PYTHON

**4. Más herramientas para control de flujo**

Además de la sentencia [while](https://docs.python.org/es/3/reference/compound_stmts.html#while) que acabamos de introducir, Python utiliza algunas más que encontraremos en este capítulo.

**4.1. La sentencia if**

Tal vez el tipo más conocido de sentencia sea el [if](https://docs.python.org/es/3/reference/compound_stmts.html#if). Por ejemplo:

>>>

x = int(input("Please enter an integer: "))

Please enter an integer: 42

if x < 0:

x = 0

print('Negative changed to zero')

elif x == 0:

print('Zero')

elif x == 1:

print('Single')

else:

print('More')

More

Puede haber cero o más bloques [elif](https://docs.python.org/es/3/reference/compound_stmts.html#elif), y el bloque [else](https://docs.python.org/es/3/reference/compound_stmts.html#else) es opcional. La palabra reservada elif es una abreviación de “else if”, y es útil para evitar un sangrado excesivo. Una secuencia if … elif … elif … sustituye las sentencias switch o case encontradas en otros lenguajes.

Si necesitas comparar un mismo valor con muchas constantes, o comprobar que tenga un tipo o atributos específicos puede que encuentres útil la sentencia match. Para más detalles véase [La sentencia match](https://docs.python.org/es/3/tutorial/controlflow.html#tut-match).

**4.2. La sentencia for**

La sentencia [for](https://docs.python.org/es/3/reference/compound_stmts.html#for) en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia for de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo:

>>>

# Measure some strings:

words = ['cat', 'window', 'defenestrate']

for w in words:

print(w, len(w))

cat 3

window 6

defenestrate 12

Código que modifica una colección mientras se itera sobre la misma colección puede ser complejo de hacer bien. Sin embargo, suele ser más directo iterar sobre una copia de la colección o crear una nueva colección:

# Create a sample collection

users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}

# Strategy: Iterate over a copy

for user, status in users.copy().items():

if status == 'inactive':

del users[user]

# Strategy: Create a new collection

active\_users = {}

for user, status in users.items():

if status == 'active':

active\_users[user] = status

**4.3. La función** [**range()**](https://docs.python.org/es/3/library/stdtypes.html#range)

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función integrada [range()](https://docs.python.org/es/3/library/stdtypes.html#range), la cual genera progresiones aritméticas:

>>>

for i in range(5):

print(i)

0

1

2

3

4

El valor final dado nunca es parte de la secuencia; range(10) genera 10 valores, los índices correspondientes para los ítems de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama “paso”):

>>>

list(range(5, 10))

[5, 6, 7, 8, 9]

list(range(0, 10, 3))

[0, 3, 6, 9]

list(range(-10, -100, -30))

[-10, -40, -70]

Para iterar sobre los índices de una secuencia, puedes combinar [range()](https://docs.python.org/es/3/library/stdtypes.html#range) y [len()](https://docs.python.org/es/3/library/functions.html#len) así:

>>>

a = ['Mary', 'had', 'a', 'little', 'lamb']

for i in range(len(a)):

print(i, a[i])

0 Mary

1 had

2 a

3 little

4 lamb

En la mayoría de los casos, sin embargo, conviene usar la función [enumerate()](https://docs.python.org/es/3/library/functions.html#enumerate), ver [Técnicas de iteración](https://docs.python.org/es/3/tutorial/datastructures.html#tut-loopidioms).

Algo extraño sucede si tan sólo muestras un range:

>>>

range(10)

range(0, 10)

El objeto retornado por [range()](https://docs.python.org/es/3/library/stdtypes.html#range) se comporta de muchas maneras como si fuera una lista, pero no lo es. Es un objeto que retorna los ítems sucesivos de la secuencia deseada cuando iteras sobre él, pero realmente no construye la lista, ahorrando entonces espacio.

Decimos que tal objeto es [iterable](https://docs.python.org/es/3/glossary.html#term-iterable); esto es, que se puede usar en funciones y construcciones que esperan algo de lo cual obtener ítems sucesivos hasta que se termine. Hemos visto que la declaración [for](https://docs.python.org/es/3/reference/compound_stmts.html#for) es una de esas construcciones, mientras que un ejemplo de función que toma un iterable es la función [sum()](https://docs.python.org/es/3/library/functions.html#sum):

>>>

sum(range(4)) # 0 + 1 + 2 + 3

6

Más adelante veremos otras funciones que aceptan iterables cómo argumentos o retornan iterables. En el capítulo [Estructuras de datos](https://docs.python.org/es/3/tutorial/datastructures.html#tut-structures), discutiremos en más detalle sobre la [list()](https://docs.python.org/es/3/library/stdtypes.html#list).

