Estructura y elementos del lenguaje (resumen)

# Estructura y elementos del lenguaje (resumen)

Dentro de los lenguajes informáticos, Python, pertenece al grupo de los lenguajes de programación y puede ser clasificado como un lenguaje interpretado de propósito general, de alto nivel, multiplataforma, de tipado dinámico y multiparadigma.

A diferencia de la mayoría de los lenguajes de programación, Python nos provee de reglas de estilos, a fin de poder escribir código fuente más legible y de manera estandarizada. Iremos viendo a lo largo del resumen estas reglas de estilo, definidas a través de la **Python Enhancement Proposal No 8** (PEP 8: https://www.python.org/dev/peps/pep-0008/ (https://www.python.org/dev/peps/pep-0008/)).

El estándar `PEP8` nos dice cómo formatear el código, pero el "El Zen de Python" (PEP 20: https://www.python.org/dev/peps/pep-0020/ (https://www.python.org/dev/peps/pep-0020/)) dice elegantemente: "Lo bello es mejor que lo feo". PEP 20 es más una filosofía y una mentalidad, de eso se trata la filosofía *Pythonic* (**PEP20**):

- Lo bello es mejor que lo feo
- Lo explícito es mejor que lo implícito
- Lo simple es mejor que lo complejo
- Lo plano es mejor que lo anidado
- Lo escaso es mejor que lo denso
- La legibilidad cuenta
- La practicidad vence a la pureza
- Los errores nunca deben pasar en silencio
- Frente a la ambigüedad, rechace la tentación de adivinar
- Una forma obvia de hacerlo
- Ahora es mejor que nunca
- Difícil de explicar, fácil de explicar
- Los espacios de nombres son una gran idea

## Elementos del Lenguaje

Como en la mayoría de los lenguajes de programación de alto nivel, Python se compone de una serie de elementos que definen su estructura. Entre ellos, podremos encontrar los siguientes:

### Variables

Una variable es un espacio en la memoria de un ordenador dónde almacenar datos modificables. En Python, una variable se define con la sintaxis:

```
nombre_de_la_variable = valor_de_la_variable
```

Cada variable, tiene un nombre y un valor, el cual define a la vez, el tipo de datos de la variable. Existe un tipo de “variable”, denominada constante, que se utiliza para definir valores fijos, que no requieran ser modificados.

> **PEP8: variables**
>
> Utilizar nombres descriptivos y en minúsculas. Para nombres compuestos, separar las palabras por guiones bajos. Antes y después del signo =, debe haber uno (y solo un) espacio en blanco

```
# Correcto
mi_variable = 12
# Incorrecto
MiVariable = 12
mivariable = 12
mi_variable=12
mi_variable  =  12
```

> **PEP8: constantes**
>
> Utilizar nombres descriptivos y en mayúsculas separando palabras por guiones bajos.

```
# Ejemplo
MI_CONSTANTE = 12
```

## Semántica del lenguaje

Python usa espacios en blanco (tabuladores o espacios) para estructurar el código en lugar de usar llaves como ocurre en muchos otros lenguajes (R, C ++, Java y Perl...).

> **PEP8: identación**
>
> Una identación de **4 (cuatro) espacios en blanco**, indicará que las instrucciones identadas, forman parte de una misma estructura de control.

```
inicio de la estructura de control:
    expresiones
    expresiones
    expresiones
```

### Comentarios

Los comentarios pueden ser de dos tipos: de una sola línea o multi-línea y se expresan de la siguiente manera:

```
# Esto es un comentario de una sola línea
mi_variable = 15
"""Y este es
   un comentario
   de varias
   líneas"""
mi_variable = 15
mi_variable = 15 # Este es un comentario en línea
```

> **PEP8: comentarios**
>
> Comentarios en la misma línea del código deben separarse con dos espacios en blanco. Luego del símbolo `#` debe ir un solo espacio en blanco.

# Estructuras de Control de Flujo

Python, como casi todos los lenguajes de programación, tiene diferentes componentes para definir lógica condicional, bucles y otros conceptos de flujo de control estándar.

## if, elif y else

La instrucción `if` es uno de los tipos de instrucciones de flujo de control más conocidos. Comprueba una condición que, si es cierta (se evalúa a `True`), continuará la ejecución con el código del bloque. Una instrucción `if` puede ir seguida opcionalmente por uno o más bloques `elif` y un bloque final `else` que se sólo evalúa si todas las condiciones anteriores han sido falsas (evaluadas a `False`). Si alguna de las condiciones se evalúa a `True`, no se alcanzarán los bloques `elif` o `else` posteriores.

```
if x < 0:
   print('Es negativo')
elif x == 0:
   print('Igual a zero')
elif 0 < x < 5:
   print('Positivo pero menor que 5')
else:
   print('Positivo y mayor o igual a 5')
```

## Bucle for

Los bucles `for` permiten iterar sobre una colección (como una *lista* o *tupla*) o un *iterador*. La sintaxis estándar para un bucle `for` es:

```
for valor in coleccion:
    # haz algo con 'valor'
    pass
```

Por ejemplo:

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
for nombre in mi_lista:
    print(nombre, len(nombre))
```

## Bucle while

Un bucle `while` especifica una condición y un bloque de código que se ejecutará hasta que la condición se evalúe como `False`.

```
x = 256
total = 0
while x > 0:
    total += x
    x = x - 1
```

Los bucles permiten el uso de `break` para forzar la salida de un bucle y `continue` para

#### Control de bucles, `break`, `continue` y `pass`

Los bucles `for` y `while` pueden ser interrumpidos con la sentencia `break`: cuando se ejecuta, el programa sale del bucle y continúa ejecutando el resto del código.

```
In [270]: for n in range(10000):
              print(n)
              if n == 3:
                  break

0
1
2
```

```
3
```

La sentencia `continue` , cuando se ejecuta, obliga a Python a dejar de ejecutar el código que haya dentro del bucle y a iniciar una nueva iteración (es decir, a volver al comienzo del bucle y seguir con la ejecución del programa).

```
In [271]: for n in range(5):
              if n == 2:
                  continue
              print(n)
```

```
0
1
3
4
```

La sentencia `pass` no hace nada. Puede utilizarse cuando se requiere una declaración sintácticamente, pero el programa no requiere ninguna acción.

```
while True:
   pass  # en espera de la interrupción del teclado (Ctrl+C)
```

Otro uso habitual es como un marcador de posición para una función o cuerpo condicional cuando está trabajando en un nuevo código, lo que le permite seguir pensando a un nivel más abstracto. El pase es ignorado silenciosamente

```
def initlog(*args):
    pass   # ¡Recuerde implementar esto!
```

# Tipos de datos

## Tipos numéricos (int, float y complex)

Hay tres tipos numéricos distintos: **enteros**, **números en coma flotante** y **números complejos**. Además, los **booleanos** son un subtipo de enteros. Los enteros tienen una precisión ilimitada. Los números de punto flotante se implementan usualmente utilizando el `double` en C. Los números complejos tienen una parte real e imaginaria, que son cada uno un número de punto flotante. Para extraer estas partes de un número complejo `z`, use `z.real` y `z.imag`. La biblioteca estándar incluye tipos numéricos adicionales, `fraction` que contienen números racionales y `decimal` que contienen números en coma flotante con una precisión definible por el usuario.

```
edad = 35  # Número entero (int)
edad = 043  # Número entero octal (int)
edad = 0x23  # Número entero hexadecimal (int)
precio = 7435.28  # Número real (float)
resultado = 4+3j  #  Número complejo (complex)
verdadero = True
falso = False
```

## Operadores Aritméticos

Entre los operadores aritméticos que Python utiliza, podemos encontrar los siguientes:

| Símbolo | Significado |
|---|---|
| + | Suma |
| - | Resta |
| * | Multiplicación |
| ** | Exponente |
| / | División |
| // | División entera |
| % | Módulo |

Python es totalmente compatible con aritmética mixta: cuando un operador de aritmética binaria tiene operandos de diferentes tipos numéricos, el operando con el tipo "más estrecho" se amplía al de los otros. Los constructores `int()`, `float()` y `complex()` se pueden usar para producir números de un tipo específico.

## Operadores Relacionales

Entre los operadores relacionales que Python utiliza, podemos encontrar los siguientes:

| Símbolo | Significado |
|---|---|
| == | igual que |
| != | distinto que |
| <, <= | menor, menor o igual |
| >, >= | mayor, mayor o igual |
| is | identidad de objeto |
| is not | identidad de objeto negada |

## Operadores Lógicos

Entre los operadores lógicos que Python utiliza, podemos encontrar los siguientes:

| Símbolo | Significado |
|---|---|
| and | Y lógica |

| Símbolo | Significado |
|---|---|
| or | O lógica |
| xor | O exclusiva |
| not | negación |

## Operadores a nivel de bit

Entre los operadores a nivel de bit que Python utiliza, podemos encontrar los siguientes:

| Símbolo | Significado |
|---|---|
| & | Y lógica |
| \| | O lógica |
| ^ | O exclusiva |
| ~ | negación |

> **PEP8: operadores**
>
> Siempre colocar un espacio en blanco, antes y después de un operador

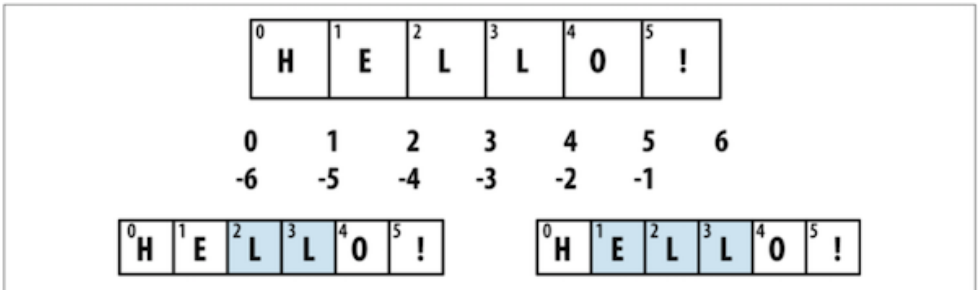
## Tipos secuencia (str, tuple, range, list)

Hay cuatro tipos de secuencia: cadenas, tuplas, rangos y listas. Existen tipos **inmutables** (su contenido no puede cambiar) y tipos **mutables** (su contenido puede variar).

#### Operaciones comunes en tipos secuencia

Las operaciones en la siguiente tabla son compatibles con la mayoría de los tipos de secuencia, tanto mutables como inmutables. En la tabla, `s y t` son secuencias del mismo tipo, `n , i , j y k` son enteros y `x` es un objeto arbitrario que cumple con las restricciones de tipo y valor impuestas por `s` .

| Operación | Significado |
|---|---|
| x in s | `True` si un item de `s` es igual a `x` , sino `False` |
| x not in s | `False` si un item de `s` es igual a `x` , sino `True` |
| s + t | concatenación de `s y t` |
| s * n, n * s | equivalente a repetir `s , n` veces |
| len(s) | longitud de `s` |
| min(s) | el item menor de `s` |
| max(s) | el item mayor de `s` |
| s.index(x[, i[, j]]) | índice de la primera ocurrencia de `x` en `s` (entre el índice `i` y antes del índice `j` ) |
| s.count(x) | número de ocurrencias de `x` en `s` |
| s[i] | iésimo item de `s` , con origen en 0 |
| s[i:j] | porción de `s` desde `i` hasta la posición anterior a `j` |
| s[i:j:k] | porción de `s` desde `i` hasta la posición anterior a `j` , con paso `k` |




**Operaciones comunes en tipos secuencia mutables**

En la tabla `s` es una instancia de un tipo de secuencia mutable, `t` es cualquier objeto iterable y `x` es un objeto arbitrario que cumple con cualquier tipo y restricciones de valor impuestas por `s`.

| Operación | Significado |
|---|---|
| s[i] = x | el item en posición `i` de `s` es reemplazado por `x` |
| s[i:j] = t | la porción de `s` desde `i` a `j` es reemplazada por los contenidos de `t` |
| del s[i:j] | elimina los elementos de `s` desde `i` a `j` |
| s[i:j:k] = t | la porción de `s[i:j:k]` es reemplazada por los contenidos de `t` |
| del s[i:j:k] | elimina los elementos `s[i:j:k]` de la secuencia |
| s.append(x) | añade `x` al final de la secuencia |
| s.clear() | elimina todos los items de `s` |
| s.copy() | crea una copia de `s` |
| s.extend(t) or s += t | extiende `s` con los contenidos de `t` |
| s *= n | actualiza `s` con su contenido repetido `n` veces |
| s.insert(i, x) | inserta `x` en `s` en la posición indicada por `i` |
| s.pop([i]) | recupera el item de la posición `i` y también lo elimina de `s` |
| s.remove(x) | elimina el primer item de `s` dónde `s[i]` sea igual a `x` |
| s.reverse() | invierte los items de `s` |

## Textos (str)

Los datos de tipo texto en Python se manejan con objetos `str`, o *cadenas*. Las cadenas son secuencias **inmutables** de caracteres Unicode. Los literales de tipo cadena de caracteres pueden estar escritos de varias maneras:

- Comillas simples: `'permite comillas "dobles" incrustadas'`
- Comillas dobles: `"permite las comillas 'simples' incrustadas"`
- Comillas triples: `'''Tres comillas simples''', """ Tres comillas dobles """`
- Utilizando el constructor `str(object='obj'[, encoding='utf-8', errors='strict'])`

Las cadenas entre comillas triples pueden abarcar varias líneas: todos los espacios en blanco asociados se incluirán en el literal de la cadena. Las cadenas implementan todas las operaciones de secuencia comunes, junto con los métodos adicionales habituales en múltiples lenguajes.

```
In [1]: a = 'cadena'

In [2]: a.<Press Tab>
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

Desde Python 3.0, las cadenas se almacenan como Unicode, es decir, cada carácter de la cadena está representado por un punto de código (cualquiera de los valores numéricos que conforman el espacio de código). De modo que una cada cadena es solo una secuencia de puntos de código Unicode. Para un almacenamiento eficiente de estas cadenas, la secuencia de puntos de código se convierte en un conjunto de bytes. El proceso se conoce como codificación.

