9               return fact_iter(acc)
10      return fact_iter(1)
11
12  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`.

4. Funciones genéricas

4.1. Definición y uso

Las funciones genéricas se definen de la siguiente forma:

```
def func[T](arg: T): ...
```

En esta definición, `T` es una **variable de tipo**, es decir, un identificador que representa a un *tipo* cualquiera que en este momento no está determinado.

Las variables de tipo sirven, por ejemplo, para indicar que todos los elementos de una colección son del mismo tipo.

Por otra parte, `[T]` representa un **parámetro de tipo**, que sirve para expresar el hecho de que la función que estamos definiendo es genérica y funciona con valores de muchos tipos distintos.

Esta forma de expresar funciones describe un cierto tipo de polimorfismo llamado **polimorfismo paramétrico**, donde la misma función puede actuar sobre valores de tipos muy diversos. Por eso, a esas funciones las podemos llamar **funciones polimórficas**.

Por ejemplo, la función que devuelve el máximo elemento de una lista, se puede anotar de la siguiente forma:

```
def maximo[T](l: list[T]) -> T:
  ...
```

y podríamos llamarla con cualquier tipo de lista, siempre que todos los elementos de la lista sean del mismo tipo:

```
>>> maximo([1, 2, 3, 4])
4
>>> maximo(['a', 'b', 'c', 'd'])
'd'
```

Según la signatura de la función, no sería correcto pasarle una lista con elementos de diferentes tipos, pero el intérprete no detectaría el error ya que no hace comprobación de tipos. De todas formas, la función probablemente no funcionaría bien en ese caso, ya que es una violación de su especificación:

```
>>> maximo([1, 'a', True, 2.5])  # Incorrecto si se comprueban los tipos
???                              # No se sabe cómo se comportará la función en este caso
```

5. Calidad

5.1. Documentación

5.1.1. *Docstrings*

La **cadena de documentación** (*docstring*) de una función es un literal de tipo cadena que aparece como primera sentencia de la función.

Las *docstrings* son comentarios que tienen la finalidad de **documentar** la función correspondiente.

Por convenio, las *docstrings* siempre se delimitan mediante triples dobles comillas (`"""`).

La función `help` muestran la *docstring* de la función para el que se solicita la ayuda.

Internamente, la *docstring* se almacena en el atributo `__doc__` de la función, ya que para Python una función también es un objeto.

Ejemplo

```
def saluda(nombre):
    """Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.

    Returns:
        str: El saludo.
    """
    return "¡Hola, " + nombre + "!"
```

Existen dos formas distintas de *docstrings*:

- **De una sola línea (*one-line*):** para casos muy obvios que necesiten poca explicación.
- **De varias líneas (*multi-line*):** para casos donde se necesita una explicación más detallada.

```
>>> help(saluda)
Help on function saluda in module ejemplo:

saluda(nombre)
    Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.
```

```
    Returns:
        str: El saludo.
```

Lo que hace básicamente la función `help(`*función*`)` es acceder al contenido del atributo `__doc__` de la función y mostrarlo de forma legible.

Siempre podemos acceder directamente al atributo `__doc__` para recuperar la *docstring* original usando *función*`.__doc__`:

```
>>> print(saluda.__doc__)
Devuelve un saludo.

    Args:
        nombre (str): El nombre de la persona a la que saluda.

    Returns:
        str: El saludo.
```

Esta información también es usada por otras herramientas de documentación externa, como `pydoc`.

¿Cuándo y cómo usar cada forma de *docstring*?

***Docstrings* de una sola línea:**

- Más apropiada para funciones sencillas.
- Las comillas de apertura y cierre deben aparecer en la misma línea.
- No hay líneas en blanco antes o despues de la *docstring*.
- Debe ser una frase acabada en punto que describa el efecto de la función («Hace *esto*», «Devuelve *aquello*»...).
- No debe ser una signatura, así que lo siguiente está mal:

  ```
  def funcion(a, b):
      """funcion(a, b) -> tuple"""
  ```

  Esto está mejor:

  ```
  def funcion(a, b):
      """Hace esto y aquello, y devuelve una tupla."""
  ```

***Docstrings* de varias líneas:**

- Toda la *docstring* debe ir indentada al mismo nivel que las comillas de apertura.
- La primera línea debe ser un resumen informativo y caber en 80 columnas.