**4.4. Las sentencias** [**break**](https://docs.python.org/es/3/reference/simple_stmts.html#break)**,** [**continue**](https://docs.python.org/es/3/reference/simple_stmts.html#continue)**, y** [**else**](https://docs.python.org/es/3/reference/compound_stmts.html#else) **en bucles**

La sentencia [break](https://docs.python.org/es/3/reference/simple_stmts.html#break) termina el bucle [for](https://docs.python.org/es/3/reference/compound_stmts.html#for) o [while](https://docs.python.org/es/3/reference/compound_stmts.html#while) más anidado.

Un bucle for o while puede incluir una cláusula else.

En un bucle [for](https://docs.python.org/es/3/reference/compound_stmts.html#for), la cláusula else se ejecuta después de que el bucle alcance su iteración final.

En un bucle [while](https://docs.python.org/es/3/reference/compound_stmts.html#while), se ejecuta después de que la condición del bucle se vuelva falsa.

En cualquier tipo de bucle, la cláusula else **no** se ejecuta si el bucle ha finalizado con [break](https://docs.python.org/es/3/reference/simple_stmts.html#break).

Esto se ejemplifica en el siguiente bucle for, que busca números primos:

>>>

for n in range(2, 10):

for x in range(2, n):

if n % x == 0:

print(n, 'equals', x, '\*', n//x)

break

else:

# loop fell through without finding a factor

print(n, 'is a prime number')

2 is a prime number

3 is a prime number

4 equals 2 \* 2

5 is a prime number

6 equals 2 \* 3

7 is a prime number

8 equals 2 \* 4

9 equals 3 \* 3

(Sí, este es el código correcto. Fíjate bien: el else pertenece al ciclo [for](https://docs.python.org/es/3/reference/compound_stmts.html#for), **no** al [if](https://docs.python.org/es/3/reference/compound_stmts.html#if).)

Cuando se usa con un bucle, la cláusula else tiene más en común con el else de una sentencia [try](https://docs.python.org/es/3/reference/compound_stmts.html#try) que con el de un [if](https://docs.python.org/es/3/reference/compound_stmts.html#if): en una sentencia [try](https://docs.python.org/es/3/reference/compound_stmts.html#try) la cláusula else se ejecuta cuando no se genera ninguna excepción, y el else de un bucle se ejecuta cuando no hay ningún break. Para más sobre la declaración try y excepciones, mira [Gestionando excepciones](https://docs.python.org/es/3/tutorial/errors.html#tut-handling).

La declaración [continue](https://docs.python.org/es/3/reference/simple_stmts.html#continue), también tomada de C, continua con la siguiente iteración del ciclo:

>>>

for num in range(2, 10):

if num % 2 == 0:

print("Found an even number", num)

continue

print("Found an odd number", num)

Found an even number 2

Found an odd number 3

Found an even number 4

Found an odd number 5

Found an even number 6

Found an odd number 7

Found an even number 8

Found an odd number 9

**4.5. La sentencia** [**pass**](https://docs.python.org/es/3/reference/simple_stmts.html#pass)

La sentencia [pass](https://docs.python.org/es/3/reference/simple_stmts.html#pass) no hace nada. Se puede usar cuando una sentencia es requerida por la sintaxis pero el programa no requiere ninguna acción. Por ejemplo:

>>>

while True:

pass # Busy-wait for keyboard interrupt (Ctrl+C)

Se usa normalmente para crear clases en su mínima expresión:

>>>

class MyEmptyClass:

pass

Otro lugar donde se puede usar [pass](https://docs.python.org/es/3/reference/simple_stmts.html#pass) es como una marca de lugar para una función o un cuerpo condicional cuando estás trabajando en código nuevo, lo cual te permite pensar a un nivel de abstracción mayor. El pass se ignora silenciosamente:

>>>

def initlog(\*args):

pass # Remember to implement this!

**4.6. La sentencia match**

Una sentencia [match](https://docs.python.org/es/3/reference/compound_stmts.html#match) recibe una expresión y compara su valor con patrones sucesivos dados en uno o más bloques case. Esto es similar a grandes rasgos con una sentencia switch en C, Java o JavaScript (y muchos otros lenguajes) pero es más similar a la comparación de patrones en lenguajes como Rust o Haskell. Sólo se ejecuta el primer patrón que coincide y también es capaz de extraer componentes (elementos de una secuencia o atributos de un objeto) de un valor y ponerlos en variables.

La forma más simple compara un valor expuesto con uno o más literales:

def http\_error(status):

match status:

case 400:

return "Bad request"

case 404:

return "Not found"

case 418:

return "I'm a teapot"

case \_:

return "Something's wrong with the internet"

Observa el último bloque: el «nombre de variable» \_ funciona como un *comodín* y nunca fracasa la coincidencia. Si ninguno de los casos case coincide, ninguna de las ramas es ejecutada.

