Universidad de Cantabria

open course ware

Programación

Tema 4. Descomposición, Abstracción y Funciones

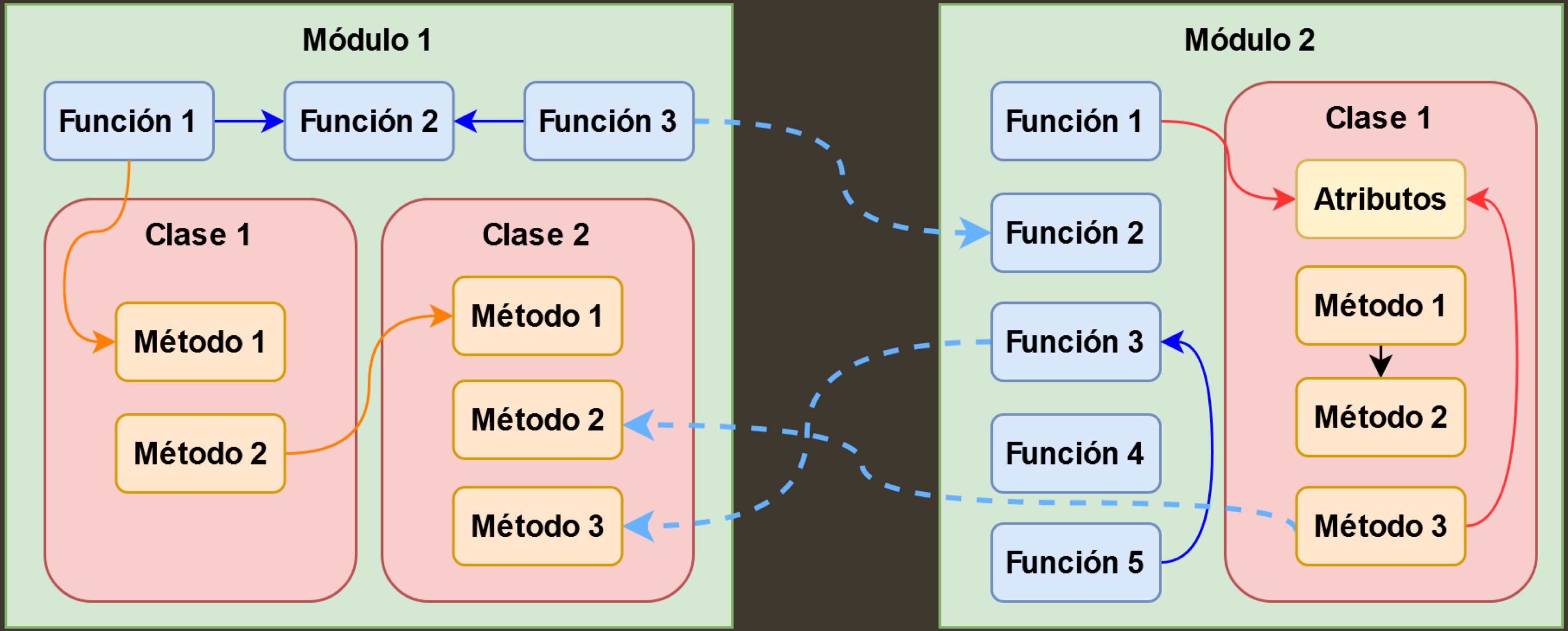
Javier González Villa
David Lázaro Urrutia
Departamento de Matemática Aplicada
y Ciencias de la Computación

Contenidos

1. Descomposición

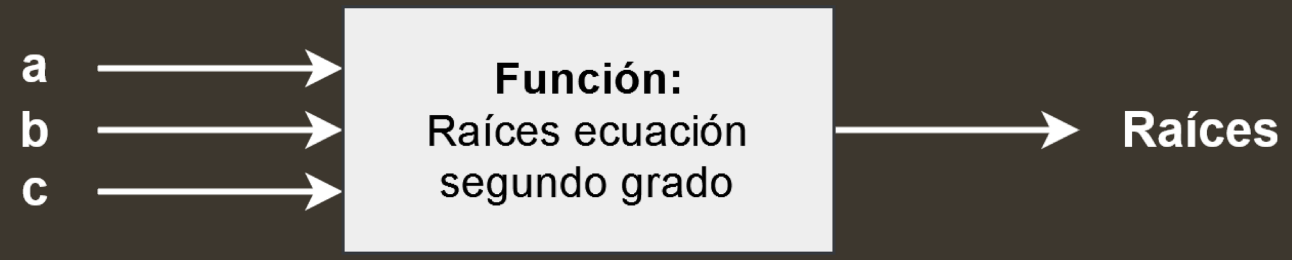
- Característica de la programación que permite dividir el código en módulos.
- Permite fragmentar el código de manera estructurada y coherente dando sentido a los módulos implementados.
- Cada módulo es autocontenido y puede ser utilizado de manera independiente, incrementando su reutilización.
- En proyectos grandes permite tener el código ordenado y estructurado reduciendo el tiempo de actualización de las implementaciones y etapas de validación.
- La descomposición puede realizarse de menor a mayor escala desde funciones pasando por clases y acabando en módulos completos que alojen múltiples clases y funciones.