  Puede ir en la misma línea que las comillas de apertura, o en la línea siguiente.
- A continuación, debe ir una línea en blanco, seguida de una descripción más detallada.
- La *docstring* de un módulo debe enumerar los elementos que exporta, con una línea resumen para cada uno.

- La *docstring* de una función debe resumir su comportamiento y documentar sus argumentos, valores de retorno, efectos laterales, excepciones que lanza y precondiciones (si tiene).

5.1.2. `pydoc`

El módulo `pydoc` es un generador automático de documentación para programas Python.

La documentación generada se puede presentar en forma de páginas de texto en la consola, enviada a un navegador web o guardada en archivos HTML.

Dicha documentación se genera a partir de los *docstrings* de los elementos que aparecen en el código fuente del programa.

La función `help` llama al sistema de ayuda en línea del intérprete interactivo, el cual usa `pydoc` para generar su documentación en forma de texto para la la consola.

En la línea de órdenes del sistema operativo, se puede usar `pydoc` pasándole el nombre de una función, módulo o atributo:

1. Si no se indican más opciones, se visualizará en pantalla la documentación del objeto indicado:

```
$ pydoc sys
$ pydoc len
$ pydoc sys.argv
```

2. Con la opción `-w` se genera un archivo HTML:

```
$ pydoc -w ejemplo
wrote ejemplo.html
```

3. Con la opción `-b` se arranca un servidor HTTP y se abre el navegador para visualizar la documentación:

```
$ pydoc -b
Server ready at http://localhost:45373/
Server commands: [b]rowser, [q]uit
server>
```

5.2. Pruebas

5.2.1. `doctest`

`doctest` es una herramienta que permite realizar pruebas de forma automática sobre una función.

Para ello, se usa la *docstring* de la función.

En ella, se escribe una *simulación* de una pretendida ejecución de la función desde el intérprete interactivo de Python.

La herramienta comprueba si la salida obtenida coincide con la esperada según dicta la *docstring* de la función.

De esta forma, la *docstring* cumple dos funciones:

- Documentación de la función.
- Especificación de casos de prueba de la función.

```python
# ejemplo.py
def factorial(n):
    """Devuelve el factorial de n, un número entero >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n debe ser >= 0
    """

    import math
    if not n >= 0:
        raise ValueError("n debe ser >= 0")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result
```

```
$ python -m doctest ejemplo.py
$ python -m doctest ejemplo.py -v
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: n debe ser >= 0
ok
1 items had no tests:
    ejemplo
1 items passed all tests:
   3 tests in ejemplo.factorial
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

5.2.2. pytest

`pytest` es una herramienta que permite realizar pruebas automáticas sobre una función o programa Python, pero de una manera más general que con `doctest`.

La forma más sencilla de usarla es crear una función llamada `test_`⟨*nombre*⟩ por cada función ⟨*nombre*⟩ que queramos probar.

Esa función `test_`⟨*nombre*⟩ será la encargada de probar automáticamente el funcionamiento correcto de la función ⟨*nombre*⟩.

Dentro de la función `test_`⟨*nombre*⟩, usaremos la orden **`assert`** para comprobar si se cumple una determinada condición.

En caso de que no se cumpla, se entenderá que la función ⟨*nombre*⟩ no ha superado dicha prueba.

En Python 3, la herramienta se llama `pytest-3` y se instala mediante:

```
$ sudo apt install python3-pytest
```

```
# test_ejemplo.py
def inc(x):
    return x + 1


def test_respuesta():
    assert inc(3) == 5
```

```
$ pytest-3
======================== test session starts ========================
platform linux -- Python 3.8.5, pytest-4.6.9, py-1.8.1, pluggy-0.13.0
rootdir: /home/ricardo/python
collected 1 item

test_ejemplo.py F                                                  [100%]

============================= FAILURES ==============================
____________________________ test_respuesta _____________________________

    def test_respuesta():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_ejemplo.py:7: AssertionError
====================== 1 failed in 2.48 seconds =====================
```

`pytest` sigue la siguiente estrategia a la hora de localizar pruebas:

- Si no se especifica ningún argumento, empieza a buscar recursivamente empezando en el directorio actual.
- En esos directorios, busca todos los archivos `test_*.py` o `*_test.py`.
- En esos archivos, localiza todas las funciones cuyo nombre empiece por `test`.

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. 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.

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.