Se pueden combinar varios literales en un solo patrón usando | («o»):

case 401 | 403 | 404:

return "Not allowed"

Los patrones pueden también verse como asignaciones que desempaquetan, y pueden usarse para ligar variables:

# point is an (x, y) tuple

match point:

case (0, 0):

print("Origin")

case (0, y):

print(f"Y={y}")

case (x, 0):

print(f"X={x}")

case (x, y):

print(f"X={x}, Y={y}")

case \_:

raise ValueError("Not a point")

¡Observa éste caso con cuidado! El primer patrón tiene dos literales y puede considerarse una extensión del patrón literal que se mostró anteriormente. Pero los siguientes dos patrones combinan un literal y una variable, y la variable *liga* uno de los elementos del sujeto (point). El cuarto patrón captura ambos elementos, lo que lo hace conceptualmente similar a la asignación que desempaqueta (x, y) = point.

Si estás usando clases para estructurar tus datos, puedes usar el nombre de la clase seguida de una lista de argumentos similar a la de un constructor, pero con la capacidad de capturar atributos en variables:

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def where\_is(point):

match point:

case Point(x=0, y=0):

print("Origin")

case Point(x=0, y=y):

print(f"Y={y}")

case Point(x=x, y=0):

print(f"X={x}")

case Point():

print("Somewhere else")

case \_:

print("Not a point")

Puedes usar argumentos posicionales en algunas clases incorporadas que proveen un orden para sus atributos (por ej. dataclasses). También puedes definir una posición especifica para los atributos de los patrones si asignas en tu clase el atributo especial \_\_match\_args\_\_. Si le asignas («x», «y»), los siguientes patrones son todos equivalentes entre sí (y todos ligan el atributo y a la variable var):

Point(1, var)

Point(1, y=var)

Point(x=1, y=var)

Point(y=var, x=1)

Una recomendación para leer patrones es verlos como una forma extendida de lo que pondrías en el lado izquierdo de una asignación, para así entender cuáles variables tomarían qué valores. Sólo los nombres que aparecen por si solos (cómo var arriba) son asignados por una sentencia match. Nunca se asigna a los nombres con puntos (como foo.bar), nombres de atributos (los x= e y= arriba) o nombres de clases (reconocidos por los «(…)» junto a ellos, como Point arriba).

Los patrones pueden anidarse arbitrariamente. Por ejemplo, si tuviéramos una lista corta de puntos, con \_\_match\_args\_\_ añadido, podríamos aplicar match así:

class Point:

\_\_match\_args\_\_ = ('x', 'y')

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

match points:

case []:

print("No points")

case [Point(0, 0)]:

print("The origin")

case [Point(x, y)]:

print(f"Single point {x}, {y}")

case [Point(0, y1), Point(0, y2)]:

print(f"Two on the Y axis at {y1}, {y2}")

case \_:

print("Something else")

Podemos añadir una clausula if a un patrón, conocida como «guarda». Si la guarda es falsa, match pasa a intentar el siguiente bloque case. Obsérvese que la captura de valores sucede antes de que la guarda sea evaluada:

match point:

case Point(x, y) if x == y:

print(f"Y=X at {x}")

case Point(x, y):

print(f"Not on the diagonal")

Algunas otras propiedades importantes de esta sentencia:

* Al igual que las asignaciones con desempaquetado, los patrones de lista o tupla tienen exactamente el mismo sentido y realmente coinciden con cualquier secuencia arbitraria. Una excepción importante es que no coinciden ni con iteradores ni con cadenas de caracteres.
* Los patrones de secuencia soportan desempaquetado extendido: [x, y, \*otros] y (x, y, \*otros) funcionan de manera similar a las asignaciones con desempaquetado. El nombre luego de \* también puede ser \_, con lo cual (x, y, \*\_) coincide con cualquier secuencia de al menos dos elementos, sin ligar ninguno de los demás elementos.
* Los patrones de mapeo: {"ancho\_de\_banda": c, "latencia": l} capturan los valores "ancho\_de\_banda" y "latencia" de un diccionario. A diferencia de los patrones de secuencia, las claves adicionales son ignoradas. Puede usarse un desempaquetado como \*\*otros . (Aunque \*\*\_ sería redundante, con lo cual no está permitido)
* Pueden capturarse subpatrones usando la palabra clave as:
* case (Point(x1, y1), Point(x2, y2) as p2): ...

capturará el segundo elemento de la entrada en p2 (siempre y cuando la entrada sea una secuencia de dos puntos)

* La mayoría de los literales se comparan por igualdad, pero las instancias únicas True, False y None se comparan por identidad.
* En un patrón pueden usarse constantes con nombres. Los nombres deben tener puntos para impedir que sean interpretados como variables a capturar:
* from enum import Enum
* class Color(Enum):
* RED = 'red'
* GREEN = 'green'
* BLUE = 'blue'
* color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))
* match color:
* case Color.RED:
* print("I see red!")
* case Color.GREEN:
* print("Grass is green")
* case Color.BLUE:
* print("I'm feeling the blues :(")

Para una explicación más detallada y más ejemplos, puede leerse [**PEP 636**](https://peps.python.org/pep-0636/) que está escrita en un formato de tutorial.