1. Descomposición



2. Abstracción

- Característica de la programación que permite utilizar módulos o funciones sin necesidad de conocer su implementación.
- Permite utilizar código de terceros solamente entendiendo la documentación.
- Incrementa la reutilización mediante la visión de los módulos o funciones como cajas negras las cuales tan solo tienen unas entradas y producen unas salidas.
- Es inherente a proyectos grandes o librerías con multitud de funciones.



3. Funciones

- Bloques de código autocontenidos y reutilizables que permiten la automatización de cálculos recurrentes.

- El proceso para trabajar con funciones sigue los siguientes pasos:

  1. Creación de la función con la tarea específica que queremos automatizar definiendo los siguientes parámetros:

     - Nombre de la función: debe ser descriptivo de la tarea que realiza dicha función.
     - Argumentos: entradas que recibe la función para realizar los cálculos (si es que los necesita).
     - Documentación: detalles de como usar y los cálculos que realiza la función.
     - Cuerpo de la función: bloque de código que implementa la tarea que queremos que desarrolle la función.
     - Retorno: valor o valores que devuelve la función como resultado de nuestros cálculos.

  2. Llamada o invocación de la función con unos parámetros específicos de entrada.

3. Funciones

1. Creación de la función:

    - Nombre de la función
    - Argumentos
    - Documentación
    - Cuerpo de la función
    - Retorno

2. Consulta de documentación

3. Llamada o invocación de la función

```
[1]: def sol_ecuacion_grado2(a, b, c):
         """
         Input: a, float coeficiente principal
                b, float coeficiente secundario
                c, float termino independiente
         Retorna las raices de la ecuación en caso de que estas existan
         """
         if(a == 0):
             return -c/b
         else:
             discriminante = (b**2)-(4*a*c)
             if(discriminante == 0):
                 return -b/(2*a)
             else:
                 return ((-b+(discriminante**(1/2)))/(2*a)
                         ,(-b-(discriminante**(1/2)))/(2*a))
```

```
[2]: help(sol_ecuacion_grado2)

     Help on function sol_ecuacion_grado2 in module __main__:

     sol_ecuacion_grado2(a, b, c)
         Input: a, float coeficiente principal
                b, float coeficiente secundario
                c, float termino independiente
         Retorna las raices de la ecuación en caso de que estas existan
```

```
[54]: solucion = sol_ecuacion_grado2(0,4,2)
      solucion

[54]: -0.5
```

3. Funciones

```
[13]: import math
      help(math)

Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.
```

```
[14]: import math
      help(math.sin)

Help on built-in function sin in module math:

sin(x, /)
    Return the sine of x (measured in radians).
```

```
[15]: import math
      help(math.pi)

Help on float object:

class float(object)
 |  float(x=0, /)
 |
 |  Convert a string or number to a floating point number, if possible.
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __ceil__(self, /)
 |      Return the ceiling as an Integral.
 |
```

3.1. Definición de funciones

Función: *subalgoritmo que describe una secuencia de órdenes.*

- Estructura:

  def nombre_función(argumentos)

  bloque

  return retorno

```
[53]: def sol_ecuacion_grado2(a,b,c):
          if(a==0):
              return -c/b
          else:
              discriminante = (b**2)-(4*a*c)
              if(discriminante == 0):
                  return -b/(2*a)
              else:
                  return ((-b+(discriminante**(1/2)))/(2*a)
                          ,(-b-(discriminante**(1/2)))/(2*a))
```

- Argumentos: variables que se introducen en la función y son utilizadas en el bloque de código.
- Retorno: variables que retorna la función como resultado de la ejecución del bloque de código.

3.2. Llamadas a funciones

Llamada a función: *invocación de un bloque de código contenido en una función.*

- Estructura:

  retorno = nombre_función(argumentos)

- Para realizar una llamada a función ha de referirse a ella por un nombre declarado previamente.

- Se ha de tener en cuenta los argumentos que requiere y las variables retornadas.

- Permite incrementar el grado de modularidad haciendo nuestro código más legible.

```
[54]: solucion = sol_ecuacion_grado2(0,4,2)
      solucion

[54]: -0.5

[55]: solucion = sol_ecuacion_grado2(2,4,2)
      solucion

[55]: -1.0

[56]: solucion = sol_ecuacion_grado2(1,4,3)
      solucion

[56]: (-1.0, -3.0)
```

3.3. Aclaraciones

- El parámetro abstracto definido dentro de la función no afecta al bloque de código que realiza la llamada.
- Las variables utilizadas dentro de la función se pierden al acabar la ejecución de la misma.
- Si la función no tiene return retorna *None*.
- Una función no puede redefinir variables de fuera de la misma, pero si consultarlas.

