Programación de computadores con Python

(Parte II)

Fundamentos de Computación

Grado en Ingeniería en Tecnologías Industriales

26 de octubre del 2021

Universidad de Cantabria

Son llamadas funciones las instrucciones mostradas a continuación, que ya hemos utilizado:

```
 >>> type(777777777)
<class 'int'>
>>> print('Hola mundo,', 2019)
Hola mundo, 2019
>>> complex(5, 7.8)
(5+7.8j)
>>> nombre = input()

>>> range(1, 24, 5)
range(1, 24, 5)
```

Una función se invoca escribiendo su nombre o identificador seguido de los ARGUMENTOS o PARÁMETROS de la función (que pueden ser varios, uno o ninguno) entre PARENTESIS y separados por una coma.

```
def positivo(x):
    '''
Entrada: x, un numero real, class float
Devuelve el booleanoTrue si es positivo, False en caso contrario
'''
    print('Instrucciones dentro de la funcion positivo')
    return x > 0
```

- La primera línea de la definición de una función contiene la palabra reservada def seguida del nombre de la función y entre paréntesis los argumentos (cero o más) y finalmente los dos puntos :
- Lo que está encerrado entre comillas triples es la **documentación** de la función (opcional), pero recomendable.
- Las instrucciones que forman la función se escriben con sangría con respecto a la primera línea, denominado el **cuerpo** de la función.
- Si durante la ejecución de una función se alcanza la instrucción return esta ejecución se termina y el intérprete utiliza el valor indicado.

```
def positivo(x):
    '''
Entrada: x, un numero real, class float
Devuelve el booleanoTrue si es positivo, False en caso contrario
'''
    print('Instrucciones dentro de la funcion positivo')
    return x > 0
```

```
>>> positivo(0)
Instrucciones dentro de la funcion positivo
False
>> positivo(3.14)
Instrucciones dentro de la funcion positivo
True
>>> positivo('pi')
Instrucciones dentro de la funcion positivo
Traceback (most recent call last):
.....
TypeError: '>' not supported between instances of 'str' and 'int
```

```
fact = 6; i = 'Peña la Milana'; n = 20
def factorial(n):
    ''' Calcula el factorial de un numero entero, si es negativo
   su valor es -1 y si es cero  1 '''
    fact = 1
    if n < 0:
        return -1
    for i in range(1, n+1):
        fact *= i
    return fact
```

Variables locales y globales

- Variables **locales**, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre. Por ejemplo, las variable fact y i introducidas en el cuerpo de la función de arriba.
- Variables **globales**, si a la variable se le ha asignado valor en el programa principal. Por ejemplo, las variables fact, i y n en el programa de arriba

```
fact = 6; i = 'Peña la Milana'; n = 20
def factorial(n):
    ''' Calcula el factorial de un numero entero, si es negativo
   su valor es -1 y si es cero  1 '''
    fact = 1
    if n < 0:
        return -1
    for i in range(1, n+1):
        fact *= i
    return fact
```

```
>>> f5 = factorial(5)
>>> f5 + factorial(1)
121
>>> i
'Peña la Milana'
>>> fact
6
>>> n
20
```

```
def positivo1(x):
    '''
Entrada: x, un numero real, class float
Devuelve None
'''
    print("El programa ha llegado, hasta la linea 10")
    x > 0
```

None

Devuelve el valor None si no aparece en el cuerpo de la función la sentencia return

```
>>> f = positivo1(3)
El programa ha llegado, hasta la linea 10
>>> type(f)
<class 'NoneType'>
>>> f
>>>
```

```
def representacion_numero_en_base():
    ''' Devuelve la lista con  la representacion de un numero n positivo
        en una base b, ambas leidas por teclado '''
    print('Dame el numero: ', end = ' ')
    n = int(input())
    print('Dame la base: ',  end = ' ')
    b = int(input())
    L = []
    if b < 2:
        return L
    while b <= n:
        L =  [n % b] + L
        n = n // b
    return [n]+ L
```