**4.7. Definir funciones**

Podemos crear una función que escriba la serie de Fibonacci hasta un límite determinado:

>>>

def fib(n): # write Fibonacci series up to n

"""Print a Fibonacci series up to n."""

a, b = 0, 1

while a < n:

print(a, end=' ')

a, b = b, a+b

print()

# Now call the function we just defined:

fib(2000)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

La palabra reservada [def](https://docs.python.org/es/3/reference/compound_stmts.html#def) se usa para definir funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

La primera sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o *docstring*. (Puedes encontrar más acerca de docstrings en la sección [Cadenas de texto de documentación](https://docs.python.org/es/3/tutorial/controlflow.html#tut-docstrings).). Existen herramientas que usan las docstrings para producir documentación imprimible o disponible en línea, o para dejar que los usuarios busquen interactivamente a través del código; es una buena práctica incluir docstrings en el código que escribes, así que acostúmbrate a hacerlo.

La *ejecución* de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; así mismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, a variables globales y a variables de funciones que engloban a una función no se les puede asignar directamente un valor dentro de una función (a menos que se las nombre en la sentencia [global](https://docs.python.org/es/3/reference/simple_stmts.html#global), o mediante la sentencia [nonlocal](https://docs.python.org/es/3/reference/simple_stmts.html#nonlocal) para variables de funciones que engloban la función local), aunque si pueden ser referenciadas.

Los parámetros reales (argumentos) para una llamada de función se introducen en la tabla de símbolos local de la función llamada cuando ésta se llama; por lo tanto, los argumentos se pasan usando *llamada por valor* (donde el *valor* es siempre una *referencia* al objeto, no el valor del objeto). [[1]](https://docs.python.org/es/3/tutorial/controlflow.html#id2) Cuando una función llama a otra función, o se llama a sí misma de forma recursiva, se crea una nueva tabla de símbolos locales para esa llamada.

Una definición de función asocia el nombre de la función con el objeto de función en la tabla de símbolos actual. El intérprete reconoce el objeto al que apunta ese nombre como una función definida por el usuario. Otros nombres también pueden apuntar a ese mismo objeto de función y también se pueden usar para acceder a la función:

>>>

fib

<function fib at 10042ed0>

f = fib

f(100)

0 1 1 2 3 5 8 13 21 34 55 89

Viniendo de otros lenguajes, puedes objetar que fib no es una función, sino un procedimiento, porque no retorna un valor. De hecho, técnicamente hablando, los procedimientos sin [return](https://docs.python.org/es/3/reference/simple_stmts.html#return) sí retornan un valor, aunque uno bastante aburrido. Este valor se llama None (es un nombre predefinido). El intérprete por lo general no escribe el valor None si va a ser el único valor escrito. Puede verlo si realmente lo desea utilizando [print()](https://docs.python.org/es/3/library/functions.html#print):

>>>

fib(0)

print(fib(0))

None

Es simple escribir una función que retorne una lista con los números de la serie de Fibonacci en lugar de imprimirlos:

>>>

def fib2(n): # return Fibonacci series up to n

"""Return a list containing the Fibonacci series up to n."""

result = []

a, b = 0, 1

while a < n:

result.append(a) # see below

a, b = b, a+b

return result

f100 = fib2(100) # call it

f100 # write the result

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

Este ejemplo, como es usual, demuestra algunas características más de Python:

* La sentencia [return](https://docs.python.org/es/3/reference/simple_stmts.html#return) retorna un valor en una función. return sin una expresión como argumento retorna None. Si se alcanza el final de una función, también se retorna None.
* La sentencia result.append(a) llama a un *método* del objeto lista result. Un método es una función que “pertenece” a un objeto y se nombra obj.methodname, dónde obj es algún objeto (puede ser una expresión), y methodname es el nombre del método que está definido por el tipo del objeto. Distintos tipos definen distintos métodos. Métodos de diferentes tipos pueden tener el mismo nombre sin causar ambigüedad. (Es posible definir tus propios tipos de objetos y métodos, usando *clases*, ver [Clases](https://docs.python.org/es/3/tutorial/classes.html#tut-classes)). El método append() mostrado en el ejemplo está definido para objetos lista; añade un nuevo elemento al final de la lista. En este ejemplo es equivalente a result = result + [a], pero más eficiente.