Hay varias codificaciones presentes que tratan una cadena de manera diferente. Las codificaciones populares son `utf-8`, `ascii`, etc. Usando el método `encode()` de la cadena, se pueden convertir cadenas sin codificar en

cualquier codificación compatible con Python. Por defecto, Python usa la codificación utf-8. La sintaxis del método `encode()` es:

```
encode(encoding = 'UTF-8', errors = 'strict')
```

De igual forma para decodificar cadenas está el métdodo `decode()`, cuya síntaxis es:

In [1]:
```
# cadena Unicode
cadena = 'pythön!'
print('La cadena es:', cadena)

# codificación por defecto a utf-8
print('La versión codificada es:', cadena.encode())
# ignore error
print('La versión codificada (con ignore) es:', cadena.encode("ascii", "ignor
e"))
# replace error
print('La versión codificada (con replace) es:', cadena.encode("ascii", "replac
e"))
```

```
La cadena es: pythön!
La versión codificada es: b'pyth\xc3\xb6n!'
La versión codificada (con ignore) es: b'pythn!'
La versión codificada (con replace) es: b'pyth?n!'
```

In [2]:
```
cadena = "Esto es una cadena de ejemplo...";
cadena = cadena.encode('utf_32');

# codificación en utf_32
print('La versión codificada es:', cadena)
# decodificación en utf_32
print('La versión decodificada es:', cadena.decode('utf_32','strict'))
```

```
La versión codificada es: b'\xff\xfe\x00\x00E\x00\x00\x00s\x00\x00\x00t\x00\x0
0\x00o\x00\x00\x00 \x00\x00\x00e\x00\x00\x00s\x00\x00\x00 \x00\x00\x00u\x00\x0
0\x00n\x00\x00\x00a\x00\x00\x00 \x00\x00\x00c\x00\x00\x00a\x00\x00\x00d\x00\x0
0\x00e\x00\x00\x00n\x00\x00\x00a\x00\x00\x00 \x00\x00\x00d\x00\x00\x00e\x00\x0
0\x00 \x00\x00\x00e\x00\x00\x00j\x00\x00\x00e\x00\x00\x00m\x00\x00\x00p\x00\x0
0\x00l\x00\x00\x00o\x00\x00\x00.\x00\x00\x00.\x00\x00\x00.\x00\x00\x00'
La versión decodificada es: Esto es una cadena de ejemplo...
```

## Tuplas (tuple)

Las tuplas son secuencias **inmutables**, que normalmente se utilizan para almacenar colecciones de datos heterogéneos. Las tuplas también se utilizan para casos en los que se necesita una secuencia inmutable de datos homogéneos. Las tuplas se pueden construir de varias maneras:

- Usando un par de paréntesis para denotar la tupla vacía: `()`
- Usando una coma para definir una tupla singleton: `a,` o `(a,)`
- Separando elementos con comas: `a, b, c` o `(a, b, c)`
- Usando el constructor `tuple ()` o `tuple (iterable)`

> Un `iterable` es cualquier secuencia, contenedor u objeto que permite iterar por sus componentes. Es la coma la que forma una tupla, no los paréntesis. Los paréntesis son opcionales, excepto en el caso de la tupla vacía, o cuando son necesarios para evitar la ambigüedad sintáctica.

In [3]:
```
una_tupla = 4, 5, 6, 7
una_tupla
```

Out[3]:
```
(4, 5, 6, 7)
```

In [4]:
```
tupla_anidada = (4, 5, 6), (7, 8)
tupla_anidada
```

Out[4]:
```
((4, 5, 6), (7, 8))
```

In [5]:
```
otra_tupla = tuple('cadena')
otra_tupla
```

Out[5]:
```
('c', 'a', 'd', 'e', 'n', 'a')
```

In [6]:
```
# Acceso a los elementos, las secuencias comienzan en el índice de 0 en Python
otra_tupla[0]
```

Out[6]:
```
'c'
```

In [7]:
```
# Una vez creada la tupla, no es posible modificar los objetos que se almacenan
en la misma:
otra_tupla[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-a97109b690f7> in <module>
      1 # Una vez creada la tupla, no es posible modificar los objetos que se
almacenan en la misma:
----> 2 otra_tupla[0] = 'x'

TypeError: 'tuple' object does not support item assignment
```

In [8]:
```
# Se pueden concatenar tuplas utilizando el operador + para producir tuplas más
largas
(4, None, 'foo') + (6, 0) + ('bar',)
```

Out[8]:
```
(4, None, 'foo', 6, 0, 'bar')
```

In [9]:
```
# Multiplicar una tupla por un número entero, tiene el efecto de concatenar jun
tas tantas copias de la tupla:
('foo', 'bar') * 4
```

Out[9]:
```
('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

#### Desempaquetando tuplas

Python permite asignar elementos de una tupla a variables:

In [10]:
```
una_tupla = (4, 5, 6)
a, b, c = una_tupla
print(a)
print(b)
print(c)
```

```
4
5
6
```

In [11]:
```
# Incluso secuencias con tuplas anidadas pueden ser desempaquetadas
una_tupla = 4, 5, (6, 7)
a, b, (c, d) = una_tupla
c
```

Out[11]:
```
6
```

In [12]:
```
# La asignación múltiple en Python permite intercambiar fácilmente los valores entre variables
a, b = 1, 7
print('a={0} y b={1}'.format(a, b))
```

```
a=1 y b=7
```

In [13]:
```
(b, a) = (a, b)
print('a={0} y b={1}'.format(a, b))
```

```
a=7 y b=1
```

In [14]:
```
# Un uso común del desempaquetado de variables es iterar sobre secuencias de tuplas o listas:
secuencia = (1,2,3),(4,5,6),(7,8,9)
for a, b, c in secuencia:
    print('a={0}, b={1}, c={2}'.format(a, b, c))
```

```
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

En las nuevas versiones de Python se puede utilizar un desempaquetado avanzado utilizando la sintaxis especial `*rest` que captura los resto de la tupla en una variable. Esta sintaxis también se usa en funciones para capturar una lista arbitrariamente larga de argumentos posicionales:

In [15]:
```
valores = 1, 2, 3, 4, 5
a, b, *resto = valores
```

In [16]:
```
a, b
```

Out[16]:
```
(1, 2)
```

In [17]:
```
resto
```

Out[17]:
```
[3, 4, 5]
```

La variable `rest` puede tener cualquier nombre válido, cuando hace referencia a variables no deseadas por convención se usa el guión bajo (_):

In [18]:
```
a, b, *_ = valores
_
```

Out[18]:
```
[3, 4, 5]
```

#### Métodos para tuplas

Dado que el tamaño y el contenido de una tupla no se pueden modificar, no tiene muchos métodos de instancia. Uno particularmente útil (también disponible en listas) es `count()`, que cuenta el número de ocurrencias de un valor:

In [19]:
```
a = (1, 2, 2, 2, 3, 4, 2)
a.count(2)
```

Out[19]:
```
4
```

## Rangos (range)

El tipo de rango representa una secuencia **inmutable** de números y se usa comúnmente para hacer un bucle un número específico de veces o para recorrer componentes iterables. Los rangos se utilizan por optimizacion de memoria, si bien un rango generado puede ser arbitrariamente grande, su uso de memoria en un momento dado suele ser muy pequeño.

```
range(stop)
range(start, stop[, step])
```

Los argumentos para el constructor de rangos deben ser enteros. Si se omite el argumento de `step` , el valor predeterminado es 1. Si se omite el argumento `start` el valor predeterminado es 0. Los rangos soportan índices negativos.

In [20]:
```
rango = range(10)
rango
```

Out[20]:
```
range(0, 10)
```

In [21]:
```
for valor in rango:
    print(valor, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9
```

In [22]:
```
rango = range(5, 0, -1)
for valor in rango:
    print(valor, end=" ")
```

```
5 4 3 2 1
```

In [23]:
```
rango = range(-1, -10, -1)
for valor in rango:
    print(valor, end=" ")
```

```
-1 -2 -3 -4 -5 -6 -7 -8 -9
```

In [24]:
```
-15 in rango
```

Out[24]:
```
False
```

In [25]:
```
rango.index(-7)
```

Out[25]:
```
6
```

In [26]:
```
rango[0:3]
```

Out[26]:
```
range(-1, -4, -1)
```

In [27]:
```
rango[-1]
```

Out[27]:
```
-9
```

In [28]:
```
range(0)
```

Out[28]:
```
range(0, 0)
```

```
In [29]: tupla = ('c', 'a', 'd', 'e', 'n', 'a')
         for i in range(len(tupla)):
             print('{0} -> {1}'.format(i, tupla[i]))
```

```
0 -> c
1 -> a
2 -> d
3 -> e
4 -> n
5 -> a
```

**Nota:** La compración de igualdad de rangos con `==` y `!=` se realiza a nivel de secuencias. Es decir, dos objetos de rango se consideran iguales si presentan la misma secuencia de valores.

## Listas (list)

Las listas son secuencias **mutables**, que **generalmente** se utilizan para almacenar **colecciones de elementos homogéneos** (donde el grado de similitud variará según la aplicación). Las listas se pueden construir de varias maneras:

- Usando un par de corchetes para denotar la lista vacía: `[]`
- Usando corchetes, separando elementos con comas: `[a]`, `[a, b, c]`
- Usando una expresión de comprensión: `[x para x en iterable]`
- Usando el constructor `list()` o `list(iterable)`

```
In [30]: lista_vacia = list()
         lista_vacia
```

```
Out[30]: []
```

```
In [31]: una_lista = [2, 3, 7, None]
         una_lista
```

```
Out[31]: [2, 3, 7, None]
```

```
In [32]: una_lista = list(range(10))
         una_lista
```

```
Out[32]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [33]: una_lista = [True, "2", 3.0, 4]
         una_lista
```

```
Out[33]: [True, '2', 3.0, 4]
```

```
In [34]: tupla = ('foo', 'bar', 'baz')
         una_lista = list(tupla)
         una_lista
```

```
Out[34]: ['foo', 'bar', 'baz']
```

```
In [35]: una_lista[1] ='peekaboo'
         una_lista
```

```
Out[35]: ['foo', 'peekaboo', 'baz']
```

```
In [36]: una_lista.append('dwarf')
         una_lista
```

```
Out[36]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

```
In [37]: una_lista.insert(3, 'root')
         una_lista
Out[37]: ['foo', 'peekaboo', 'baz', 'root', 'dwarf']

In [38]: una_lista.pop(2)
Out[38]: 'baz'

In [39]: una_lista
Out[39]: ['foo', 'peekaboo', 'root', 'dwarf']

In [40]: una_lista.remove('root')
         una_lista
Out[40]: ['foo', 'peekaboo', 'dwarf']

In [41]: 'bar' in una_lista
Out[41]: False

In [42]: 'root' not in una_lista
Out[42]: True

In [43]: # El operador + concatena listas, pero es más rápido utilizar el método extend
         una_lista.extend(['root', 'pie', 'body'])
         una_lista
Out[43]: ['foo', 'peekaboo', 'dwarf', 'root', 'pie', 'body']
```

#### Métodos para listas

Las listas también proporcionan el siguiente método adicional `sort (*, key = None, reverse = False)` que ordena la lista en su lugar, utilizando solo comparaciones con `<` entre elementos. El parámetro `key` especifica una función de un argumento que se usa para extraer una clave de comparación de cada elemento de la listay el parámetro `reverse` si se establece en `True` , los elementos de la lista se ordenan como si se hubiera invertido cada comparación.

```
In [44]: otra_lista = [7, 2, 5, 1, 3]

In [45]: otra_lista.sort()
         otra_lista
Out[45]: [1, 2, 3, 5, 7]

In [46]: otra_lista = ['saw', 'small', 'He', 'foxes', 'six']
         otra_lista.sort(key=len)
         otra_lista
Out[46]: ['He', 'saw', 'six', 'small', 'foxes']
```

## Métodos para secuencias

### enumerate

Cuando se realiza una iteración sobre una secuencia es común realizar un seguimiento del índice del elemento actual. Un enfoque habitual se vería así:

```
i = 0
for valor in coleccion:
    # hacer algo con valor
    i += 1
```

Python tiene una función incorporada, `enumerate` , que devuelve una secuencia de tuplas (i, valor):

```
for i, valor in enumerate(coleccion):
    # hacer algo con valor
```

In [47]:
```
una_lista = ['foo', 'bar', 'baz']
for i, v in enumerate(una_lista):
    print(i, " -> ", v)
```

```
0  ->  foo
1  ->  bar
2  ->  baz
```

### sorted

La función `sorted` devuelve una nueva lista ordenada de los elementos de cualquier secuencia:

In [48]:
```
mi_lista = [7, 1, 2, 6, 0, 3, 2]
sorted(mi_lista)
```

Out[48]:
```
[0, 1, 2, 2, 3, 6, 7]
```

In [49]:
```
mi_lista
```

Out[49]:
```
[7, 1, 2, 6, 0, 3, 2]
```

In [50]:
```
sorted('horse race')
```

Out[50]:
```
[' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

### zip

Esta función "empareja" los elementos de una serie de listas, tuplas u otras secuencias para crear una lista de tuplas:

In [51]:
```
secuencia_1 = [100, 101, 102]
secuencia_2 = ['foo', 'bar', 'baz']
zipped = zip(secuencia_1, secuencia_2)
zipped
```

Out[51]:
```
<zip at 0x7fc1ade2e3c0>
```

In [52]:
```
list(zipped)
```

Out[52]:
```
[(100, 'foo'), (101, 'bar'), (102, 'baz')]
```

In [53]:
```
# zip puede tomar un número arbitrario de secuencias,
# y el número de elementos que produce está determinado por la secuencia más co
rta:
secuencia_3 = [False, True]
list(zip(secuencia_1, secuencia_2, secuencia_3))
```

Out[53]:
```
[(100, 'foo', False), (101, 'bar', True)]
```

In [54]:
```
# Un uso muy común de zip es iterar simultáneamente en múltiples secuencias,
# posiblemente también combinado con enumerate:
for i, (a, b) in enumerate(zip(secuencia_1, secuencia_2)):
    print("{0}: {1}, {2}".format(i, a, b))
```

```
0: 100, foo
1: 101, bar
2: 102, baz
```

In [55]:
```
# Dada una secuencia "comprimida", zip se puede aplicar de una manera inteligen
te para "descomprimir"
# la secuencia. Otra forma de pensar acerca de esto es convertir una lista de f
ilas en una lista de columnas:
jugadores = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
             ('Schilling', 'Curt', 'Smith')]
nombres, apellidos = zip(*jugadores)
nombres
```

Out[55]:
```
('Nolan', 'Roger', 'Schilling')
```

In [56]:
```
apellidos
```

Out[56]:
```
('Ryan', 'Clemens', 'Curt')
```

**`reversed`**

Itera sobre los elementos de una secuencia en orden inverso:

In [57]:
```
list(reversed(range(10)))
```

Out[57]:
```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

## Tipos conjunto (set, frozenset)

Un objeto conjunto es una colección **desordenada de objetos distintos** susceptibles de tener un valor de tipo `hash`. Los usos comunes incluyen la prueba de membresía, la eliminación de duplicados de una secuencia y el cálculo de operaciones matemáticas como intersección, unión, diferencia y diferencia simétrica. Al ser una colección desordenada, los conjuntos no registran la posición del elemento ni el orden de inserción. **En consecuencia, los conjuntos no son compatibles con la indexación, la segmentación u otro comportamiento similar a una secuencia**.

Las conjuntos se pueden construir de varias maneras:

- Usando un par de llaves para denotar el conjunto vacío: `{}`
- Usando llaves, separando elementos con comas: `{a}`, `{a, b, c}`
- Usando una expresión de comprensión: `{x para x en iterable}`
- Usando el constructor `set()` o `set(iterable)`

Actualmente hay dos tipos de conjuntos incorporados, `set` y `frozenset`. El tipo `set` es **mutable**: el contenido se puede cambiar utilizando métodos como `add()` y `remove()`. Dado que es mutable, no tiene valor `hash` y no se puede utilizar como clave de diccionario ni como elemento de otro conjunto. El tipo `frozenset` es **inmutable** y *hashable*: su contenido no puede alterarse después de su creación; por lo tanto, se puede utilizar como una clave de diccionario o como un elemento de otro conjunto.

Los tipos conjunto admiten operaciones de conjuntos matemáticos como unión, intersección, diferencia y diferencia simétrica. Las operaciones que suponen actualización de los conjuntos sólo se pueden aplicar a conjuntos mutables de tipo `set`. La siguiente tabla recoge las principales funciones aplicables a conjuntos:

| Función | Sintaxis alt. | Descripción |
|---|---|---|
| a.add(x) | --- | Añade el elmento `x` al conjunto `a` |
| a.clear() | --- | Restaure el conjunto `a` un estado vacío, descartando todos sus elementos |
| a.copy() | --- | Crea una copia del conjunto `a` |
| a.remove(x) | --- | Elimina el elemento `x` del conjunto `a`, generando un error si el `x` no está en `a` |
| a.discard(x) | --- | Elimina el elemento `x` del conjunto `a` |
| a.pop() | --- | Elimina un elemento arbitrario del conjunto `a`, generando un error si el conjunto está vacío |
| a.union(b) | a \| b | Define un conjunto con todos los elementos diferenciados de `a` y `b` |
| a.update(b) | a \|= b | Actualiza `a` con todos los elementos diferenciados de `a` y `b` |
| a.intersection(b) | a & b | Define un conjunto con todos los elementos que están en `a` y `b`, en ambos conjuntos |
| a.intersection_update(b) | a &= b | Actualiza `a` con todos los elementos que están en `a` y `b`, en ambos conjuntos |
| a.difference(b) | a - b | Define un conjunto con los elementos que están en `a` y no en `b` |
| a.difference_update(b) | a -= b | Actualiza `a` con todos los elementos que están en `a` y no en `b` |
| a.symmetric_difference(b) | a ^ b | Define un conjunto con los elementos que están en `a` o en `b`, pero no en ambos |
| a.symmetric_difference_update(b) | a ^= b | Actualiza `a` con los elementos que están en `a` o en `b`, pero no en ambos |
| a.issubset(b) | --- | `True` si todos los elementos de `a` están contenidos en `b` |
| a.issuperset(b) | --- | `True` si todos los elementos de `b` están contenidos en `a` |
| a.isdisjoint(b) | --- | `True` si `a` y `b` no tienen elementos en común |
| a.update(*otros) | --- | Actualiza `a` añadiendo todos los elementos presentes enlos cojuntos definidos por `otros` |

In [58]:
```
a = {2, 2, 2, 1, 3, 3}
a
```

Out[58]:
```
{1, 2, 3}
```

In [59]:
```
a = set([2, 2, 2, 1, 3, 3])
a
```

Out[59]:
```
{1, 2, 3}
```

In [60]:
```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
a.union(b)
```

Out[60]:
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [61]:
```
a.intersection(b)
```

Out[61]:
```
{3, 4, 5}
```