return vs print

- return solo tiene significado dentro de la función, solo se ejecuta una vez y tiene un valor asociado.
- La función print puede usarse fuera del cuerpo de la función, ejecutarse varias veces dentro del cuerpo de la función.

```
>>> representacion_numero_en_base()
Dame el numero:  12
Dame la base:  2
[1, 1, 0, 0]
>>> a = representacion_numero_en_base()
Dame el numero:  2019
Dame la base:  1024
>>> a
[1, 995]
>>> representacion_numero_en_base()
Dame el numero:  2019
Dame la base:  -2
[]
>>> type(representacion_numero_en_base)
<class 'function'>
```

```
def rep_base(n, b):
    '''Devuelve la cadena con la representacion
     de el numero n en la base b '''
    l =''
    while n != 0:
        l = str(n%b)+l
        n = n // b
    return l
```

```
>>> rep_base(18,2)
'10010'
>>> bin(18)[2:] == rep_base(18,2)
True
```

```
def letras(palabra, letra='a'):
    ''' Entrada una cadena y una letra (por defecto la letra a)
Devuelve el numero de veces que la cadena contiene la letra '''
    i = 0
    for x in palabra:
        if x == letra:
            i += 1
    return i
```

```
>>> letras('abracadabra')
5
>>> letras('abracadabra', 'b')
2
>>> letras('abracadabra',1)
0
```

Los argumentos de la función pueden estar fijos. En el ejemplo el segundo argumento por defecto es la letra a, pero podemos invocar la función con otro argumento.

```
def func_a():
    print('funcion_a')
    return 8
def func_b(y):
    print('funcion_b')
    return y
def func_c(z):
    print('funcion_c')
    return z()
```

```
>>> print(5+func_b(2))
funcion_b
7
>>> print(func_c(func_a))
funcion_c
funcion_a
8
```

Los argumentos pueden ser objetos de cualquier clase, incluso de la clase FUNCTION, funciones.

Un principio útil para diseñar un programa consiste en dividir la tarea que se quiere alcanzar en procesos más sencillos e independientes que, combinados, permiten resolver el problema: **modularidad**.

$$coseno(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

```
def coseno(x, m = 100):
    cos = 0
    for j in range(m+1):
        cos += (-1)**j * x**(2 * j) / factorial(2 * j)
    return cos
```

- Los valores/datos (números, cadenas, listas, etc.) se guardan en objetos.
  1. Objetos inmutables (no se pueden modificar/mutar) como enteros, cadenas o tuplas.
  2. Objetos mutables (se pueden modificar/mutar) como listas y diccionarios.
- Todo objeto tiene un identificador `id`.

```
>>> id(2019)
4330880944
>>> n = 2019; id(n)
4330880944
>>> L1 = [2,4,8,16]
>>> id(L1)
4331182024
```

```
>>> L = [0] + L1; L
[0, 2, 4, 8, 16]
>>> id(L)
4331182408
>>> L2 = L1
>>> L2
[2, 4, 8, 16]
```

```
>>> id(L2)
4331182024
>>> L1[0] = 1
>>> L2
[1, 4, 8, 16]
>>> id(L2)
4331182024
```

- L y L1 son distintos objetos, distinto identificador. Sin embargo, L2 es un objeto idéntico a L1. Se dice que L2 es un apodo (un alias) de L1.
- En la última columna L1 se ha transformado(mutado) a [1, 2, 4, 8, 16].

A continuación mostramos otras instrucciones con o sobre listas:

- L.append(x) Añade el elemento x al final de la lista L.
- L.insert(i,x) Inserta el elemento x en la posición i. El primer argumento es el índice del elemento delante del cual se insertará.
- L.remove(x) Elimina el elemento x de la lista L. Produce un error si no existe tal elemento.
- del(L[i]) Permite eliminar el elemento $L[i]$ de la lista L o varios elementos a la vez, e incluso la misma lista.

L= [2, 1, 3, 6, 3, 7]

- L.append(0) $---> $ se muta a $L = [2, 1, 3, 6, 3, 7, 0]$
- L.remove(3) $---> $ se muta a $[2, 1, 6, 3, 7, 0]$
- del(L[2]) $----->$ se muta a $[2, 1, 3, 7, 0]$
- L.insert(1,100) $-->$ se muta a $[2, 100, 1, 3, 7, 0]$
- del(L[1:3]) $-->$ se muta a $[2, 3, 7, 0]$

```
>>> M1 = [x**2 for x in [0,1,2,3]]
>>> M1
[0, 1, 4, 9]
>>> id(M1)
4320805448
>>> M2 = M1 + [16]
 M2
[0, 1, 4, 9, 16]
```

```
>>> id(M2)
4320815496
>>> M1.append(16); M1
[0, 1, 4, 9, 16]
>>> id(M1)
4320805448
>>> M1 == M2
True
```