**4.8. Más sobre definición de funciones**

También es posible definir funciones con un número variable de argumentos. Hay tres formas que pueden ser combinadas.

**4.8.1. Argumentos con valores por omisión**

La forma más útil es especificar un valor por omisión para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos que los que permite. Por ejemplo:

def ask\_ok(prompt, retries=4, reminder='Please try again!'):

while True:

reply = input(prompt)

if reply in {'y', 'ye', 'yes'}:

return True

if reply in {'n', 'no', 'nop', 'nope'}:

return False

retries = retries - 1

if retries < 0:

raise ValueError('invalid user response')

print(reminder)

Esta función puede ser llamada de distintas maneras:

* pasando sólo el argumento obligatorio: ask\_ok('Do you really want to quit?')
* pasando uno de los argumentos opcionales: ask\_ok('OK to overwrite the file?', 2)
* o pasando todos los argumentos: ask\_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

Este ejemplo también introduce la palabra reservada [in](https://docs.python.org/es/3/reference/expressions.html#in), la cual prueba si una secuencia contiene o no un determinado valor.

Los valores por omisión son evaluados en el momento de la definición de la función en el ámbito de la definición, entonces:

i = 5

def f(arg=i):

print(arg)

i = 6

f()

imprimirá 5.

**Advertencia importante:** El valor por omisión es evaluado solo una vez. Existe una diferencia cuando el valor por omisión es un objeto mutable como una lista, diccionario, o instancia de la mayoría de las clases. Por ejemplo, la siguiente función acumula los argumentos que se le pasan en subsiguientes llamadas:

def f(a, L=[]):

L.append(a)

return L

print(f(1))

print(f(2))

print(f(3))

Imprimirá

[1]

[1, 2]

[1, 2, 3]

Si no se quiere que el valor por omisión sea compartido entre subsiguientes llamadas, se pueden escribir la función así:

def f(a, L=None):

if L is None:

L = []

L.append(a)

return L

**4.8.2. Palabras claves como argumentos**

Las funciones también puede ser llamadas usando [argumentos de palabras clave](https://docs.python.org/es/3/glossary.html#term-keyword-argument) (o argumentos nombrados) de la forma kwarg=value. Por ejemplo, la siguiente función:

def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):

print("-- This parrot wouldn't", action, end=' ')

print("if you put", voltage, "volts through it.")

print("-- Lovely plumage, the", type)

print("-- It's", state, "!")

…acepta un argumento obligatorio (voltage)) y tres argumentos opcionales (state, action, y type). Esta función puede llamarse de cualquiera de las siguientes maneras:

parrot(1000) # 1 positional argument

parrot(voltage=1000) # 1 keyword argument

parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments

parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments

parrot('a million', 'bereft of life', 'jump') # 3 positional arguments

parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword

…pero estas otras llamadas serían todas inválidas:

parrot() # required argument missing

parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument

parrot(110, voltage=220) # duplicate value for the same argument

parrot(actor='John Cleese') # unknown keyword argument

En una llamada a una función, los argumentos nombrados deben seguir a los argumentos posicionales. Cada uno de los argumentos nombrados pasados deben coincidir con un argumento aceptado por la función (por ejemplo, actor no es un argumento válido para la función parrot), y el orden de los mismos no es importante. Esto también se aplica a los argumentos obligatorios (por ejemplo, parrot(voltage=1000) también es válido). Ningún argumento puede recibir más de un valor al mismo tiempo. Aquí hay un ejemplo que falla debido a esta restricción:

>>>

def function(a):

pass

function(0, a=0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: function() got multiple values for argument 'a'

Cuando un parámetro formal de la forma \*\*nombre está presente al final, recibe un diccionario (ver [Tipos mapa — dict](https://docs.python.org/es/3/library/stdtypes.html#typesmapping)) conteniendo todos los argumentos nombrados excepto aquellos correspondientes a un parámetro formal. Esto puede ser combinado con un parámetro formal de la forma \*nombre (descrito en la siguiente sección) que recibe una [tupla](https://docs.python.org/es/3/tutorial/datastructures.html#tut-tuples) conteniendo los argumentos posicionales además de la lista de parámetros formales. (\*nombre debe ocurrir antes de \*\*nombre). Por ejemplo, si definimos una función así:

def cheeseshop(kind, \*arguments, \*\*keywords):

print("-- Do you have any", kind, "?")

print("-- I'm sorry, we're all out of", kind)

for arg in arguments:

print(arg)

print("-" \* 40)

for kw in keywords:

print(kw, ":", keywords[kw])

Puede ser llamada así:

cheeseshop("Limburger", "It's very runny, sir.",

"It's really very, VERY runny, sir.",

shopkeeper="Michael Palin",

client="John Cleese",

sketch="Cheese Shop Sketch")

…y por supuesto imprimirá:

-- Do you have any Limburger ?