```
In [62]: a.difference(b)
Out[62]: {1, 2}

In [63]: a.symmetric_difference(b)
Out[63]: {1, 2, 6, 7, 8}
```

## Tipos mapa (dict)

Un objeto de tipo mapa permite asignar valores `hash` a objetos arbitrarios. Los mapas son objetos **mutables**. Actualmente solo hay un tipo de mapa estándar, el *diccionario*. El diccionario es probablemente la estructura de datos incorporada en Python más importante. Un nombre más común para los diccionarios es *mapa hash* o *array asociativo*. Es una colección de pares de clave-valor de tamaño flexible, donde la clave y el valor son objetos de Python.

Las claves de un diccionario son valores *normalmente* arbitrarios. los valores que no pueden tener una valor `hash` asignado no se pueden usar como claves, es decir, los valores que contienen listas, diccionarios u otros tipos mutables (que se comparan por valor en lugar de por identidad de objeto) no pueden ser claves. Los tipos numéricos utilizados como claves obedecen a las reglas normales de comparación numérica: si dos números se comparan de la misma manera (como 1 y 1.0), se pueden usar indistintamente para indexar la misma entrada del diccionario.

Las diccionarios se pueden construir de varias maneras:

- Usando pares clave-valor separados por comas entre llaves: `{c:v, c:v, ...}`
- Usando una expresión de comprensión: `{c:v para c, v en iterable}`
- Usando el constructor `dict ()`

```
In [64]: dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
Out[64]: {'sape': 4139, 'guido': 4127, 'jack': 4098}

In [65]: dict(sape=4139, guido=4127, jack=4098)
Out[65]: {'sape': 4139, 'guido': 4127, 'jack': 4098}

In [66]: un_diccionario = {'a' : 'un valor', 'b' : [1, 2, 3, 4]}
         un_diccionario
Out[66]: {'a': 'un valor', 'b': [1, 2, 3, 4]}

In [67]: # Se puede acceder, insertar o establecer elementos utilizando la misma sintaxi
         s
         # que para acceder a los elementos de una lista o tupla:
         un_diccionario[7] = 'un entero'
         un_diccionario
Out[67]: {'a': 'un valor', 'b': [1, 2, 3, 4], 7: 'un entero'}

In [68]: un_diccionario['b']
Out[68]: [1, 2, 3, 4]

In [69]: # Se puede verificar si un diccionario contiene una clave usando la misma sinta
         xis
         # que se usa para verificar si una lista o tupla contiene un valor:
         'b' in un_diccionario
Out[69]: True
```

In [70]:
```
# Se puede eliminar valores utilizando la función del o el método pop
un_diccionario[5] = 'un valor'
un_diccionario['dummy'] = 'otro valor'
un_diccionario
```

Out[70]:
```
{'a': 'un valor',
 'b': [1, 2, 3, 4],
 7: 'un entero',
 5: 'un valor',
 'dummy': 'otro valor'}
```

In [71]:
```
del un_diccionario[5]
un_diccionario
```

Out[71]:
```
{'a': 'un valor', 'b': [1, 2, 3, 4], 7: 'un entero', 'dummy': 'otro valor'}
```

In [72]:
```
valor = un_diccionario.pop('dummy')
valor
```

Out[72]:
```
'otro valor'
```

In [73]:
```
un_diccionario
```

Out[73]:
```
{'a': 'un valor', 'b': [1, 2, 3, 4], 7: 'un entero'}
```

#### Sobre las claves de los diccionarios

Si bien los valores de un diccionario pueden ser cualquier objeto Python, las claves generalmente tienen que ser objetos inmutables como tipos escalares (int, float, string) o tuplas (todos los objetos en la tupla también deben ser inmutables). El término técnico es `hashability` . Se puede verificar si un objeto es `hashable` (se puede usar como una clave en un diccionario) con la función `hash` :

In [74]:
```
hash('cadena')
```

Out[74]:
```
-7750850285370336504
```

In [75]:
```
hash((1, 2, (2, 3)))
```

Out[75]:
```
-9209053662355515447
```

In [76]:
```
hash((1, 2, [2, 3])) # falla porque las listas son mutables
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-a5a61418fc8e> in <module>
----> 1 hash((1, 2, [2, 3])) # falla porque las listas son mutables

TypeError: unhashable type: 'list'
```

In [77]:
```
# Para unsar una lista como clave, una opción es convertirla en una tupla
otro_diccionario = {}
otro_diccionario[tuple([1, 2, 3])] = 5
otro_diccionario
```

Out[77]:
```
{(1, 2, 3): 5}
```

#### Creando diccionarios a partir de secuencias

Una de las operativas más habituales con diccionarios es emparejar dos secuencias de forma inteligente. Una primera aproximación podría ser:

```
mapa = {}
for clave, valor in zip (lista_de_claves, lista_de_valores):
    mapa [clave] = valor
```

Dado que un diccionario es esencialmente una colección de 2-tuplas, la función `dict` acepta una lista de 2-tuplas:

In [78]:
```
palabras = ['apple', 'bat', 'bar', 'atom', 'book']
mapa = dict(zip(range(len(palabras)), palabras))
mapa
```

Out[78]:
```
{0: 'apple', 1: 'bat', 2: 'bar', 3: 'atom', 4: 'book'}
```

#### Métodos para Diccionarios

**Claves y valores**. Los métodos `keys()` y `values()` proporcionan iteradores de las claves y valores del diccionario, respectivamente. Si bien los pares clave-valor no están en ningún orden en particular, estas funciones devuelven las claves y los valores en el mismo orden. También hay un método `items ()` que devuelve una lista de tuplas `(clave, valor)`, que es la forma más eficiente de examinar todos los datos de valores clave en el diccionario. Todas estas listas se pueden pasar a la función `sort()`:

In [79]:
```
list(un_diccionario.keys())
```

Out[79]:
```
['a', 'b', 7]
```

In [80]:
```
list(un_diccionario.values())
```

Out[80]:
```
['un valor', [1, 2, 3, 4], 'un entero']
```

In [81]:
```
list(un_diccionario.items())
```

Out[81]:
```
[('a', 'un valor'), ('b', [1, 2, 3, 4]), (7, 'un entero')]
```

**Actualización: `update`** El método `update` permite fusionar un diccionario con otro. El método cambia los diccionarios, por lo que cualquier clave existente en los datos que se pasan en la actualización descartarán sus valores antiguos.

In [82]:
```
un_diccionario.update({'b' : 'foo', 'c' : 12})
un_diccionario
```

Out[82]:
```
{'a': 'un valor', 'b': 'foo', 7: 'un entero', 'c': 12}
```

**Valores por defecto: `setdefault`** Es muy común seguir la lógica siguiente al trabajar con diccionarios:

```
if clave in diccionario:
    valor = diccionario[clave]
else:
    valor = valor_por_defecto
```

De este modo, los métodos `get` y `pop` pueden tomar un valor predeterminado para ser devuelto, de modo que el bloque `if-else` anterior puede escribirse simplemente como:

```
valor = diccionario.get (clave, valor_por_defecto)
```

El método `get` por defecto devolverá `None` si la clave no está presente, mientras que `pop` provocará una excepción.

Es habitual en el uso de diccionarios que los valores sean otras colecciones, como listas. Por ejemplo, podría imaginar categorizar una lista de palabras por sus primeras letras como un dictado de listas:

In [83]:
```
palabras = ['apple', 'bat', 'bar', 'atom', 'book']
por_letras = {}
for palabra in palabras:
    letra = palabra[0]
    if letra not in por_letras:
        por_letras[letra] = [palabra]
    else:
        por_letras[letra].append(palabra)
por_letras
```

Out[83]:
```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

El método `setdefault` se utiliza para este propósito. El bucle anterior podría escribirse como:

In [84]:
```
por_letras = {}
for palabra in palabras:
    por_letras.setdefault(palabra[0], []).append(palabra)
por_letras
```

Out[84]:
```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

## Colecciones avanzadas

Las siguientes colecciones, módulo de `collections` , son en su mayoría solo extensiones de colecciones base, algunas de ellas bastante simples y otras un poco más avanzadas. Sin embargo, para todos ellos es importante conocer las características de las estructuras subyacentes. Sin comprenderlos, será difícil comprender las características de estas colecciones. Hay algunas colecciones que se implementan en código C nativo por razones de rendimiento, pero todas ellas también se pueden implementar fácilmente en Python puro.

#### ChainMap - la lista de diccionarios

Introducido en Python 3.3, `ChainMap` le permite combinar múltiples asignaciones (diccionarios, por ejemplo) en una. Esto es especialmente útil cuando se combinan varios contextos. Es una clase similar a `dict` para crear una vista única de múltiples mapas, incorporados por referencia, de modo que si alguno se actualiza esos cambios se reflejarán en ChainMap.

In [85]:
```
from collections import ChainMap
baseline = {'music': 'bach', 'art': 'rembrandt'}
adjustments = {'art': 'van gogh', 'opera': 'carmen'}
combined = ChainMap(baseline, adjustments)
print(combined)
print(combined['art'])
```

```
ChainMap({'music': 'bach', 'art': 'rembrandt'}, {'art': 'van gogh', 'opera': '
carmen'})
rembrandt
```

#### Counter - seguimiento de los elementos más frecuentes

Un contador es una clase para realizar un seguimiento del número de apariciones de un elemento. La clase `Counter` es una subclase de `dict` para contar objetos hashables. Es una colección donde los elementos se almacenan como claves de diccionario y sus conteos como valores de diccionario. Se permite que los conteos sean cualquier valor entero, incluidos cero o valores negativos.

In [86]:
```
from collections import Counter
c = Counter(['eggs', 'ham', 'eggs'])
print(c['bacon'])
print(c['eggs'])
```

```
0
2
```

Los objetos `Counter` admiten tres métodos más allá de los disponibles para todos los diccionarios:

- **elements()**, que devuelve un iterador sobre los elementos que se repiten tantas veces como su conteo.

In [87]:
```
c = Counter(a=4, b=2, c=0, d=-2)
sorted(c.elements())
```

Out[87]:
```
['a', 'a', 'a', 'a', 'b', 'b']
```

- **most_common([n])**, que devuelve una lista de los `n` elementos mas comunes y sus conteos, del mas común al menos común.

In [88]:
```
Counter('abracadabra').most_common(3)
```

Out[88]:
```
[('a', 5), ('b', 2), ('r', 2)]
```

- **subtract([iterable-o-mapping])**, los elementos se restan de un iterable o de otro mapeo (o contador). Como dict.update() pero resta los conteos en lugar de reemplazarlos. Tanto las entradas como las salidas pueden ser cero o negativas.

In [89]:
```
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
c.subtract(d)
c
```

Out[89]:
```
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

**deque - la cola de dos extremos**

El objeto `deque` (abreviatura de Double Ended Queue - "cola de dos extremos") es una de las colecciones más antiguas, fue introducido en Python 2.4 y son una generalización de pilas y colas. Admiten hilos seguros, `appends` y `pops` eficientes en memoria desde cualquier lado con aproximadamente el mismo rendimiento `O(1)` en cualquier dirección.

Los objetos `deque` admiten, entre otros, los siguientes métodos: `append(x)`, `appendleft(x)`, `clear()`, `copy()`, `count(x)`, `extend(iterable)`, `extendleft(iterable)`, `index(x[, start[, stop]])`, `insert(i, x)`, `pop()`, `popleft()`, `remove(value)`, `reverse()` y `rotate(n=1)`. También proporcionan un atributo de solo lectura `maxlen` que informa de su tamaño máximo o `None` si no está limitado.

In [90]:
```
from collections import deque
d = deque('ghi')
d.append('j')
d.appendleft('f')
d
```

Out[90]:
```
deque(['f', 'g', 'h', 'i', 'j'])
```

**defaultdict - diccionario con un valor predeterminado**

La clase `defaultdict` es una subclase de `dict` que recibe como parámetro un tipo o una función para generar el valor predeterminado de cada entrada en el diccionario. En lugar de tener que verificar la existencia de una clave y agregar un valor cada vez, puede simplemente declarar el valor predeterminado desde el principio, y no hay necesidad de preocuparse por el resto.

In [91]:
```
from collections import defaultdict
palabras = ['apple', 'bat', 'bar', 'atom', 'book']
por_letras = defaultdict(list)
for palabra in palabras:
    por_letras[palabra[0]].append(palabra)
por_letras
```

Out[91]:
```
defaultdict(list, {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']})
```

Cuando se encuentra una clave por primera vez, al no estar en el mapping se crea automáticamente usando la función `list` que retorna una list vacía. La operación `list.append()` luego adjunta el valor a la nueva lista. Cuando se encuentra una clave existente, la búsqueda procede normalmente (retornando la lista para esa clave) y la operación `list.append()` agrega otro valor a la lista. Esta técnica es más simple y rápida que una técnica equivalente usando `dict.setdefault()`.

**OrderedDict: un diccionario donde importa el orden de inserción**

El objeto `OrderdDict` es un diccionario que realiza un seguimiento del orden en el que se insertaron los elementos. Mientras que un diccionario normal devolverá sus claves en el orden de hash, un `OrderedDict` devolverá sus claves por orden de inserción.

In [92]:
```
from collections import OrderedDict
spam = OrderedDict()
spam['b'] = 2
spam['c'] = 3
spam['a'] = 1
spam
```

Out[92]:
```
OrderedDict([('b', 2), ('c', 3), ('a', 1)])
```

#### namedtuple - tuplas con nombres de campo

El objeto `namedtuple` es exactamente lo que implica su nombre: una tupla con un nombre. Las tuplas con nombre asignan significado a cada posición en una tupla y permiten un código más legible y autodocumentado. Se pueden usar donde se usen tuplas regulares y agregan la capacidad de acceder a los campos por nombre en lugar del índice de posición.