- Observar la diferencia M1.append(x), con concatenar listas : M1+[x],
- Podemos construir listas por comprensión, como la lista M1 o como:

matriz_identitad.py

```
def matriz_unidad(n):
    In = [ [0 for x in range(n)] for x in range(n)]
    for i in range(n):
        In[i][i] = 1
    return In
```

Dadas dos listas L1 y L2, queremos eliminar los elementos de L1 que están en L2:

```
def quitar_comunes1(L1,L2):
    for x in L1:
        if x in L2:
            L1.remove(x)
    return L1
```

```
def quitar_comunes2(L1,L2):
    L1c=L1[:]
    for x in L1c:
        if x in L2:
            L1.remove(x)
    return L1
```

```
>>> quitar_comunes1([1,2,3,4], [1,2,5,6])
[2, 3, 4]
quitar_comunes2([1,2,3,4], [1,2,5,6])
[3, 4]
```

La lista `L1c = L1[:]` es una clonación de la lista `L1`.

Queremos almacenar la información de los clientes de una compañía de seguros de vehículos (nombre, fecha del carnet de conducir, número de poliza, etc.).

```
list_nombre = ['Boole', 'Neumann', 'Torres', 'Turing']
list_fecha = [ '2/1987', '5/2011', '4/1765', '10/1959]
list_poliza = [ 20.83, 36.8, 211.334,  981.45]
```

- Una lista para cada pieza de información.
- Todas las listas deben tener la misma longitud.
- La información almacenada en listas en el mismo índice.
- Actualizar y recuperar la información del cliente:

```
def info_cliente(cliente, list_nombre, list_fecha, list_poliza):
    i = list_nombre.index(cliente)
    fecha = list_fecha[i]
    poliza = list_poliza[i]
    return  (fecha, poliza)
```

```
>>> A = [1, 'H', 3.5, (True, list)]
>>> A.index('H')
1
```

Incovenientes

- Desordenado para realizar un seguimiento, si tiene un montón de información diferente.
- Debe mantener muchas listas, pasarlas como argumentos y siempre debe indexar usando enteros.

Alternativamente, podríamos implementar esto en una sólo lista `clientes`:

```
clientes = [('Boole',['2/1987',20.83]),('Neumann',['5/2011', 36.8])
('Torres', ['4/1765', 211.334]), ('Turing',['10/1959',981.45])
```

- Donde cada elemento de una lista es una tupla con el formato (nombre, información).
- Para obtener información sobre el nombre necesitaríamos averiguar el índice y luego usar `clientes [1]`
  - **Incoveniente:** Muy lento para encontrar el índice, si la lista es muy grande

Un diccionario es una asignación entre un conjunto de índices (claves) y un conjunto de valores. Cada clave se asigna a un valor. La asociación de una clave y un valor se denomina par `clave : valor` o un elemento.

- Un diccionario es una secuencia de elementos del tipo `clave : valor` encerrados entre llaves y separados por comas:

  `{clave0: valor0, clave1: valor1,.....}`
- Los diccionarios son de la clase `<class 'dict'>`
- Para extraer un valor asociado a una clave: `nombre_diccionario[clave]`

```
>>> clientes = {'Boole': ['2/1987',20.83] , \
'Neumann' : ['5/2011', 36.8], 'Torres' :  ['4/1765', 211.334], \
 'Turing' : ['10/1959',981.45]}
>>> type(clientes)
<class 'dict'>
>>> clientes['Torres']
['4/1765', 211.334]
>>> id(clientes)
4302132928
```

- Las claves deben ser objetos inmutables, pero los valores pueden ser cualquier objeto. Las claves deben ser únicas, pero no necesariamente los valores.
- Los diccionarios son objetos **mutables**, podemos modificar y añadir elementos: `clientes[clave] = valor`
- No hay ningún orden para las claves y los valores.
- Para saber si una clave esta en el diccionario: `'Torres' in clientes`
- Para eliminar un elemento del diccionario: `del(clientes['Boole'])`

```
>>> clientes['Gauss'] = ['11/1578', 78.6]
>>> clientes['Turing'] = ['10/1959',100.2]
>>> 'Torres' in clientes
True
>>> del(clientes['Boole'])
>>> clientes
{'Neumann': ['5/2011', 36.8], 'Torres': ['4/1765', 211.334],
'Turing': ['10/1959', 100.2], 'Gauss': ['11/1578', 78.6]}
>>> id(clientes)
4302132928
```

A continuación mostramos otras instrucciones con o sobre diccionarios D:

- D.keys() y D.values() devuelve las claves y los valores (respectivamente).
- D.items() devuelve los elementos (clave, valor).