-- I'm sorry, we're all out of Limburger

It's very runny, sir.

It's really very, VERY runny, sir.

----------------------------------------

shopkeeper : Michael Palin

client : John Cleese

sketch : Cheese Shop Sketch

Se debe notar que el orden en el cual los argumentos nombrados son impresos está garantizado para coincidir con el orden en el cual fueron provistos en la llamada a la función.

**4.8.3. Parámetros especiales**

Por defecto, los argumentos pueden enviarse a una función Python o bien por posición o explícitamente por clave. Para legibilidad y rendimiento tiene sentido restringir como se pueden enviar los argumentos, así un desarrollador necesitará mirar solamente la definición de la función para determinar si los argumentos se deben enviar por posición, por posición o clave, o por clave.

La definición de una función puede ser como la siguiente:

def f(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):

----------- ---------- ----------

| | |

| Positional or keyword |

| - Keyword only

-- Positional only

donde / y \* son posicionales. Si se utilizan, esos símbolos indican el tipo de parámetro según la forma en que los argumentos deben enviarse a la función: solo por posición (*positional-only*), por posición o clave (*positional-or-keyword*) y solo por clave (*keyword-only*). Parámetros por clave pueden también denominarse parámetros por nombre o nombrados.

**4.8.3.1. Argumentos posicionales o de palabras claves**

Si / y \* no están presentes en la definición de la función, los parámetros pueden ser pasados a una función posicionalmente o por palabra clave.

**4.8.3.2. Parámetros únicamente posicionales**

Mirando esto con un poco más de detalle, es posible señalar algunos parámetros como *únicamente posicionales*. En ese caso el orden de los parámetros es importante, y los parámetros no pueden ser indicados utilizando palabras claves. Parámetros únicamente posicionales son ubicados antes de una / (barra). La / es utilizada para separar lógicamente parámetros únicamente posicionales del resto. Si no existe una / en la definición de la función, no existen parámetros únicamente posicionales.

Los parámetros luego de una / pueden ser *únicamente posicionales* o *unicamente de palabras claves*.

**4.8.3.3. Argumentos únicamente de palabras clave**

Para señalar parámetros como *unicamente de palabras clave*, indicando que los parámetros deben ser pasados con una palabra clave, indiqué un \* en la lista de argumentos antes del primer parámetro *únicamente de palabras clave*.

**4.8.3.4. Ejemplos de Funciones**

Considere el siguiente ejemplo de definiciones de funciones prestando especial atención a los marcadores / y \*:

>>>

def standard\_arg(arg):

print(arg)

def pos\_only\_arg(arg, /):

print(arg)

def kwd\_only\_arg(\*, arg):

print(arg)

def combined\_example(pos\_only, /, standard, \*, kwd\_only):

print(pos\_only, standard, kwd\_only)

La primer definición de función, standard\_arg, la forma mas familiar, no indica ninguna restricción en las condiciones para llamarla y los parámetros deben ser pasados por posición o utilizando palabras clave:

>>>

standard\_arg(2)

2

standard\_arg(arg=2)

2

La segunda función pos\_only\_arg está restringida a utilizar únicamente parámetros posicionales ya que existe una / en la definición de la función:

>>>

pos\_only\_arg(1)

1

pos\_only\_arg(arg=1)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: pos\_only\_arg() got some positional-only arguments passed as keyword arguments: 'arg'

La tercer función kwd\_only\_args solo permite parámetros con palabras clave, indicado por un \* en la definición de la función:

>>>

kwd\_only\_arg(3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: kwd\_only\_arg() takes 0 positional arguments but 1 was given

kwd\_only\_arg(arg=3)

3

La última utiliza las tres convenciones en una misma definición de función:

>>>

combined\_example(1, 2, 3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: combined\_example() takes 2 positional arguments but 3 were given

combined\_example(1, 2, kwd\_only=3)

1 2 3

combined\_example(1, standard=2, kwd\_only=3)

1 2 3

combined\_example(pos\_only=1, standard=2, kwd\_only=3)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: combined\_example() got some positional-only arguments passed as keyword arguments: 'pos\_only'

Finalmente, considere esta definición de función que contiene una colisión potencial entre los parámetros posicionales name y \*\*kwds que incluye name como una clave:

def foo(name, \*\*kwds):

return 'name' in kwds

No hay una llamada posible que lo haga retornar True ya que la palabra clave 'name' siempre se vinculará al primer parámetro. Por ejemplo:

>>>

foo(1, \*\*{'name': 2})

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: foo() got multiple values for argument 'name'

Pero utilizando / (parámetros únicamente posicionales), es posible ya que permite utilizar name como un parámetro posicional y name como un parámetro de palabras clave:

>>>

def foo(name, /, \*\*kwds):

return 'name' in kwds

foo(1, \*\*{'name': 2})

True

En otras palabras, los nombres de parámetros únicamente posicionales pueden ser utilizados en \*\*kwds sin ambigüedad.