In [93]:
```
from collections import namedtuple
Punto = namedtuple('Punto', ['x', 'y', 'z'])
punto_a = Punto(1, 2, 3)
print(punto_a)
print(punto_a.z)
```

```
Punto(x=1, y=2, z=3)
3
```

## Otras estructuras de datos de interés

Existen otros paquetes que proporcionan estructuras de datos interesantes que permiten, entre otras posibilidades, trabajar con tipos enumerados o con colas con prioridad o

#### enum - un grupo de constantes

El paquete `enum` es bastante similar a `namedtuple` pero tiene un objetivo y una interfaz completamente diferentes. El objeto básicode `enum` hace que sea realmente fácil tener constantes en sus módulos mientras evita los números mágicos. Una enumeración es un conjunto de nombres simbólicos (miembros) ligados a valores únicos y constantes. Dentro de una enumeración, los miembros pueden compararse por identidad y la enumeración en sí puede recorrerse.

In [94]:
```
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

print(Color.RED)
print(Color(1))
print(Color.RED.name)
print(Color.RED.value)
Color.RED
```

```
Color.RED
Color.RED
RED
1
```

Out[94]:
```
<Color.RED: 1>
```

#### heapq - lista con prioridad

El módulo `heapq` es un pequeño módulo que hace que sea muy fácil crear una cola de prioridad en Python (implementación del algoritmo de montículos). Una estructura que siempre hará que el elemento más pequeño (o más grande, según la implementación) esté disponible con el mínimo esfuerzo. Las `heapq` son árboles binarios para los que cada nodo padre tiene un valor menor o igual que cualquiera de sus hijos, sin embargo, esta implementación utiliza matrices, para las cuales `heap[k] <= heap[2*k+1] y heap[k] <= heap[2*k+2]` para todo `k` , contando los elementos desde cero. Para poder comparar, los elementos inexistentes se consideran infinitos. La propiedad interesante de un montículo es que su elemento más pequeño es siempre la raíz, `heap[0]` .

Para crear un `heapq` , use una lista inicializada en `[]` , o puede transformar una lista poblada en un `heapq` a través de la función `heapify()` . El API contine múltiples funciones para trabajar con las listas ordenadas.

In [95]:
```
from heapq import heappush, heappop
h = []
heappush(h, (5, 'write code'))
heappush(h, (7, 'release product'))
heappush(h, (1, 'write spec'))
heappush(h, (3, 'create tests'))
heappop(h)
```

Out[95]:
```
(1, 'write spec')
```

**bisect – la lista ordenada**

Este módulo brinda soporte para mantener una lista ordenada sin tener que reordenar la lista tras cada nueva inserción. Para listas largas de elementos que tienen operaciones de comparación costosas, será una mejora respecto a la estrategia más habitual. El módulo se llama `bisect` porque usa un algoritmo de bisección básico para lograr su objetivo.

Como es el caso de `heapq`, `bisect` no crea realmente una estructura de datos especial. Simplemente funciona en una lista estándar y espera que esa lista siempre esté ordenada. Es importante comprender las implicaciones de rendimiento de esto; simplemente agregar elementos a la lista usando el algoritmo bisect puede ser muy lento porque una inserción en una lista toma `O (n)`. Efectivamente, crear una lista ordenada usando bisect requiere `O (n * n)`, que es bastante lento, especialmente porque crear la misma lista ordenada usando `heapq` o `sorted` toma `O (n * log (n))` en su lugar.

In [96]:
```
# Using the regular sort:
sorted_list = []
sorted_list.append(5) # O(1)
sorted_list.append(3) # O(1)
sorted_list.append(1) # O(1)
sorted_list.append(2) # O(1)
sorted_list.sort() # O(n * log(n)) = O(4 * log(4)) = O(8)
sorted_list
[1, 2, 3, 5]
```

Out[96]:
```
[1, 2, 3, 5]
```

In [97]:
```
# Using bisect:
import bisect
sorted_list = []
bisect.insort(sorted_list, 5) # O(n) = O(1)
bisect.insort(sorted_list, 3) # O(n) = O(2)
bisect.insort(sorted_list, 1) # O(n) = O(3)
bisect.insort(sorted_list, 2) # O(n) = O(4)
sorted_list
```

Out[97]:
```
[1, 2, 3, 5]
```

Para una pequeña cantidad de elementos, la diferencia es insignificante, pero crece rápidamente hasta un punto en el que la diferencia será grande. Sin embargo, la búsqueda dentro de la lista es muy rápida; debido a que está ordenado, podemos usar un algoritmo de búsqueda binaria muy simple, de forma que nunca se necesitarán más de `O (log (n))` pasos para encontrar un número.

## Tipado dinámico

Una de las características de las variables en Python es que pueden tomar valores de distinto tipo a lo largo del código, propiedad que se conoce como **tipado dinámico**. Este tipo de flexibilidad es una pieza que hace que Python y otros lenguajes similares sean fáciles de usar. Comprender cómo funciona el tipado dinámico es una pieza importante de aprendizaje para analizar datos de manera eficiente y efectiva con Python. Pero lo que este tipo de flexibilidad también apunta es el hecho de que las variables de Python son más que solo su valor; también contienen información adicional sobre el tipo de valor.

> **Nota**: Siempre es posible recuperar el tipo de una variable haciendo uso de la función `type` :

In [98]:
```
variable = 3
print(type(variable))
```

```
<class 'int'>
```

In [99]:
```
variable = 3.7
print(type(variable))
```

```
<class 'float'>
```

In [100]:
```
variable = "3"
print(type(variable))
```

```
<class 'str'>
```

In [101]:
```
variable = True
print(type(variable))
```

```
<class 'bool'>
```

In [102]:
```
variable = (3, 5, 6)
print(type(variable))
```

```
<class 'tuple'>
```

In [103]:
```
variable = [3, 5, 6]
print(type(variable))
```

```
<class 'list'>
```

In [104]:
```
variable = {3, 5, 6}
print(type(variable))
```

```
<class 'set'>
```

In [105]:
```
variable = {'a':3, 'b':5, 'c':6}
print(type(variable))
```

```
<class 'dict'>
```

La implementación estándar de Python está escrita en C. Esto significa que cada objeto de Python es simplemente una estructura en C inteligentemente disfrazada, que contiene no solo su valor, sino también otra información. Mirando a través del código fuente de Python 3.4, encontramos que la definición de tipo entero (largo) efectivamente se ve así (una vez que se expanden las macros C):

```
struct _longobject {
    long ob_refcnt;          # recuento de referencias para manejo de memoria
    PyTypeObject *ob_type;   # tipo de variable
    size_t ob_size;          # tamaño de los datos
    long ob_digit[1];        # value entero actual
};
```

Un entero C es esencialmente una etiqueta a una posición en la memoria cuyos bytes codifican un valor entero. Un entero en Python es un puntero a una posición en la memoria que contiene toda la información del objeto de Python, incluidos los bytes que contienen el valor entero.

Esta información adicional, que permite que Python se codifique de manera tan libre y dinámica, tiene un costo, que se vuelve especialmente evidente en estructuras que combinan muchos de estos objetos. Por ejemplo, el contenedor multielemento mutable estándar en Python es la lista, y como hemos visto aunque generalmente se utiliza para datos homogéneos, también se pueden crear listas heterogéneas:

```
In [106]: variable = [True, "2", 3.0, 4]
          [type(item) for item in variable]
Out[106]: [bool, str, float, int]
```

Pero esta flexibilidad tiene un costo: para permitir estos tipos flexibles, cada elemento de la lista debe contener su propia información de tipo, recuento de referencias y otra información, es decir, cada elemento es un objeto Python completo. En el caso especial de que todas las variables sean del mismo tipo, gran parte de esta información es redundante: puede ser mucho más eficiente almacenar datos en una matriz de tipo fijo.

## Asignación en Python

La asignación en Python, por defecto, manipula referencias de objetos. Es decir, las variables en Python no guardan directamente valores ni objetos sino referencias a éstos. Por lo que cuando se hace una asignación no se están copiando esos valores:

```
x = y  # No hace una copia de 'y' en 'x'
x = y  # Hace que 'x' referencie al objeto referenciado por 'y'
```

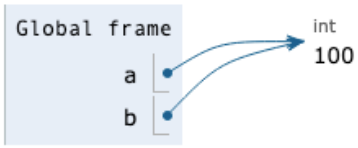
> En Python cuando se declara una variable el intérprete le asigna un identificador único interno que se corresponde con un número entero que el sistema utiliza para diferenciarla de otras variables (y de otros objetos) en la memoria. Dicho identificador se corresponde con un número entero que se genera en cada ejecución y se puede obtener con la función `id` .

Estos mecanismos del intérprete para referenciar variables y duplicar la información se utilizan junto a otros para optimizar el uso de los recursos. Es una característica del lenguaje es muy útil, pero es necesario tenerla en cuenta siempre. Además, su comportamiento depende de la mutabilidad de los datos que intervengan en la asignación:

#### Tipos de datos inmutables: números, cadenas y tuplas

El intérprete de Python tiene un mecanismo de optimización que de forma automática no sólo comparte direcciones entre variables sino contenidos mientras éstos no varían. Los objetos inmutables no permiten el cambio sin generar una nueva instancia en memoria de la variable que los referencia.

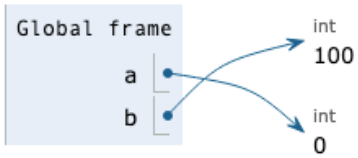
In [107]:
```
# comportamiento idéntico
# a = 100
# b = 100
a = 100
b = a
```




In [108]:
```
# a y b comparten dirección/id y contenido
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = 100
b = 100
id(a) = 94638675364192
id(b) = 94638675364192
```

In [109]:
```
a = 0  # se crea una nueva variable
```



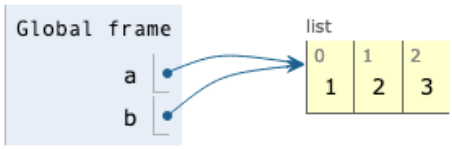

In [110]:
```
# a y b tienen dirección/id y contenidos distintos
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = 0
b = 100
id(a) = 94638675360992
id(b) = 94638675364192
```

**Tipos de datos mutables: listas, conjuntos y diccionarios**

Los objetos mutables comparten las direcciones y los datos.

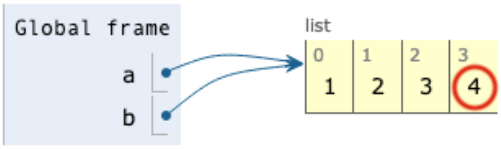
In [111]:
```
a = [1, 2, 3]
b = a
```



In [112]:
```
# a y b comparten dirección/id y contenido
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = [1, 2, 3]
b = [1, 2, 3]
id(a) = 140469832044160
id(b) = 140469832044160
```

In [113]:
```
a.append(4)
```

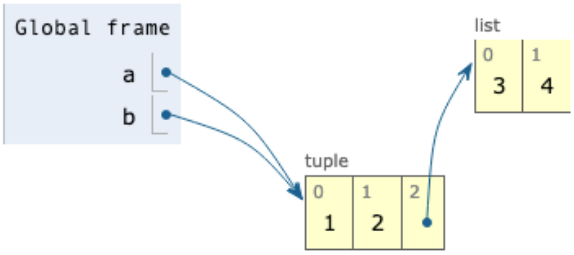



In [114]:
```
# a y b comparten dirección/id y contenido
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
id(a) = 140469832044160
id(b) = 140469832044160
```

**Objetos mutables dentro de objetos inmutables:**

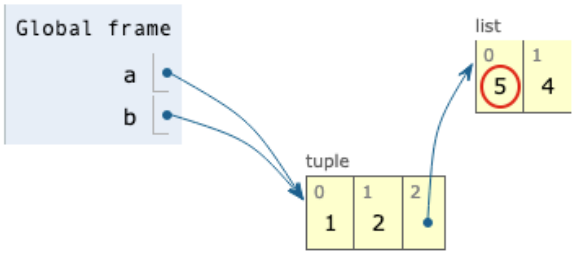
In [115]:
```
a = (1, 2, [3, 4])
b = a
```




In [116]:
```
# a y b comparten dirección/id y contenido
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = (1, 2, [3, 4])
b = (1, 2, [3, 4])
id(a) = 140469831959936
id(b) = 140469831959936
```

In [117]:
```
b[2][0] = 5
```

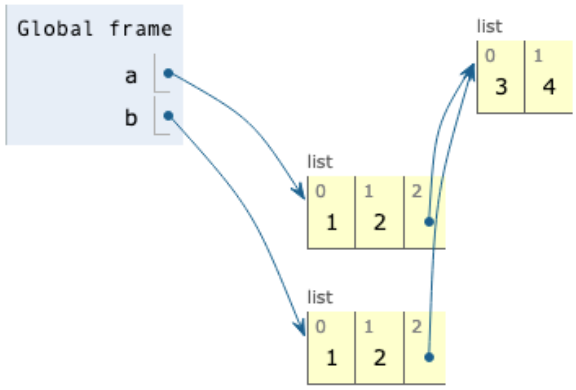



In [118]:
```
# a y b comparten dirección/id y contenido
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = (1, 2, [5, 4])
b = (1, 2, [5, 4])
id(a) = 140469831959936
id(b) = 140469831959936
```

Un caso con especial es el corte de las listas que no genera copias de los objetos en la lista; sólo copia las referencias a ellos. Es lo que se denomina una copia *superficial*. Cuando la lista está compuesta por elementos inmutables no es ningún problema, los cambios en la lista original no afectan a la sección puesto que son variables distintas, pero cuando la lista está compuesta por elementos mutables hay que tener en cuenta que éstos comparten la referencia:

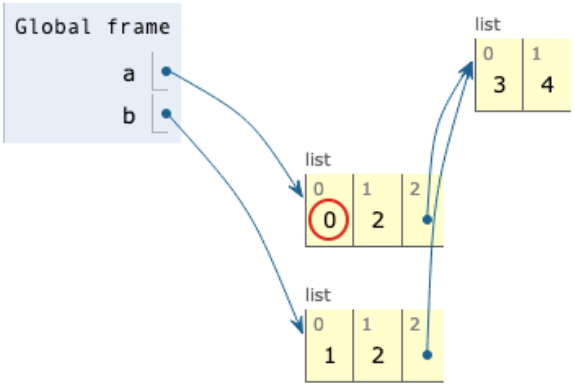
In [119]:
```
a = [1, 2, [3, 4]]
b = a[:]  # copia superficial
```




In [120]:
```
# a y b comparten las referencias a los objetos, pero no dirección/id
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = [1, 2, [3, 4]]
b = [1, 2, [3, 4]]
id(a) = 140469822745856
id(b) = 140470505795520
```

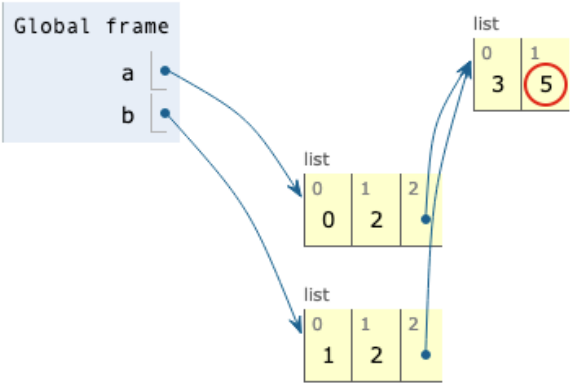
In [121]:
```
a[0] = 0
```




In [122]:
```
# a y b comparten las referencias a los objetos, pero no dirección/id
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = [0, 2, [3, 4]]
b = [1, 2, [3, 4]]
id(a) = 140469822745856
id(b) = 140470505795520
```

In [123]:
```
a[2][1] = 5
```




In [124]:
```
# a y b comparten las referencias a los objetos, pero no dirección/id
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = [0, 2, [3, 5]]
b = [1, 2, [3, 5]]
id(a) = 140469822745856
id(b) = 140470505795520
```

**Operaciones de copia superficial y profunda**

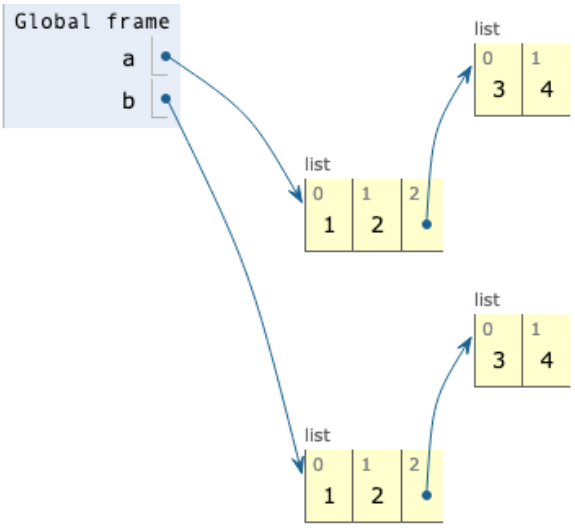
Para evitar errores se recomienda el uso de las funciones que proporciona el módulo **copy** de la librería estándar de Python. Este módulo consta de funciones para duplicar variables y otros objetos con distinto nivel del profundidad: `copy` para copias superficiales y `deepcopy` para copias profundas.