Los tres métodos son iterables

```
>>> D = {'uno':'one', 'dos':'two', 'tres':'three'}
>>> D.keys(); D.values()
dict_keys(['uno', 'dos', 'tres'])
dict_values(['one', 'two', 'three'])
>>> D.items()
dict_items([('uno', 'one'), ('dos', 'two'), ('tres', 'three')])
>>> for x in D.items():
...   print(x)
...
('uno', 'one')
('dos', 'two')
('tres', 'three')
```

Queremos contar cuántas veces aparece cada letra en una cadena: `histograma`.

- Podríamos crear la lista **letras** con las 27 variables, una para cada letra del alfabeto. Luego, recorremos la cadena y, para cada carácter que este en la cadena, incrementamos el contador correspondiente.

```
def histograma(c):
    b = []
    for x in letras:
        j = 0
        for a in c:
            if x == a:
                j += 1
        if j > 0:
           b.append((x,j))
    return b
```

- Podríamos crear un diccionario con caracteres como claves y contadores como los valores correspondientes. La primera vez que vea un carácter, agregará un elemento al diccionario. Después de eso, incrementaría el valor de un elemento existente.

```
def histograma(c):
    d = {}
    for x in c:
        if x not in d:
            d[x] = 1
        else:
            d[x] += 1
     return d
```

Cada una de estas opciones realiza el mismo cálculo, pero cada una de ellas implementa ese cálculo de una manera diferente.

Usando listas hemos recorrido la cadena 27 veces, con diccionarios una.

Programación segura/defensiva

Garantizar el comportamiento y la comprensión del programa

- Diseñar el programa en modulos/funciones.
- Escribir especificaciones para las funciones.
- Comprobar las condiciones de entrada/salida

Probando/Testing

- Probar muchas entradas/salidas
- Intentar probar que el programa no es correcto
- ¿ No funciona ?

Depurando/Debugging

- Analizar la situación del error.
- ¿Por qué no funciona ?
- ¿Cómo puedo arreglar el programa ?

- Desde el principio diseñar el código para facilitar esta importante tarea.
- Probar y depurar las funciones del programa individualmente
- Documentar las restricciones en las entradas y salidas.

Probando

- Edsger Dijkstra 2002: PROGRAM TESTING CAN BE USED TO SHOW THE PRESENCE OF BUGS, BUT NEVER TO SHOW THEIR ABSENCE!
- Las pruebas solo pueden aumentar nuestra confianza en la corrección del programa.

¿ Cuando estamos listos para probar un programa ?

- El código se ejecuta:
  - eliminar los errores sintácticos SYNTAXERROR; por ejemplo: `3+ )`.
  - eliminar errores *semánticos* TYPEERROR; por ejemplo `3/'tres'`.
- Tener un conjunto de resultado esperados (Entradas-Salidas), es decir:
  - Conjunto de entradas
  - Para cada entrada la salida esperada.

Probando

- Probar cada trozo(función) del programa separadamente.
- Corregir los errores encontrados y probar de nuevo.
- Probar todo el programa.

Dos estrategias complementarias de pruebas

- Black box testing: elegir los datos de la prueba sin mirar la implementación, solo probar el comportamiento mencionado en la especificación.
- White box testing: elegir datos de las pruebas con conocimiento de la implementación (por ejemplo, pruebe que todas las rutas a través de su código se ejercen y corrigen en bucles, bifurcaciones). Comprobar los denominados **casos frontera**, por ejemplo, listas vacias o con un solo elemento, enteros y floats, cero, números positivos y negativos.

Probando

esPrimo.py

```
def  esPrimo(m):
''' Asumimos que m es un int positivo
Devuelve True si m es primo; False en caso contrario '''
    if m <= 2:
        return  False
    for i in range(2, m // 2+1):
            if m % i == 0:
                return  False
    return True
```

Black box testing:

```
>>> esPrimo(2037)
False
>>> esPrimo(5)
True
```

White box testing:

```
>>> esPrimo(2)
False
```

Afirmaciones: ASSERT

```
>>> assert esPrimo(6) == False
>>> assert esPrimo(2) == True
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    assert esPrimo(2) == True
AssertionError
```

Utilizamos la sentencia `assert` para:

- `assert` ---> `True` no hace nada y la prueba pasa.
- `assert` ---> `False` el programa se bloquea, y devuelve un error.
- Tambien podemos usar `assert` como ayuda a depurar el programa. Puesto que devolverá un error si la afirmación no es cierta.