**4.8.3.5. Resumen**

El caso de uso determinará qué parámetros utilizar en una definición de función:

def f(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):

A modo de guía:

* Utilice únicamente posicionales si quiere que el nombre del parámetro esté disponible para el usuario. Esto es útil cuando el nombre del parámetro no tiene un significado real, si se quiere imponer el orden de los parámetros cuando una función es llamada o si necesita tomar algunos parámetros posicionales y palabras claves arbitrarias.
* Utilice parámetros únicamente de palabras clave cuando los nombres de los parámetros tienen un significado y la definición de la función será más entendible usando nombres explícitos o cuando desea evitar que los usuarios dependan de la posición de los parámetros que se pasan.
* En el caso de una API, use solo posicional para evitar que se rompan los cambios de la API si el nombre del parámetro se modifica en el futuro.

**4.8.4. Listas de argumentos arbitrarios**

Finalmente, la opción menos frecuentemente usada es especificar que una función puede ser llamada con un número arbitrario de argumentos. Estos argumentos serán organizados en una tupla (ver [Tuplas y secuencias](https://docs.python.org/es/3/tutorial/datastructures.html#tut-tuples)). Antes del número variable de argumentos, cero o más argumentos normales pueden estar presentes.:

def write\_multiple\_items(file, separator, \*args):

file.write(separator.join(args))

Normalmente estos argumentos *variádicos* serán los últimos en la lista de parámetros formales, porque toman todo el remanente de argumentos que se pasan a la función. Cualquier parámetro que suceda luego del \*args será “sólo de palabra clave”, o sea que sólo se pueden usar como argumentos nombrados y no como posicionales.

>>>

def concat(\*args, sep="/"):

return sep.join(args)

concat("earth", "mars", "venus")

'earth/mars/venus'

concat("earth", "mars", "venus", sep=".")

'earth.mars.venus'

**4.8.5. Desempaquetando una lista de argumentos**

La situación inversa ocurre cuando los argumentos ya están en una lista o tupla pero necesitan ser desempaquetados para llamar a una función que requiere argumentos posicionales separados. Por ejemplo, la función predefinida [range()](https://docs.python.org/es/3/library/stdtypes.html#range) espera los parámetros *inicio* y *fin*. Si estos no están disponibles en forma separada, se puede escribir la llamada a la función con el operador \* para desempaquetar argumentos desde una lista o una tupla:

>>>

list(range(3, 6)) # normal call with separate arguments

[3, 4, 5]

args = [3, 6]

list(range(\*args)) # call with arguments unpacked from a list

[3, 4, 5]

Del mismo modo, los diccionarios pueden entregar argumentos nombrados con el operador \*\*:

>>>

def parrot(voltage, state='a stiff', action='voom'):

print("-- This parrot wouldn't", action, end=' ')

print("if you put", voltage, "volts through it.", end=' ')

print("E's", state, "!")

d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}

parrot(\*\*d)

-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

**4.8.6. Expresiones lambda**

Pequeñas funciones anónimas pueden ser creadas con la palabra reservada [lambda](https://docs.python.org/es/3/reference/expressions.html#lambda). Esta función retorna la suma de sus dos argumentos: lambda a, b: a+b Las funciones Lambda pueden ser usadas en cualquier lugar donde sea requerido un objeto de tipo función. Están sintácticamente restringidas a una sola expresión. Semánticamente, son solo azúcar sintáctica para definiciones normales de funciones. Al igual que las funciones anidadas, las funciones lambda pueden hacer referencia a variables desde el ámbito que la contiene:

>>>

def make\_incrementor(n):

return lambda x: x + n

f = make\_incrementor(42)

f(0)

42

f(1)

43

El ejemplo anterior muestra el uso de una expresión lambda para retornar una función. Otro uso es para pasar pequeñas funciones como argumentos

>>>

pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

pairs.sort(key=lambda pair: pair[1])

pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

**4.8.7. Cadenas de texto de documentación**

Acá hay algunas convenciones sobre el contenido y formato de las cadenas de texto de documentación.

La primera línea debe ser siempre un resumen corto y conciso del propósito del objeto. Para ser breve, no se debe mencionar explícitamente el nombre o tipo del objeto, ya que estos están disponibles de otros modos (excepto si el nombre es un verbo que describe el funcionamiento de la función). Esta línea debe empezar con una letra mayúscula y terminar con un punto.

Si hay más líneas en la cadena de texto de documentación, la segunda línea debe estar en blanco, separando visualmente el resumen del resto de la descripción. Las líneas siguientes deben ser uno o más párrafos describiendo las convenciones para llamar al objeto, efectos secundarios, etc.