La diferencia entre copia superficial y profunda solo es relevante para objetos compuestos (objetos que contienen otros objetos, como listas o instancias de clase):

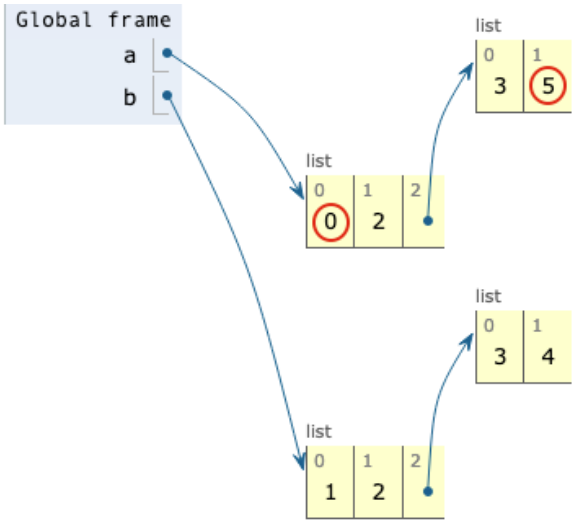
- Una copia superficial (*shallow copy*) construye un nuevo objeto compuesto y luego (en la medida de lo posible) inserta referencias en él a los objetos encontrados en el original.
- Una copia profunda (*deep copy*) construye un nuevo objeto compuesto y luego, recursivamente, inserta copias en él de los objetos encontrados en el original. Por ello, un objeto copiado de forma superficial no es completamente independiente del original como sí ocurre con uno obtenido de una copia profunda, implicando esta última un proceso de creación más lento.

Para copias superficiales además de la función `copy` se pueden utilizar la funciones `list`, `dict` y `set` empleadas para declarar listas, diccionarios y conjuntos, respectivamente.

In [125]:
```
import copy
a = [1, 2, [3, 4]]
b = copy.deepcopy(a)  # copia profunda
```




In [126]:
```
a[0] = 0
a[2][1] = 5
```




In [127]:
```
# a y b no comparten nada
print("a =",a)
print("b =",b)
print("id(a) =",id(a))
print("id(b) =",id(b))
```

```
a = [0, 2, [3, 5]]
b = [1, 2, [3, 4]]
id(a) = 140469831915392
id(b) = 140469822677504
```

## Comprensiones de listas, conjuntos y diccionarios

Las comprensiones (*comprehensions*) de listas son una de las funciones de lenguaje de Python más potentes. Permiten formar una nueva lista de forma concisa al filtrar los elementos de una colección, transformando los elementos que pasan el filtro en una expresión concisa. Su sintaxis básica es:

```
[expresion for valor in coleccion if condicion(valor)]
```

El código equivalente sería:

```
resultado = []
for valor in coleccion:
    if condicion (valor):
        resultado.append(expresion(valor))
```

La *condición* de filtrado puede omitirse, dejando únicamente la *expresión*.

In [128]:
```
cadenas = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in cadenas if len(x) > 2]
```

Out[128]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Las comprensiones de conjuntos y diccionarios son una extensión natural, produciendo conjuntos y diccionarios de una manera similar a la utilizada con las listas. Una comprensión para un diccionario sería:

```
dict_comp = {key-expr : value-expr for value in coleccion if condicion}
```

Para un conjunto sólo habría que sustituir las llaves por corchetes:

```
set_comp = {expr for value in coleccion if condicion}
```

In [129]:
```
{len(x) for x in cadenas}
```

Out[129]:
```
{1, 2, 3, 4, 6}
```

In [130]:
```
# También podríamos expresar esto más funcionalmente usando la función map
set(map(len, cadenas))
```

Out[130]:
```
{1, 2, 3, 4, 6}
```

In [131]:
```
{indice : valor for indice, valor in enumerate(cadenas)}
```

Out[131]:
```
{0: 'a', 1: 'as', 2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

En general, las operaciones sobre compresiones serán uno o dos (o más) órdenes de magnitud más rápidas que sus equivalentes puras de Python, con el mayor impacto en cualquier tipo de cálculo numérico, ver temporización y profiling:

In [132]:
```
%%time
a = range(100000000)
b = []
for i in a:
    b.append(i^2)
```

```
CPU times: user 17.9 s, sys: 1.8 s, total: 19.7 s
Wall time: 19.8 s
```

In [133]:
```
%time b = [i^2 for i in range(100000000)]
```

```
CPU times: user 9.97 s, sys: 2.3 s, total: 12.3 s
Wall time: 12.4 s
```

#### Comprensiones de listas anidadas

In [134]:
```
# Queremos obtener una lista única que contenga todos los nombres con una o más letras 'e' en ellos
datos = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
         ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
nombres_buscados = []
for lista_nombres in datos:
    for nombre in lista_nombres:
        if nombre.upper().count('E') >= 1:
            nombres_buscados.append(nombre)
nombres_buscados
```

Out[134]:
```
['Emily', 'Michael', 'Steven', 'Javier']
```

In [135]:
```
[nombre for nombres in datos for nombre in nombres if nombre.upper().count('E')
>= 1]
```

Out[135]:
```
['Emily', 'Michael', 'Steven', 'Javier']
```

In [136]:
```
lista_varias_tuplas = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
[x for tupla in lista_varias_tuplas for x in tupla]
```

Out[136]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [137]:
```
# [list(tupla) for tupla in lista_varias_tuplas]
[[x for x in tupla] for tupla in lista_varias_tuplas]
```

Out[137]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

## Funciones

Las funciones son el método principal y más importante de organización de código y reutilización en Python. Como regla general, si se anticipa la necesidad de repetir el mismo código o uno muy similar más de una vez, puede valer la pena escribir una función reutilizable. Las funciones también pueden ayudar a hacer que el código sea más legible al darle un nombre a un grupo de declaraciones.

Las funciones se declaran con la palabra clave `def` y devuelven valores con la palabra clave `return` :

```
def mi_funcion(x, y, z=1.5):
    '''Multiplica/divide z por la suma de x e y
       en función del valor de z'''
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

No hay problema con tener múltiples declaraciones `return` . Si Python llega al final de una función sin encontrar una declaración de retorno, se devuelve automáticamente `None` .

Cada función puede tener argumentos posicionales y argumentos de tipo *palabra clave*. Los argumentos de tipo *palabra clave* se usan comúnmente para especificar valores predeterminados o argumentos opcionales. En la función anterior, `x` e `y` son argumentos *posicionales*, mientras que `z` es un argumento de tipo *palabra clave*. Esto significa que la función se puede llamar de cualquiera de estas maneras:

```
mi_funcion(5, 6, z=0.7)    # z define un valor por defecto, sino se indica lo contr
ario tomará ese valor
mi_funcion(3.14, 7, 3.5)
mi_funcion(10, 20)
```

La principal restricción en los argumentos de una función es que los de tipo *palabra clave* deben seguir a los argumentos *posicionales* (si los hay). Se pueden especificar argumentos de tipo *palabra clave* en cualquier orden, por lo que no hay que recordar en qué orden se especificaron los argumentos de la función y solo cuáles son sus nombres.

También es posible utilizar palabras clave para pasar argumentos posicionales.

```
mi_funcion(x=5, y=6, z=7)
mi_funcion(y=6, x=5, z=7)
```

> **PEP8: Funciones**
>
> A la definición de una función la deben anteceder dos líneas en blanco. Al asignar parámetros por defecto, no debe dejarse espacios en blanco ni antes ni después del signo `=` .

Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de tupla. Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco ( `*` ):

```
def mi_funcion(x, y, z=1.5, *otros)
```

Es posible también, obtener parámetros arbitrarios como pares de `clave=valor` . En estos casos, al nombre del parámetro deben precederlo dos astericos ( `**` ):

```
def mi_funcion(x, y, z=1.5, *otros, **pares)
```

Puede ocurrir además, una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla. En este caso, el signo asterisco ( `*` ) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función:

```
parametros = (3.14, 7, 3.5)
mi_funcion(*parametros)
```

## Devolviendo múltiples valores

Una de las características más flexibles de las funciones en Python, es la capacidad de devolver múltiples valores desde una función con una sintaxis simple. Aquí hay un ejemplo:

```
def f():
    a=5
    b=6
    c=7
    return a, b, c
a, b, c = f()
```

En el análisis de datos y otras aplicaciones científicas, puede que te encuentres haciendo esto a menudo. Lo que sucede aquí es que la función en realidad solo devuelve un objeto, es decir, una tupla, que luego se desempaqueta en las variables de resultado. En el ejemplo anterior, podríamos haber hecho esto en su lugar:

```
valor_retorno = f()
```

En este caso, `valor_retorno` sería una tupla 3 con las tres variables devueltas. Una alternativa potencialmente atractiva para devolver varios valores como antes podría ser devolver un diccionario:

```
def f ():
    a = 5
    b = 6
    c = 7
    return {'a': a, 'b': b, 'c': c}
```

## Alcance y ámbito de las variables

Las funciones pueden acceder a las variables en dos ámbitos diferentes: global y local. Todas las variables que se asignan dentro de una función por defecto se asignan al espacio de nombres local. El espacio de nombres local se crea cuando se llama a la función y se rellena inmediatamente con los argumentos de la función. Una vez finalizada la función, el espacio de nombres local se destruye (con algunas excepciones que están fuera del alcance de este capítulo).

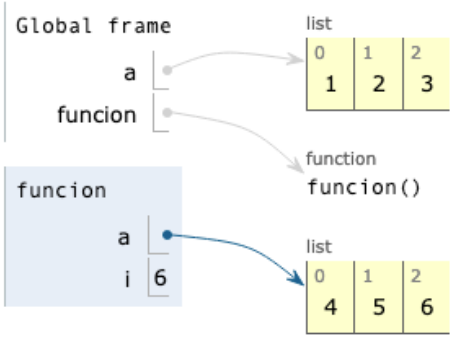
Las variables definidas dentro de una función tienen alcance local. Su alcance es limitado dentro de la función. Considere la siguiente función:

```
def funcion():
    a = []
    for i in range(5):
        a.append(i)
```

Cuando se llama a `funcion()`, se crea la lista vacía `a`, se agregan cinco elementos y luego se destruye `a` cuando se sale de la función. En este caso después de la ejecución de `funcion()`, la variable `a` no estaría definida.

Cuando existen dos variables con el mismo nombre dentro y fuera de la función, se tratan como variables diferentes. Supongamos que en cambio hubiéramos declarado lo siguiente:

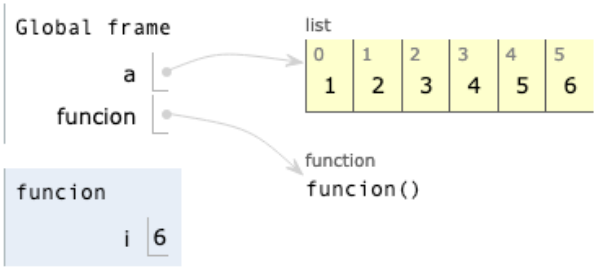
```
a = [1, 2, 3]
def funcion():
    a = []
    for i in range(4, 7):
        a.append(i)
```



En este caso después de la ejecución de `funcion()`, la variable `a` valdría `[1, 2, 3]`.

La asignación de variables fuera del alcance de la función es posible, pero esas variables deben declararse como globales a través de la palabra clave `global`:

```
a = [1, 2, 3]
def funcion():
    global a
    for i in range(4, 7):
        a.append(i)
```




En este caso después de la ejecución de `funcion()`, la variable `a` valdría `[1, 2, 3, 4, 5, 6]`.

## Funciones anónimas (funciones Lambda)

Python es compatible con las llamadas funciones anónimas o lambda, que son una forma de escribir funciones que consisten en una sola declaración, cuyo resultado es el valor de retorno. Se definen con la palabra clave `lambda`, que no tiene otro significado que "*estamos declarando una función anónima*":

```
def doblar(x):
    return x * 2
doblar_anonima = lambda x: x * 2
```

Son especialmente convenientes en el análisis de datos porque, como veremos, hay muchos casos en los que las funciones de transformación de datos tomarán funciones como argumentos. A menudo lleva menos codificación (y más claridad) pasar una función lambda en lugar de escribir la declaración completa de una función o incluso asignar la

In [138]:
```
# lista de cadenas ordenada en base al número distinto de letras
cadenas = ['foo', 'card', 'bar', 'aaaa', 'abab']
cadenas.sort(key=lambda x: len(set(list(x))))
cadenas
```

Out[138]:
```
['aaaa', 'foo', 'abab', 'bar', 'card']
```

## Llamadas de retorno (callback)

Para poder hacer llamadas a funciones de manera dinámica, es decir, desconociendo el nombre de la función a la que se deseará llamar (por ejemplo funciones pasadas por parámetro a otras funciones), Python dispone de dos funciones nativas: `locals()` y `globals()`. Ambas funciones, retornan un diccionario. En el caso de `locals()`, éste diccionario se compone de todos los elementos de ámbito local, mientras que el de `globals()`, retorna lo propio pero a nivel global. Estas funciones tienen una gran variedad de usos y son especialmente útiles en sincronización de procesos.

In [139]:
```
def una_funcion(nombre):
    return "Hola "+nombre

def otra_funcion(funcion_param):
    """Llamada de retorno a nivel global"""
    return globals()[funcion_param]("Laura")

print(otra_funcion("una_funcion"))
```

```
Hola Laura
```

In [140]:
```
print(locals()["una_funcion"]("Facundo"))
```

```
Hola Facundo
```

Python proporciona funcionalidades para comprobar que una función existe y pueda ser llamada. El operador `in` , nos permitirá conocer si un elemento se encuentra dentro de una colección, mientras que la función `callable()` nos dejará saber si esa función puede ser llamada.