DEPURAR EL PROGRAMA

Afirmaciones: ASSERT

```
def  esPrimo(m):
''' Asumimos que m es un int positivo
Devuelve True si m es primo; False en caso contrario '''
    # Depurar Cambiar < = por <
    # m <= 2
     if m < 2:
         return  False
     for i in range(2, m // 2 + 1):
             if m % i == 0:
                 return  False
     return True
```

```
>>> esPrimo(2037)
False
>>> esPrimo(2)
True
>>> assert esPrimo(2) == True
```

Depurando

HERRAMIENTAS

- Mensajes de error de PYTHON
- La función `print`
  - Usarla para comprobar los argumentos/paramétros.
  - Usarla para mostrar la salida de trozos de código intermedios.
  - Usarla para indicar el camino recorrido en el programa y, acotar donde se pudo producir el error.
- La sentencia `assert`.
  - Para asegurar que las suposiciones son las correctas.
  - Para generar un error de excepción si no es cierta la afirmación.
- Python Tutor
- Usar la cabeza

Depurando

Uno de los métodos más importantes para ordenar listas fue diseñado en 1945 por John Von Neumann, el creador del ordenador.

Diseña una función `ordena_mezclando` que tenga como argumento dos listas ordenadas cuyos elementos son datos numéricos, y devuelva una única lista con todos sus elementos ordenados. Siguiendo la siguiente estrategia:

Tenemos dos pilas de cartas (las dos listas) con la cara por arriba. Cada pila está ordenada tal que la carta más pequeña es la superior. Queremos mezclar las dos pilas para obtener una única pila ordenada.

- Elegimos la más pequeña entre de las dos cartas superiores de cada pila;
- la quitamos de esa pila y la ponemos en una nueva pila con la cara por abajo;
- repetimos este procedimiento hasta que no haya cartas en alguna de las pilas;
- entonces sólo tenemos que coger el resto de las cartas de la otra pila y ponerlas boca abajo en la nueva pila.

Depurando

ordena_mezclando.py

```
def  ordena_mezclando(izda, dercha):
    ordenada = []
    while len(izda) > 1 and len(dercha) > 1:
        if izda[0] <= dercha[0]:
            ordenada.append(izda[0])
            izda.remove(izda[0])
        else:
            ordenada.append(dercha[0])
            del(dercha[0])
    if len(izda) == 0:
        ordenada += dercha[0:]
    else:
        ordenada += izda[0:]
    return ordenada
```

Depurando

```
>>> ordena_mezclando([5],[])
[5]
>>> assert ordena_mezclando([],[3]) == [3]
>>> >>> ordena_mezclando([],[1,3])
[1, 3]
>>> ordena_mezclando([],[])
[]
>>> ordena_mezclando([-1,4,9,20,70],[2,8,12,70])
[-1, 2, 4, 8, 9, 12, 20, 70]
>>> ordena_mezclando([1,2,3],[1,6])
[1, 1, 2, 3]
>>> assert ordena_mezclando([1,2,3],[5,6]) === [1,2,3,5,6]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert ordena_mezclando([1,2,3],[5,6]) == [1,2,3,5,6]
AssertionError
```

No es correcto. Depurar el programa.

Depurando

Añadimos un `print` al finalizar el bucle `while`

ordena_mezclando.py

```
def  ordena_mezclando(izda, dercha):
    ordenada = []
    while len(izda) > 1 and len(dercha) > 1:
        if izda[0] <= dercha[0]:
            ordenada.append(izda[0])
            izda.remove(izda[0])
        else:
            ordenada.append(dercha[0])
            del(dercha[0])
    print(ordenada)        # Aqui el print añadido
    if len(izda) == 0:
        ordenada += dercha[0:]
    else:
        ordenada += izda[0:]
    return ordenada
```

Depurando

```
>>> ordena_mezclando([1,2],[-1,-1,5])
 [-1, -1]
[-1, -1, 1, 2]
>>> >>> ordena_mezclando([1,2,3],[1,6])
[1, 1]
[1, 1, 2, 3]
>>>
```

El problema esta en el bucle, no recorre todos los elementos de las listas.