El analizador de Python no quita el sangrado de las cadenas de texto literales multi-líneas, entonces las herramientas que procesan documentación tienen que quitarlo si así lo desean. Esto se hace mediante la siguiente convención. La primera línea que no está en blanco *siguiente* a la primer línea de la cadena determina la cantidad de sangría para toda la cadena de documentación. (No podemos usar la primer línea ya que generalmente es adyacente a las comillas de apertura de la cadena y el sangrado no se nota en la cadena de texto). Los espacios en blanco «equivalentes» a este sangrado son luego quitados del comienzo de cada línea en la cadena. No deberían haber líneas con una sangría menor, pero si las hay todos los espacios en blanco del comienzo deben ser quitados. La equivalencia de espacios en blanco debe ser verificada luego de la expansión de tabuladores (a 8 espacios, normalmente).

Este es un ejemplo de un docstring multi-línea:

>>>

def my\_function():

"""Do nothing, but document it.

No, really, it doesn't do anything.

"""

pass

print(my\_function.\_\_doc\_\_)

Do nothing, but document it.

No, really, it doesn't do anything.

**4.8.8. Anotación de funciones**

Las [anotaciones de funciones](https://docs.python.org/es/3/reference/compound_stmts.html#function) son información completamente opcional sobre los tipos usadas en funciones definidas por el usuario (ver [**PEP 3107**](https://peps.python.org/pep-3107/) y [**PEP 484**](https://peps.python.org/pep-0484/) para más información).

Las [anotaciones](https://docs.python.org/es/3/glossary.html#term-function-annotation) se almacenan en el atributo \_\_annotations\_\_ de la función como un diccionario y no tienen efecto en ninguna otra parte de la función. Las anotaciones de los parámetros se definen luego de dos puntos después del nombre del parámetro, seguido de una expresión que evalúa al valor de la anotación. Las anotaciones de retorno son definidas por el literal ->, seguidas de una expresión, entre la lista de parámetros y los dos puntos que marcan el final de la declaración [def](https://docs.python.org/es/3/reference/compound_stmts.html#def). El siguiente ejemplo tiene un argumento posicional, uno nombrado, y el valor de retorno anotado:

>>>

def f(ham: str, eggs: str = 'eggs') -> str:

print("Annotations:", f.\_\_annotations\_\_)

print("Arguments:", ham, eggs)

return ham + ' and ' + eggs

f('spam')

Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}

Arguments: spam eggs

'spam and eggs'

**4.9. Intermezzo: Estilo de programación**

Ahora que estás a punto de escribir piezas de Python más largas y complejas, es un buen momento para hablar sobre *estilo de programación*. La mayoría de los lenguajes pueden ser escritos (o mejor dicho, *formateados*) con diferentes estilos; algunos son mas fáciles de leer que otros. Hacer que tu código sea más fácil de leer por otros es siempre una buena idea, y adoptar un buen estilo de programación ayuda tremendamente a lograrlo.

Para Python, [**PEP 8**](https://peps.python.org/pep-0008/) se erigió como la guía de estilo a la que más proyectos adhirieron; promueve un estilo de programación fácil de leer y visualmente agradable. Todos los desarrolladores Python deben leerlo en algún momento; aquí están extraídos los puntos más importantes:

* Usar sangrías de 4 espacios, no tabuladores.

4 espacios son un buen compromiso entre una sangría pequeña (permite mayor nivel de sangrado)y una sangría grande (más fácil de leer). Los tabuladores introducen confusión y es mejor dejarlos de lado.

* Recortar las líneas para que no superen los 79 caracteres.

Esto ayuda a los usuarios con pantallas pequeñas y hace posible tener varios archivos de código abiertos, uno al lado del otro, en pantallas grandes.

* Usar líneas en blanco para separar funciones y clases, y bloques grandes de código dentro de funciones.
* Cuando sea posible, poner comentarios en una sola línea.
* Usar docstrings.
* Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: a = f(1, 2) + g(3, 4).
* Nombrar las clases y funciones consistentemente; la convención es usar NotacionCamello para clases y minusculas\_con\_guiones\_bajos para funciones y métodos. Siempre usa self como el nombre para el primer argumento en los métodos (ver [Un primer vistazo a las clases](https://docs.python.org/es/3/tutorial/classes.html#tut-firstclasses) para más información sobre clases y métodos).
* No uses codificaciones estrafalarias si esperas usar el código en entornos internacionales. El default de Python, UTF-8, o incluso ASCII plano funcionan bien en la mayoría de los casos.
* De la misma manera, no uses caracteres no-ASCII en los identificadores si hay incluso una pequeñísima chance de que gente que hable otro idioma tenga que leer o mantener el código.