In [141]:
```
def otra_funcion(funcion_param):
    if funcion_param in globals():
        if callable(globals()[funcion_param]):
            return globals()[funcion_param]("Laura")
    else:
        return "Función no encontrada"
print(otra_funcion("pepe"))
```

```
Función no encontrada
```

## Generadores

Tener una forma consistente de iterar sobre secuencias, como los objetos en una lista o líneas en un archivo, es una característica importante de Python. Esto se logra mediante el protocolo del iterador, una forma genérica de hacer iterables los objetos. Un generador, en su forma más simple, es una función que devuelve elementos de uno en uno en lugar de devolver una colección de elementos. La ventaja más importante de esto es que requiere muy poca memoria y no necesita tener un tamaño predefinido.

Por ejemplo, al iterar sobre un dict se obtienen las claves de dict:

In [142]:
```
un_diccionario = {'a': 1, 'b': 2, 'c': 3}
for clave in un_diccionario:
    print(clave, end=" ")
```

```
a b c
```

Al ejecutar el bucle `for` el intérprete de Python intenta crear un iterador a partir de `un_diccionario` :

In [143]:
```
iterator_un_diccionario = iter(un_diccionario)
iterator_un_diccionario
```

Out[143]:
```
<dict_keyiterator at 0x7fc1ade4cd60>
```

Un iterador es cualquier objeto que proporciona objetos al intérprete de Python cuando se use en un contexto como un bucle `for` . La mayoría de los métodos que esperan una lista o un objeto similar a una lista también aceptarán cualquier objeto iterable.

In [144]:
```
list(iterator_un_diccionario)
```

Out[144]:
```
['a', 'b', 'c']
```

Un *generador* es una forma concisa de construir un nuevo objeto iterable. Mientras que las funciones normales ejecutan y devuelven un solo resultado a la vez, los generadores devuelven una secuencia de múltiples resultados, deteniéndose después de cada uno hasta que se solicita el siguiente. Para crear un *generador*, se define una función que utiliza la palabra clave `yield` en lugar de `return` :

In [145]:
```
def cuadrados(n=10):
    print('Generando cuadrados de 1 a {0}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i**2

# Cuando se llama directamente a un generador, no se ejecuta el código de forma inmediata
cuadrados()
```

Out[145]:
```
<generator object cuadrados at 0x7fc1ade4a3c0>
```

In [146]:
```
# El código se ejecuta cuando solicitan los elementos del generador
for x in cuadrados():
    print(x, end=" ")
```

```
Generando cuadrados de 1 a 100
1 4 9 16 25 36 49 64 81 100
```

Otra forma aún más concisa de hacer un generador es utilizar una *expresión de un generador*, similar a las expresiones de compresión de listas y diccionarios:

In [147]:
```
generador = (x ** 2 for x in range(1,11))
print(next(generador))
print(next(generador))
for x in generador:
    print(x, end=" ")
```

```
1
4
9 16 25 36 49 64 81 100
```

Sin embargo, es importante tener en cuenta sus ventajas y desventajas. Los siguientes son los pros más importantes:

- Uso de memoria. Los elementos se pueden procesar de uno en uno, por lo que generalmente no es necesario mantener la lista completa en la memoria.
- Los resultados pueden depender de factores externos, en lugar de tener una lista estática. Piense en procesar una cola / pila, por ejemplo.
- Los generadores son vagos. Esto significa que si está utilizando solo los primeros cinco resultados de un generador, el resto ni siquiera se calculará.
- Generalmente, es más sencillo escribir que las funciones generadoras de listas.

Los contras más importantes:

- Los resultados están disponibles solo una vez. Después de procesar los resultados de un generador, no se puede volver a utilizar.
- Se desconoce el tamaño hasta que termine el procesamiento, lo que puede ser perjudicial para ciertos algoritmos.
- Los generadores no son indexables, lo que significa que `generador[5]` no funcionará.

Los generadores también permiten que se les envíe información, de forma que podemos interactuar entre la información enviada y el valor generado en el paso:

In [148]:
```python
def impares(n=10):
    for i in range(1, n + 1, 2):
        valor = yield
        print(f'-> El generador recibe {valor}')
        yield valor ** i
        print(f'<- El generador devuelve {valor}^{i}: {valor ** i}')

impar = impares()
next(impar)
impar.send(4)
next(impar)
impar.send(3)
next(impar)
impar.send(2)
next(impar)
```

```
-> El generador recibe 4
<- El generador devuelve 4^1: 4
-> El generador recibe 3
<- El generador devuelve 3^3: 27
-> El generador recibe 2
<- El generador devuelve 2^5: 32
```

## Módulos, paquetes y espacios de nombres

Las funciones se pueden organizar en bibliotecas o módulos, ofrecen por tanto, otra capa de organización del código. Idealmente, se debería incluir un grupo de funciones relacionadas en la misma biblioteca o módulo. En Python, cada uno de nuestros archivos `.py` se denominan módulos. Estos módulos, a la vez, pueden formar parte de paquetes. Un paquete, es una carpeta que contiene archivos `.py` . Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado `__init__.py` . Estos archivos son necesarios para hacer que Python trate los directorios que contienen el archivo como paquetes. En el caso más simple, `__init__.py` solo puede ser un archivo vacío, pero también puede ejecutar el código de inicialización del paquete o establecer la variable `__all__` , que se describe más adelante.

Los paquetes, a la vez, también pueden contener otros sub-paquetes. Los módulos, no necesariamente, deben pertenecer a un paquete

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Al importar el paquete, Python busca en los directorios en `sys.path` buscando el subdirectorio del paquete. Los usuarios del paquete pueden importar módulos individuales del paquete, por ejemplo:

```
import sound.effects.echo
```

Esto carga el submódulo `sound.effects.echo` . Debe ser referenciado con su nombre completo.

```
sound.effects.echo.echofilter (entrada, salida, retardo = 0.7, atten = 4)
```

Una forma alternativa de importar el submódulo es:

```
from sound.effects import echo
```

Esto también carga el submódulo `echo` , y lo hace disponible sin su prefijo de paquete, por lo que se puede usar de la siguiente manera:

```
echo.echofilter (entrada, salida, retardo = 0.7, atten = 4)
```

Otra variación más es importar la función o variable deseada directamente:

```
from sonido.effects.echo import echofilter
```

De nuevo, esto carga el submódulo de eco, pero hace que su función echofilter () esté disponible directamente:

```
echofilter (entrada, salida, retardo = 0.7, atten = 4)
```

Tenga en cuenta que cuando se utiliza `from package import item` , el elemento puede ser un submódulo (o subpaquete) del paquete, o algún otro nombre definido en el paquete, como una función, clase o variable. La

declaración de importación primero prueba si el artículo está definido en el paquete. Si no, asume que es un módulo e intenta cargarlo. Si no lo encuentra, se genera una excepción `ImportError` .

Por el contrario, cuando se usa una sintaxis `import item.subitem.subsubitem` , cada elemento, excepto el último, debe ser un paquete; el último elemento puede ser un módulo o un paquete, pero no puede ser una clase o función o variable definida en el elemento anterior.

Es posible también abreviar los `namespace` mediante un "*alias*". Para ello, durante la importación, se asigna la palabra clave `as` seguida del *alias* con el cuál nos referiremos en el futuro a ese namespace importado:

```
import sound.effects.echo as eco
```

Esto carga el submódulo `sound.effects.echo` . Debe ser referenciado con el alias definido.

```
eco.echofilter (entrada, salida, retardo = 0.7, atten = 4)
```

Es posible también, importar más de un elemento en la misma instrucción. Para ello, cada elemento irá separado por una coma (,) y un espacio en blanco:

```
from sound.effects import echo, surround
```

¿Qué sucede cuando el usuario escribe `from sound.effects import *` ? Idealmente, uno esperaría que esta importación solicite al sistema de archivos que encuentre qué submódulos están presentes en el paquete y los importe todos. Esto puede llevar mucho tiempo y la importación de submódulos puede tener efectos secundarios no deseados que solo deberían ocurrir cuando el submódulo se importa explícitamente.

La única solución es que el autor del paquete proporcione un índice explícito del paquete. La declaración de importación utiliza la siguiente convención: si el código `__init__.py` de un paquete define una lista llamada `__all__` , se considera la lista de nombres de módulos que deben importarse cuando se encuentra desde la importación de paquetes `*` . Depende del autor del paquete mantener esta lista actualizada cuando se lance una nueva versión del paquete. Los autores de paquetes también pueden decidir no admitirlo, si no ven un uso para importar * de su paquete. Por ejemplo, el archivo `sound/effects/__init__.py` podría contener el siguiente código:

```
__all__ = ["echo", "surround", "reverse"]
```

Esto significaría que `from sound.effects import *` importaría los tres submódulos nombrados del paquete de sonido. Si `__all__` no está definido, la declaración de `sound.effects import *` no importa todos los submódulos del paquete `sound.effects` en el espacio de nombres actual; solo garantiza que el paquete `sound.effects` haya sido importado (posiblemente ejecutando cualquier código de inicialización en `__init__.py` ) y luego importa los nombres que estén definidos en el paquete. Esto incluye cualquier nombre definido (y submódulos cargados explícitamente) por `__init__.py` . También incluye los submódulos del paquete que fueron cargados explícitamente por las declaraciones de importación anteriores.

> **PEP8: importación**
>
> La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos. Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación. Entre cada bloque de importaciones, debe dejarse una línea en blanco.
> Aunque ciertos módulos están diseñados para exportar solo nombres que siguen ciertos patrones cuando se usa `import *` , se considera una mala práctica en el código de producción.

```
In [149]: import math
          math.sqrt(144)
```

```
Out[149]: 12.0
```

```
In [150]: import math as m
          m.sqrt(12)
```

```
Out[150]: 3.4641016151377544
```

```
In [151]: from math import sqrt
          sqrt(39)
```

```
Out[151]: 6.244997998398398
```

### La función `dir`

La función incorporada `dir` se usa para averiguar qué nombres define un módulo. Devuelve una lista ordenada de cadenas:

In [152]:
```
dir(math)
```

Out[152]:
```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'comb',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'dist',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'isqrt',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'perm',
 'pi',
 'pow',
 'prod',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

In [153]:
```
'cosh' in dir(math)
```

Out[153]:
```
True
```

## Librerías en Data Science

Hay varias librerías de Python que son fundamentales en Data Science. Echemos un vistazo rápido al tipo de funcionalidad que ofrecen:

- NumPy: Ofrece una estructura crítica para el almacenamiento y operaciones con datos: el array multidimensional. NumPy es una librería de bajo nivel sobre la que se han desarrollado otras.
- pandas: Ejemplo de librería desarrollada sobre NumPy. Ofrece dos estructuras de datos basadas en el array NumPy: la serie (estructura unidimensional) y el DataFrame (estructura bidimensional).
- SciPy: Esta librería ofrece herramientas matemáticas de todo tipo: resolución de ecuaciones diferenciales, distribuciones, gestión de matrices...
- Matplotlib: Es la librería de visualización referencia en el entorno Python. Aun cuando ofrece herramientas de bajo nivel y su uso no es especialmente amigable, sigue siendo obligado su conocimiento, más cuando otras librerías de visualización se han construido sobre ésta.
- seaborn: Otra librería de visualización, en este caso desarrollada sobre Matplotlib. Mucho más amigable que Matplotlib y con un estilo visual mucho más atractivo, es la primera opción en muchos casos.
- Bokeh: Tercera librería de visualización de esta lista, aunque en este caso no está basada en Matplotlib. Bokeh ofrece visualizaciones interactivas muy atractivas y útiles.
- Scikit-learn: Librería de referencia en el mundo del Machine Learning para Python. Ofrece innumerables algoritmos y herramientas imprescindibles en cualquier proyecto de análisis de datos.
- TensorFlow: TensorFlow ofrece herramientas para la definición y entrenamiento de redes neuronales.
- Keras: Keras se ofrece como interfaz de alto nivel para librerías como TensorFlow, Theano o CNTK.
- NLTK: Librería de procesamiento de lenguaje natural, con multitud de herramientas orientadas al análisis de textos.
- XGBoost, LightGBM: Librerías que implementan los algoritmos homónimos, fundamentales en entornos tabulares.

## Errores y manejo de Excepciones

Incluso si una declaración o expresión es sintácticamente correcta, puede causar un error cuando se intenta ejecutarla. Los errores detectados durante la ejecución se llaman excepciones y no son incondicionalmente fatales. El manejo adecuado de los errores o excepciones de Python es una parte importante de la creación de programas sólidos. Python tiene muchas excepciones integradas que responden a posibles problemas con la ejecución del código. Algunas de las más comunes son `ZeroDivisionError` , `ValueError` , `IndexError` y `TypeError` . Cuando se produce una excepción, el proceso actual se detiene y pasa el error al proceso llamador hasta que el error se gestiona o el programa finaliza.

El control de excepciones se realiza mediante el bloque `try-except` :

```
try:
   # hacer algo susceptible de generar un error
   pass
except <Error>:
   # manejar la exception de tipo <Error>
   pass
except (<Error1>, <Error2>):
   # manejar múltiples excepciones
   pass
except:
   # manejar el resto de posibles excepciones
   pass
```

Se puede usar la palabra clave `else` para definir un bloque de código que se ejecutará si no se generaron errores:

```
try:
   # hacer algo susceptible de generar un error
   pass
except <Error>:
   # manejar la exception de tipo <Error>
   pass
else:
   # se ejecuta cuando no se generan excepciones
   pass
```

El bloque `finally` , si se especifica, se ejecutará independientemente de si el bloque `try` genera un error o no:

```
try:
   # hacer algo susceptible de generar un error
   pass
except:
   # maneja cualquier excepción
   pass
finally:
   # se ejecuta en cualquier caso
   pass
```

Los controladores de excepciones no solo manejan excepciones si ocurren inmediatamente en la cláusula `try` , sino también si ocurren dentro de funciones que se llaman (incluso indirectamente) en la cláusula `try` .

Como ejemplo, la función `float` de Python es capaz de convertir una cadena a un número en punto flotante, pero falla generando el error `ValueError` con entradas incorrectas:

```
In [154]: float('1.2345')
```

```
Out[154]: 1.2345
```

In [155]:
```
float('un texto')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-155-57269809c8ac> in <module>
----> 1 float('un texto')

ValueError: could not convert string to float: 'un texto'
```

In [156]:
```
def procesar_float(valor):
    try:
        float(valor)
    except ValueError:
        print('Valor proporcionado no válido:', end=" ")
    except TypeError:
        print('Tipo proporcionado no válido, se esperaba un número o una caden
a:', end=" ")
    else:
        print('Procesamiento correcto:', end=" ")
        return float(valor)
    finally:
        print(valor)

procesar_float('1.2345')
```

```
Procesamiento correcto: 1.2345
```

Out[156]:
```
1.2345
```

In [157]:
```
procesar_float('un texto')
```

```
Valor proporcionado no válido: un texto
```

In [158]:
```
procesar_float((1, 2))
```

```
Tipo proporcionado no válido, se esperaba un número o una cadena: (1, 2)
```

Desde el código del programa pueden lanzar excepciones utilizando la sentenica `raise` , que fuerza que una excepción específica ocurra:

```
raise ValueError  # equivalente a 'raise ValueError()'
raise NameError('Se ha producido un error...')
```

El único argumento que se indica es la excepción que se quiere elevar/generar. Del usuario puede definir sus propias excepciones. Las excepciones normalmente deben derivarse de la clase `Exception` , ya sea directa o indirectamente. Si se pasa una clase de excepción, se instanciará implícitamente llamando a su constructor sin argumentos.

Se pueden definir clases de excepción que hacen cualquier cosa que cualquier otra clase pueda hacer, pero generalmente se mantienen simples, a menudo solo ofrecen una serie de atributos que permiten que los manejadores extraigan información sobre el error para la excepción.

In [159]:
```
def verificar_par(valor):
    if valor % 2 == 0:
        return valor
    else:
        raise ValueError("El valor proporcionado no es par")

verificar_par(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-159-c91168707c77> in <module>
      5         raise ValueError("El valor proporcionado no es par")
      6
----> 7 verificar_par(3)

<ipython-input-159-c91168707c77> in verificar_par(valor)
      3         return valor
      4     else:
----> 5         raise ValueError("El valor proporcionado no es par")
      6
      7 verificar_par(3)

ValueError: El valor proporcionado no es par
```

## Control de excepciones: `%xmode`

La mayoría de las veces, cuando un script de Python falla, generará una excepción. Cuando el intérprete encuentra una de estas excepciones, la información sobre la causa del error se puede encontrar en el rastreo, al que se puede acceder desde Python. Con la función mágica `%xmode`, IPython permite controlar la cantidad de información a mostrar cuando se genera la excepción.

In [160]:
```
def func1(a, b):
    return a/b

def func2(x):
    a=x
    b=x-1
    return func1(a, b)

func2(1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-160-a62b6a523a8f> in <module>
      7     return func1(a, b)
      8
----> 9 func2(1)

<ipython-input-160-a62b6a523a8f> in func2(x)
      5     a=x
      6     b=x-1
----> 7     return func1(a, b)
      8
      9 func2(1)

<ipython-input-160-a62b6a523a8f> in func1(a, b)
      1 def func1(a, b):
----> 2     return a/b
      3
      4 def func2(x):
      5     a=x

ZeroDivisionError: division by zero
```

El comando `%xmode` tiene un solo argumento, el modo, y hay tres posibilidades: simple ( `Plain` ), contextual ( `Context` ) y detallado ( `Verbose` ). El modo por defecto es contextual.

In [161]:
```
%xmode Plain
func2(1)
```

```
Exception reporting mode: Plain

Traceback (most recent call last):

  File "<ipython-input-161-2c0c65acd0a8>", line 2, in <module>
    func2(1)

  File "<ipython-input-160-a62b6a523a8f>", line 7, in func2
    return func1(a, b)

  File "<ipython-input-160-a62b6a523a8f>", line 2, in func1
    return a/b

ZeroDivisionError: division by zero
```

In [162]:
```
%xmode Verbose
func2(1)
```

```
Exception reporting mode: Verbose

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-162-180acea4108b> in <module>
      1 get_ipython().run_line_magic('xmode', 'Verbose')
----> 2 func2(1)
        global func2 = <function func2 at 0x7fc1ade525e0>

<ipython-input-160-a62b6a523a8f> in func2(x=1)
      5     a=x
      6     b=x-1
----> 7     return func1(a, b)
        global func1 = <function func1 at 0x7fc1ade52820>
        a = 1
        b = 0
      8
      9 func2(1)

<ipython-input-160-a62b6a523a8f> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a/b
        a = 1
        b = 0
      3
      4 def func2(x):
      5     a=x

ZeroDivisionError: division by zero
```

## Ficheros y Sistema Operativo

Aunque para el procesamiento de datos se utilizan normalmente herramientas de alto nivel como las proporcionadas por la librería `Pandas` para leer archivos de datos del disco en estructuras de datos de Python. Sin embargo, es importante comprender los conceptos básicos de cómo trabajar con archivos en Python.

Para abrir un archivo para leer o escribir, se usa la función incorporada `open` con una ruta de archivo relativa o absoluta, un modo de apertura y una codificación.

```
open(ruta, [[modo="rt"], encoding="none"])
```

De forma predeterminada, los archivos se abre en modo de solo lectura y texto `'rt'` . Los manejadores de archivos se pueden usar como una lista e iterar sobre las líneas que contienen. Es importante cerrar explícitamente los archivos cuando haya terminado de procesarlos. Al cerrar un archivo se liberan sus recursos en el sistema operativo.

In [163]:
```
path = './data/teoria/segismundo.txt'
f = open(path)
# las líneas incluyen el caracter de fin-de-linea (EOL), se pueden eliminar con el método 'rstrip'
lineas = [x.rstrip() for x in f]
f.close()
lineas
```

Out[163]:
```
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

Una de las maneras de facilitar la limpieza de archivos abiertos es usar la instrucción `with` , que cierra autoimáticamente los archivos al alcanzar el final del bloque:

In [164]:
```
with open(path) as f:
    lineas = [x.rstrip() for x in f]
lineas
```

Out[164]:
```
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

La siguiente tabla muestra los diferentes modos de trabajo de los ficheros en Python:

| Modo | Descripción |
|---|---|
| r | Solo lectura |
| w | Solo escritura. Sobreescribe el archivo si existe. Crea el archivo si no existe. |
| x | solo escritura. Falla si existe el archivo. |
| a | Añade a un fichero existente. Crea el archivo si no existe. |
| r+ | Lectura y escritura |
| b | Modo binario (i.e., `rb`) |
| t | Modo texto (valor por defecto). Automáticamente decodifica los bytes a Unicode |

La siguiente tabla muestra los métodos y atributos más habituales de los ficheros:

| Método | Descripción |
|---|---|
| read([size]) | Devuelve los datos del archivo como una cadena, el argumento opcional `size` indica el número de bytes para leer |
| readlines([size]) | Devuelve una lista con las líneas del archivo, el argumento opcional `size` indica el número de líneas para leer |
| write(str) | Escribe la cadena pasada al archivo |
| writelines(strings) | Escribe las cadenas pasada al archivo |
| close() | Cierra el fichero |
| flush() | Vaciar el búfer de E/S interno en el disco |
| seek(pos) | Mueve el apuntador del fichero a la posición indicada por `pos` (entero) |
| tell() | Devuelve la posición actual del apuntador del fichero |
| closed | `True` si el fichero está cerrado |
| mode | Devuelve el modo de operación del fichero |
| encoding | Devuelve la codificación de caracteres utilizada por el fichero |

Como se ha indicado, el modo de trabajo por defecto de los archivos de Python (ya sea en lectura o en escritura) es el modo de texto, lo que significa que se trabajará con cadenas de Python (es decir, Unicode). Esto se contrapone con el modo binario, que se configura añadiendo `b` en el modo de archivo.

In [165]:
```
# el fichero 'segismundo.txt' contiene caracteres no-ASCII con codificación UTF
-8
# UTF-8 es una codificación Unicode de longitud variable, así que cuando se sol
icita un número de caracteres
# del archivo, Python lee suficientes bytes del archivo para decodificar esa ca
ntidad de caracteres.
with open(path) as f:
    caracteres = f.read(10)
caracteres
```

Out[165]:
```
'Sueña el r'
```

In [166]:
```
# Si abrimos el archivo en modo 'rb' en cambio, se leen únicamente el número de
bytes solicitados
with open(path, 'rb') as f:
    caracteres = f.read(10)
caracteres
```

Out[166]:
```
b'Sue\xc3\xb1a el '
```

In [167]:
```
# Se puede utilizar el método `decode` de las cadenas de texto para decodificar
la información
# siempre que esta esté correctamente formada
caracteres.decode('utf8')
```

Out[167]:
```
'Sueña el '
```

El modo de texto, combinado con la opción de codificación `encoding` del método `open`, proporciona una manera conveniente de convertir de una codificación Unicode a otra. El módulo `sys` de Python permite conocer información sobre la codificación utilizada por el sistema:

| Método | Descripción |
|---|---|
| sys.getdefaultencoding() | Retorna la codificación de caracteres por defecto |
| sys.getfilesystemencoding() | Retorna la codificación de caracteres que se utiliza para convertir los nombres de archivos unicode en nombres de archivos del sistema |

## El módulo `os` de Python

El módulo os nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos permiten manipular la estructura de directorios. Entre los métodos más destacados de este módulo se encuentran:

| Método | Descripción |
|---|---|
| os.access(path, modo_de_acceso) | Saber si se puede acceder a un archivo o directorio |
| os.getcwd() | Conocer el directorio actual |
| os.chdir(nuevo_path) | Cambiar de directorio de trabajo |
| os.chroot() | Cambiar al directorio de trabajo raíz |
| os.chmod(path, permisos) | Cambiar los permisos de un archivo o directorio |
| os.chown(path, permisos) | Cambiar el propietario de un archivo o directorio |
| os.mkdir(path[, modo]) | Crear un directorio |
| os.mkdirs(path[, modo]) | Crear directorios recursivamente |
| os.remove(path) | Eliminar un archivo |
| os.rmdir(path) | Eliminar un directorio |
| os.removedirs(path) | Eliminar directorios recursivamente |
| os.rename(actual, nuevo) | Renombrar un archivo |
| os.symlink(path, nombre_destino) | Crear un enlace simbólico |

El módulo `os` también nos provee de un diccionario con las variables de entorno relativas al sistema. Se trata del diccionario `environ`:

```
import os
for variable, valor in os.environ.items():
    print("{0}: {1}".format(variable, valor))
```

El módulo os también nos provee del submódulo path (os.path) el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios. Entre ellas, las más destacadas se describen en la siguiente tabla:

| Método | Descripción |
|---|---|
| os.path.abspath(path) | Ruta absoluta |
| os.path.basename(path) | Directorio base |
| os.path.exists(path) | Saber si un directorio existe |
| os.path.getatime(path) | Conocer último acceso a un directorio |
| os.path.getsize(path) | Conocer tamaño del directorio |
| os.path.isabs(path) | Saber si una ruta es absoluta |
| os.path.isfile(path) | Saber si una ruta es un archivo |
| os.path.isdir(path) | aber si una ruta es un directorio |
| os.path.islink(path) | Saber si una ruta es un enlace simbólico |
| os.path.ismount(path) | Saber si una ruta es un punto de montaje |

## Expresiones Regulares

Una expresión regular es una secuencia especial de caracteres que ayuda a encontrar otras cadenas o conjuntos de cadenas que coincidan con patrones específicos; es un lenguaje poderoso para hacer coincidir patrones de texto.

Se pueden utilizar diferentes sintaxis de expresiones regulares para extraer datos de archivos de texto, XML, JSON, contenedores HTML, etc. La siguientes tabla muestra algunas sintaxis de expresiones regulares de Python.

| Patrón | Descripción |
|---|---|
| ^ | Coincide con el comienzo de la línea |
| $ | Coincide con el final de la línea |
| . | Coincide con cualquier carácter individual excepto una nueva línea |
| [...] | Coincide con cualquier carácter individual entre paréntesis |
| [^...] | Coincide con cualquier carácter individual que no esté entre corchetes |
| re* | Coincide con cero o más apariciones de la expresión anterior |
| re+ | Coincide con una o más ocurrencias de la expresión anterior |
| re? | Coincide con cero o una aparición de la expresión anterior |
| re{n} | Coincide exactamente con n número de apariciones de la expresión anterior |
| re{n,} | Coincide con no más apariciones de la expresión anterior |
| re{n,m} | Coincide con al menos n y como máximo m apariciones del expresión precedente |
| a\|b | Coincide con a o b |
| (re) | Agrupa expresiones regulares y recuerda el texto coincidente |
| (?imx) | Alterna temporalmente entre las opciones i, m o x dentro de una expresión regular |
| (?-imx) | Desactiva temporalmente las opciones i, m o x dentro de una expresión regular |
| (?: re) | Agrupa expresiones regulares sin recordar el texto coincidente |
| (?imx: re) | Alterna temporalmente las opciones i, m o x entre paréntesis |
| (?-imx: re) | Desactiva temporalmente las opciones i, m o x entre paréntesis. |
| (?#...) | Comentario. |
| (?= re) | Especifica la posición mediante un patrón. no tiene rango. |
| (?! re) | Especifica la posición usando la negación del patrón. no tiene rango. |
| (?> re) | Coincide con el patrón independiente sin retroceso |
| \w | Coincide con caracteres de palabra |
| \W | Coincide con caracteres que no son palabras |
| \s | Coincide con los espacios en blanco. equivalente a [\ t \ n \ r \ f] |
| \S | Coincide con espacios que no son en blanco |
| \d | Coincide con dígitos. equivalente a [0-9] |
| \D | Coincide con no dígitos |
| \A | Coincide con el comienzo de la cadena |
| \Z | Coincide con el final de la cadena, si existe una nueva línea, coincide justo antes de la nueva línea |
| \z | Coincide con el final de la cadena |
| \G | Los partidos apuntan donde terminó el último partido |
| \b | Coincide con los límites de las palabras cuando están fuera de los corchetes |
| \B | Coincide con los límites que no son de palabras |
| \n, \t, etc. | Coincide con nuevas líneas, retornos de carro, tabulaciones, etc. |
| \1...\9 | Coincide con la enésima subexpresión agrupada |

In [168]:
```
import re
CoursesData = """
    101 COM Computers
    205 MAT Mathematics
    189 ENG English
"""

# Extract all course numbers
Course_numbers = re.findall('[0-9]+', CoursesData)
print (Course_numbers)
# Extract all course codes
Course_codes = re.findall('[A-Z]{3}', CoursesData)
print (Course_codes)
# Extract all course names
Course_names = re.findall('[A-Za-z]{4,}', CoursesData)
print (Course_names)
# define the course text pattern groups and extract
course_pattern = '([0-9]+)\s*([A-Z]{3})\s*([A-Za-z]{4,})'
print(re.findall(course_pattern, CoursesData))
# Matches any character inside
print(re.findall('[a-zA-Z]+', CoursesData))
```

```
['101', '205', '189']
['COM', 'MAT', 'ENG']
['Computers', 'Mathematics', 'English']
[('101', 'COM', 'Computers'), ('205', 'MAT', 'Mathematics'), ('189', 'ENG', 'E
nglish')]
['COM', 'Computers', 'MAT', 'Mathematics', 'ENG', 'English']
```

## Entrada y Salida básica

La entrada básica desde teclado en Python utiliza la función incorporada `input` . El valor de entrada siempre se asume como una cadena de caracteres, que posteriormente podrá convertirse a otro tipo de datos, como entero utilizando la función integrada `int` o número en coma flotante con `float` .

In [169]:
```
nombre = input('Dame tu nombre: ')
print(nombre, type(nombre))
edad_txt = input('Dame tu edad: ')
edad = int(edad_txt)
print(edad, type(edad))
```

```
Dame tu nombre: Pepe
Pepe <class 'str'>
Dame tu edad: 90
90 <class 'int'>
```

Hay varias formas de presentar la salida de un programa. Los datos pueden imprimirse en un formato legible para las personas, o escribirse en un archivo para uso futuro. Este apartado muestra algunas de las posibilidades.

Hasta ahora hemos encontrado dos formas de escribir valores: las declaraciones de expresión y la función `print` . A menudo, querrá más control sobre el formato de su salida que simplemente imprimiendo valores separados por espacios. Hay varias formas de formatear la salida.

Para usar literales de cadena con formato, comience una cadena con `f` o `F` antes de la comilla de apertura o de la comilla triple. Dentro de esta cadena, puede escribir una expresión de Python entre los caracteres `{` y `}` que pueden referirse a variables o valores literales.