Depurando

Modificamos las condiciones del bucle `while`: cambiar `> 1` por `> 0`.

ordena_mezclando.py

```
def  ordena_mezclando(izda, dercha):
    ordenada = []
    while len(izda) > 0 and len(dercha) > 0:
        if izda[0] <= dercha[0]:
            ordenada.append(izda[0])
            izda.remove(izda[0])
        else:
            ordenada.append(dercha[0])
            del(dercha[0])
    print(ordenada)        # Aqui el print añadido
    if len(izda) == 0:
        ordenada += dercha[0:]
    else:
        ordenada += izda[0:]
    return ordenada
```

Depurando

```
>>> ordena_mezclando([1,2],[-1,-1,5])
[-1, -1, 1, 2]
[-1, -1, 1, 2, 5]
>>> ordena_mezclando([1,2,3],[1,6])
 [1, 1, 2, 3]
[1, 1, 2, 3, 6]
>>> >>> assert ordena_mezclando([1,2,3],[5,6]) == [1,2,3,5,6]
>>>
```

Seguimos probando con distintos tipos de listas listas: vacias, con todos los elementos iguales,...

Depurando: excepciones

Las EXCEPCIONES se producen durante la ejecución del programa, debido a una condición inesperada

Algunas excepciones

- `['a','b'][2]` ----------> `Index Error`
- `x` ----------> `NameErrror`
- `3/0` ----------> `ZeroDivisionError`
- `int('a')` ----------> `ValueError`
- `'a'/4` ----------> `TypeError`
- `assert 1 is 1.0` ----------> `AssertionError`
- `'abc'.append('d')` ----------> `AttributeError`

Depurando: controlar las excepciones

```
try:
    a = int(input('Dame un numero: '))
    b = int(input('Dame otro numero: '))
    print('a/b = ', a / b)
    print('a*b = ', a * b)
except ValueError:
    print(' No puedo convertir a un dato numerico.')
except ZeroDivisionError:
    print(' No se puede dividir por cero')
except:
    print('Otros errores.')
```

- Primero, se ejecuta el bloque try (el código entre las declaración try y except).
- Si no ocurre ninguna excepción, el bloque except se salta y termina la ejecución de la declaración try.
- Si ocurre una excepción durante la ejecución del bloque try, el resto del bloque se salta.

Depurando: controlar las excepciones

```
--RESTART: depurar.py
Dame un numero: 5
Dame otro numero: 7
a/b =  0.7142857142857143
a*b =  35
--RESTART: depurar.py
Dame un numero: 5
Dame otro numero: 0
No se puede dividir por cero
--RESTART: depurar.py
Dame un numero: 'a'
No puedo convertir a un dato numerico.
```

Depurando: crear excepciones

```
def ratios(L1, L2):
    ''' Suponemos que  L1 y L2 son listas de igual longitud de
         numeros.
        Devuelve: una lista conteniendo  L1[i]/L2[i] '''
    assert len(L1) == len(L2)
    ratios = []
    for i in range(len(L1)):
        try:
            ratios.append(L1[i]/L2[i])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = Not a Number
        except:
            raise ValueError('error de argumento en ratios')
    return ratios
```

Si alcanza la segunda sentencia except crea la excepción.

Depurando: crear excepciones

```
>>> ratios([1,2],[1])
Traceback (most recent call last):.....
    ratios([1,2],[1])
    assert len(L1) == len(L2)
AssertionError
>>> ratios([2,4],[1,2])
[2.0, 2.0]
>>> ratios([2,4],[1,0])
[2.0, nan]
>>> ratios([2,4],[1,'x'])
TypeError: unsupported operand type(s) for /: 'int' and 'str'
During handling of the above exception, another exception occurred:
Traceback (most recent call last): ....
    raise ValueError('error de argumento en ratios')
ValueError: error de argumento en ratios
```

PEP 8: Style Guide for Python Code

PEP 8 es una guía de estilo que facilita la lectura del código y la consistencia entre programas de distintos usuarios. Esta guía no es de seguimiento obligatorio, pero es muy recomendable.

Principios básicos PEP 8

- Siempre preferir espacios en vez de tabular.
- Usar 4 espacios en el sangrado.
- Las lineas deben tener menos de 80 caracteres.
- En una fila, las funciones y las clases deben estar separadas por dos lineas en blanco.
- Cuando sea posible, poner comentarios en una sola línea.
- Usar documentación, verb.,docstrings,.
- Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: `x = f(1, 2) + h(3, 7.1)`.