```
In [170]: anio = 2016
          evento = 'Referendum'
          f'Resultados de {evento} de {anio}'
Out[170]: 'Resultados de Referendum de 2016'
```

El método de cadenas `str.format` requiere más esfuerzo manual. Seguirá utilizando `{` y `}` para marcar donde se sustituirá una variable y puede proporcionar directivas de formato detalladas, pero también deberá proporcionar la información a formatear.

```
In [171]: votos_si = 42_572_654
          votos_no = 43_132_495
          porcentaje = votos_si / (votos_si + votos_no)
          '{:-9} votos SI, un {:2.2%}'.format(votos_si, porcentaje)
Out[171]: ' 42572654 votos SI, un 49.67%'
```

Cuando no necesita resultados sofisticados, pero solo desea una visualización rápida de algunas variables para fines de depuración, puede convertir cualquier valor en una cadena con las funciones `repr` o `str`.

La función `str` está diseñada para devolver representaciones de valores que son bastante legibles para las personas, mientras que `repr` está diseñada para generar representaciones que pueden ser leídas por el intérprete (o forzarán un error de sintaxis si no hay una sintaxis equivalente). Para los objetos que no tienen una representación particular para el consumo humano, `str` devolverá el mismo valor que `repr`.

```
In [172]: s = 'Hola Mundo...'
          str(s)
Out[172]: 'Hola Mundo...'

In [173]: repr(s)
Out[173]: "'Hola Mundo...'"
```

## Usando el intérprete de Python

El intérprete de Python generalmente se instala como `/usr/local/bin/python3.7` en aquellas máquinas donde está disponible; al poner `/usr/local/bin` en la ruta de búsqueda de su shell de Unix es posible iniciarlo escribiendo el comando:

python

```
$ python
Python 3.7 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>
```

Dado que la elección del directorio donde vive el intérprete es una opción de instalación, otros lugares son posibles; Consulte con su gurú de Python local o administrador del sistema. (Por ejemplo, `/usr/local/python` es una ubicación alternativa popular).

En las máquinas Windows en las que haya instalado desde Microsoft Store, el comando `python3.7` estará disponible. Si tiene instalado el iniciador `py.exe` , puede usar el comando `py` .

Si se escribe un carácter de final de archivo ( `Control-D` en Unix, `Control-Z` en Windows) en el indicador primario, el intérprete sale con un estado de salida cero. Si eso no funciona, puede salir del intérprete escribiendo el siguiente comando: `quit()` .

### Scripts Python ejecutables

En los sistemas BSD Unix, los scripts de Python se pueden hacer directamente ejecutables, como los scripts de shell, poniendo la línea:

```
#!/usr/bin/env python3.7
```

(asumiendo que el intérprete se encuentra en la RUTA del usuario) al comienzo de la secuencia de comandos y le da al archivo un modo ejecutable. Los `#!` Deben ser los dos primeros caracteres del archivo. En algunas plataformas, esta primera línea debe terminar con un final de línea de estilo Unix ( `'\ n'` ), no con un final de línea de Windows ( `'\ r \ n'` ). Tenga en cuenta que el carácter hash o pound, `'#'` , se usa para iniciar un comentario en Python.

El script puede recibir un modo ejecutable, o permiso, mediante el comando chmod.

```
$ chmod +x myscript.py
```

En los sistemas Windows, no hay noción de un "modo ejecutable". El instalador de Python asocia automáticamente los archivos .py con python.exe, de modo que un doble clic en un archivo de Python lo ejecutará como un script. La extensión también puede ser `.pyw` , en ese caso, se suprime la ventana de la consola que aparece normalmente.

### La variable especial `__name__` en Python

Como no hay una función `main` en Python, cuando se le da al intérprete el comando para ejecutar un programa de Python, se debe ejecutar el código que está en el nivel 0 de sangría. Sin embargo, antes de hacerlo, definirá algunas variables especiales. `__name__` es una de esas variables especiales. Si el archivo fuente se ejecuta como el programa principal, el intérprete configura la variable `__name__` para que tenga un valor `"__main__"` . Si este archivo se importa desde otro módulo, `__name__` se establecerá con el nombre del módulo.

`__name__` es una variable integrada que evalúa el nombre del módulo actual. Por lo tanto, se puede usar para verificar si el script actual se está ejecutando solo o si se está importando en otro lugar combinándolo con la declaración `if` , como se muestra a continuación.

```
# contenido de myscript.py
if __name__ == "__main__":
    print("El fichero se está ejecutando directamente")
else:
    print ("El fichero se está importando")
```

## IPython

IPython es un shell interactivo que añade funcionalidades extra al modo interactivo incluido con Python, como resaltado de líneas y errores mediante colores, una sintaxis adicional para el shell, autocompletado mediante tabulador de variables, módulos y atributos; entre otras funcionalidades. IPython proporciona una arquitectura rica para la computación interactiva con:

- Una potente shell interactiva.
- Un núcleo para Jupyter.
- Soporte para visualización de datos interactiva y uso de kits de herramientas GUI.
- Intérpretes flexibles e integrables para cargar en sus propios proyectos.
- Herramientas fáciles de usar y de alto rendimiento para computación paralela.

In [174]:
```
# !dir en Windows
!ls
```

```
cachedir            docs                joblib.ipynb       __pycache__
dask-worker-space   ejercicios          matplotlib.ipynb   python.ipynb
data                images              mprun_demo.py      python_poo.ipynb
data-pandas.ipynb   intro-pandas.ipynb  numpy.ipynb
```

In [175]:
```
# !cd en Windows
!pwd
```

```
/workspace/Curso Python
```

### Pasar valores hacia y desde el Shell

Los comandos de shell no solo pueden invocarse desde IPython, sino que también pueden interactuar con el espacio de nombres de IPython.

In [176]:
```
listado = !ls *.ipynb
print(listado)
```

```
['data-pandas.ipynb', 'intro-pandas.ipynb', 'joblib.ipynb', 'matplotlib.ipynb
', 'numpy.ipynb', 'python.ipynb', 'python_poo.ipynb']
```

In [177]:
```
type(listado)
```

Out[177]:
```
IPython.utils.text.SList
```

> **Nota**: Los resultados no se devuelven como listas, sino como un tipo de retorno de shell especial definido en IPython, que se parece mucho a una lista de Python, pero tiene funcionalidades adicionales como los métodos `grep` y `fields` y las propiedades `s`, `n` y `p` que permiten buscar, filtrar y mostrar los resultados de manera avanzada.

La comunicación en la otra dirección, pasar variables de Python al shell, es posible a través de la sintaxis `{varname}`:

In [178]:
```
patron = "*.ipynb"
!ls {patron}
```

```
data-pandas.ipynb   joblib.ipynb      numpy.ipynb   python_poo.ipynb
intro-pandas.ipynb  matplotlib.ipynb  python.ipynb
```

Adicionalmente IPython contiene una versión de los comandos de shell que le permiten comportarse como una propia shell: `%cd` , `%cat` , `%cp` , `%env` , `%ls` , `%man` , `%mkdir` , `%more` , `%mv` , `%pwd` , `%rm` , y `%rmdir` . Cualquiera de estos comando *mágicos* se puede usar sin el caracter `%` si el parámetro `automagic` se configura en `on` .

```
In [179]: %ls *.ipynb
```

```
data-pandas.ipynb*   joblib.ipynb       numpy.ipynb   python_poo.ipynb
intro-pandas.ipynb*  matplotlib.ipynb*  python.ipynb
```

**Depuración**

La herramienta estándar de Python para la depuración interactiva es `pdb` , el depurador de Python. Este depurador permite al usuario recorrer el código línea por línea para ver qué podría estar causando un error más difícil. La versión mejorada de IPython de esto es `ipdb` , el depurador de IPython. En IPython, quizás la interfaz más conveniente para la depuración es el comando mágico `%debug` . Si se llama después de hacer pulsar en una excepción, se abrirá automáticamente un mensaje de depuración interactivo en el punto de la excepción. El prompt `ipdb` le permite explorar el estado actual de la pila, explorar las variables disponibles e incluso ejecutar comandos de Python. También podemos subir ( `up` ) y bajar ( `down` ) por la pila y explorar los valores de las variables. La siguiente tabla muestra algunos de los comandos más habituales:

| Comando | Descripción |
|---|---|
| list | Muestra la ubicación actual en el archivo |
| h(elp) | Muestra una lista de comandos o busca ayuda sobre un comando específico |
| q(uit) | Salir del depurador y del programa |
| c(ontinue) | Salir del depurador; continuar en el programa |
| n(ext) | Ir al siguiente paso del programa |
| \ | Repite el comando anterior |
| p(rint) | Imprimir variables |
| s(tep) | Paso a una subrutina |
| r(eturn) | Salir de una subrutina |

In [180]:
```
func2(1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-180-7cb498ea7ed1> in <module>
----> 1 func2(1)
        global func2 = <function func2 at 0x7fc1ade525e0>

<ipython-input-160-a62b6a523a8f> in func2(x=1)
      5     a=x
      6     b=x-1
----> 7     return func1(a, b)
        global func1 = <function func1 at 0x7fc1ade52820>
        a = 1
        b = 0
      8
      9 func2(1)

<ipython-input-160-a62b6a523a8f> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a/b
        a = 1
        b = 0
      3
      4 def func2(x):
      5     a=x

ZeroDivisionError: division by zero
```

In [181]:
```
%debug
```

```
> <ipython-input-160-a62b6a523a8f>(2)func1()
      1 def func1(a, b):
----> 2     return a/b
      3
      4 def func2(x):
      5     a=x

ipdb> p(a)
1
ipdb> p(b)
0
ipdb>
0
ipdb> r
```

**Nota**: Si se desea que el depurador se inicie automáticamente cada vez que se genera una excepción, puede usar la función mágica `%pdb` para activar este comportamiento automático.

#### Temporización y profiling

Una vez que tenemos el código funcionando, puede ser útil profundizar un poco en su eficiencia. A veces es útil comprobar el tiempo de ejecución de un comando o conjunto de comandos dado; otras veces es útil profundizar en un proceso de varias líneas y determinar dónde se encuentra el cuello de botella. IPython proporciona acceso a una amplia gama de funciones para este tipo de sincronización y creación de perfiles de código. Entre ellas destacan:

- `%time` : Tiempo de ejecución de una sola instrucción
- `%timeit` : Tiempo de ejecución repetida de una sola declaración para mayor precisión
- `%prun` : Ejecuta código con el generador de perfiles
- `%lprun` : Ejecuta código con el generador de perfiles línea por línea
- `%memit` : Mide el uso de memoria de una sola declaración
- `%mprun` : Ejecutar código con el perfilador de memoria línea por línea

> **Nota**: Los últimos cuatro comandos no están incluidos con IPython, necesitan de la instalación de las extensiones `line_profiler` y `memory_profiler` .

In [182]:
```
%time sum(range(10000))
```

```
CPU times: user 245 µs, sys: 0 ns, total: 245 µs
Wall time: 247 µs
```

Out[182]:
```
49995000
```

In [183]:
```
%timeit sum(range(10000))
```

```
304 µs ± 19.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

> **Nota**: Haciendo uso de un único caracter `%` se ejecuta el comando mágico a nivel de línea, haciendo uso de dos caracteres `%%` se ejecuta a nivel de celda (scripts multilínea).

In [184]:
```
%%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i*(-1)**j
```

```
404 ms ± 8.47 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

A veces, es interesante poder cronometrar los tiempos de las diferentes declaraciones de un bloque de instrucciones en el contexto y no a nivel individual. Python contiene un generador de perfiles de código incorporado, pero IPython ofrece una forma mucho más conveniente de usar este generador de perfiles, en la forma de la función mágica `%prun` .

In [185]:
```
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j^(j>>i) for j in range(N)]
        total += sum(L)
    return total
```

In [186]:
```
%prun sum_of_lists(1000000)
```

```
14 function calls in 0.925 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    0.810    0.162    0.810    0.162 <ipython-input-2-604d7a3a485b>:4(<lis
tcomp>)
        1    0.057    0.057    0.910    0.910 <ipython-input-2-604d7a3a485b>:1(sum_
of_lists)
        5    0.043    0.009    0.043    0.009 {built-in method builtins.sum}
        1    0.015    0.015    0.925    0.925 <string>:1(<module>)
        1    0.000    0.000    0.925    0.925 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profile
r' objects}
```

El perfil función por función de `%prun` es útil, pero a veces es más conveniente tener un informe de perfil línea por línea. Esta funcionalidad no está integrada en Python ni en IPython, pero hay un paquete `line_profiler` disponible para la instalación que la proporciona. Los pasos a realizar para utilizar la funcionalidad son:

1. Instalar el paquete `line_profiler` :

   ```
   $ pip install line_profiler
   ```

2. Cargar la extensión `line_profiler` , que se ofrece como parte de este paquete:

   ```
   %load_ext line_profiler
   ```

3. Utilizar el comando mágico `%lprun` :

```
In [187]: %load_ext line_profiler
```

```
In [188]: %lprun -f sum_of_lists sum_of_lists(5000)
```

```
Timer unit: 1e-06 s

Total time: 0.007568 s
File: <ipython-input-2-604d7a3a485b>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1          2.0      2.0      0.0      total = 0
     3         6          5.0      0.8      0.1      for i in range(5):
     4         5       7362.0   1472.4     97.3          L=[j^(j>>i) for j in range
(N)]
     5         5        199.0     39.8      2.6          total += sum(L)
     6         1          0.0      0.0      0.0      return total
```

Otro aspecto de la creación de perfiles es la cantidad de memoria que utiliza una operación. Esto se puede evaluar con otra extensión de IPython, `memory_profiler` . Los pasos a realizar para utilizar la funcionalidad son:

1. Instalar el paquete `memory_profiler` :

   ```
   $ pip install memory_profiler
   ```

2. Cargar la extensión `memory_profiler` , que se ofrece como parte de este paquete:

   ```
   %load_ext memory_profiler
   ```

3. Utilizar el comando mágico `%memit` :

In [189]:
```
%load_ext memory_profiler
```

In [190]:
```
%memit sum_of_lists(1000000)
```

```
peak memory: 4020.48 MiB, increment: 65.40 MiB
```

Para una descripción línea por línea del uso de la memoria, podemos usar el comando mágico `%mprun` . El único incoveniente es que solo para funciones definidas en módulos separados, no se puede utilizar directamente desde el notebook.

Esta situación se solventa creando un módulo python con la funcionalidad a evaluar:

In [191]:
```
%%file mprun_demo.py
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j^(j>>i) for j in range(N)]
        total += sum(L)
        del L # remove reference to L
    return total
```

```
Overwriting mprun_demo.py
```

Y posteriormente importar la función y ejecutar el comando mágico:

In [192]:
```
from mprun_demo import sum_of_lists
%mprun -f sum_of_lists sum_of_lists(100000)
```

```
Filename: /workspace/Curso Python/mprun_demo.py

Line #    Mem usage    Increment  Occurences   Line Contents
=============================================================
     1     89.1 MiB     89.1 MiB           1   def sum_of_lists(N):
     2     89.1 MiB      0.0 MiB           1       total = 0
     3     89.1 MiB      0.0 MiB           6       for i in range(5):
     4     89.1 MiB      0.0 MiB       50015           L = [j^(j>>i) for j in range
(N)]
     5     89.1 MiB      0.0 MiB           5           total += sum(L)
     6     89.1 MiB      0.0 MiB           1           del L # remove reference to
L
     7                                             return total
```

> **Nota**: Este gasto de mamoria hay que añadirlo al que utiliza el propio intérprete de Python.