[9]:
```python
def suma(x, y):
    res = x + y
    return res

res = 1000
print("Resultado de la suma: ",suma(4, 6))
print("Valor de la variable res: ",res)
```
```
Resultado de la suma:  10
Valor de la variable res:  1000
```

[7]:
```python
def suma(x, y):
    res = x + y

print(suma(4, 6))
```
```
None
```

[12]:
```python
def suma():
    res = x + y
    z = 1000
    return res

x = 100
y = 100
z = 30
print("Resultado de la suma: ",suma())
print("Valor de la variable z: ",z)
```
```
Resultado de la suma:  200
Valor de la variable z:  30
```

4. Modularidad

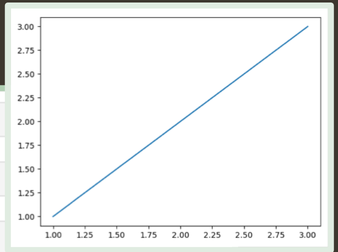
***Modularidad:** característica de un sistema dividido en partes que interactúan entre sí.*

- Estructura:
    - **from** librería **import** función
    - **import** librería

- Si tenemos una librería de funciones que hemos creado previamente, podemos llamar a las funciones de la misma importándolas en nuestro nuevo código de manera individual o en conjunto (*).

- Permiten abstraerse de la implementación de las mismas centrándose solo en los argumentos requeridos y los resultados retornados.

```
•[55]: from math import *

[51]: import math

[52]: from math import cos

[53]: from math import sin, cos

[56]: matplotlib.plot([1,2,3],[1,2,3])

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [56], line 1
----> 1 matplotlib.plot([1,2,3],[1,2,3])

NameError: name 'matplotlib' is not defined

[57]: import matplotlib

[59]: matplotlib.pyplot.plot([1,2,3],[1,2,3])

[59]: [<matplotlib.lines.Line2D at 0x37d3908>]

[60]: matplotlib.pyplot.show()
```



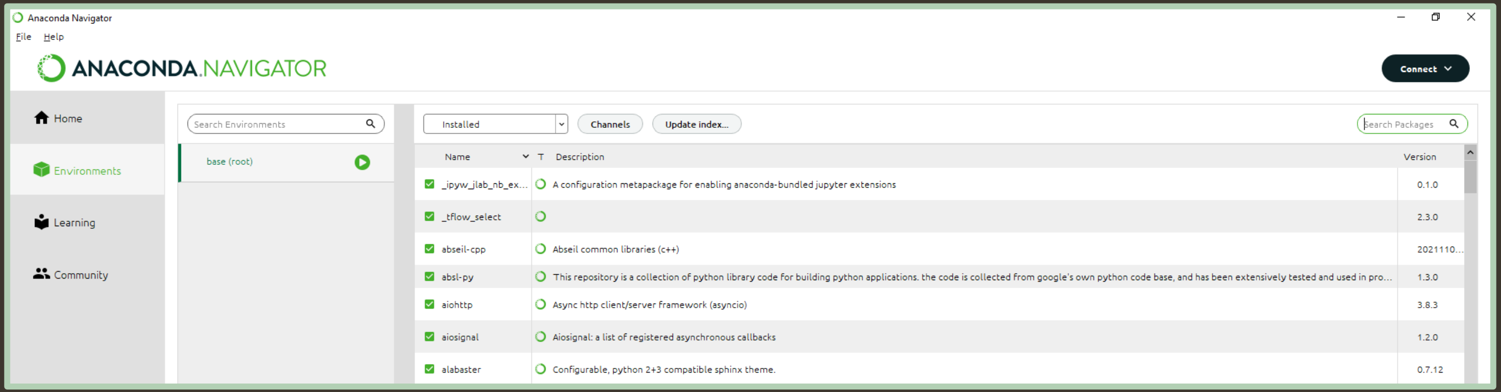
4. Modularidad

> ***Modularidad:*** *característica de un sistema dividido en partes que interactúan entre sí.*

- Librerías por defecto en Python:

  https://docs.python.org/es/3/library/index.html

- Instalación de librerías adicionales de otros desarrolladores:



5. Recursión

- En matemáticas se da el nombre de recursión a la técnica consistente en definir una función en términos de sí misma.
- En computación se llama recursividad a un proceso mediante el que una función se llama a sí misma de forma repetida, haciendo uso del resultado anterior, hasta que se satisface alguna determinada condición.
- Estas funciones deben definir un caso base explícito para alguno de sus argumentos para no caer en posibles bucles infinitos.
- Para implementar una función de manera recursiva se deben satisfacer dos condiciones:

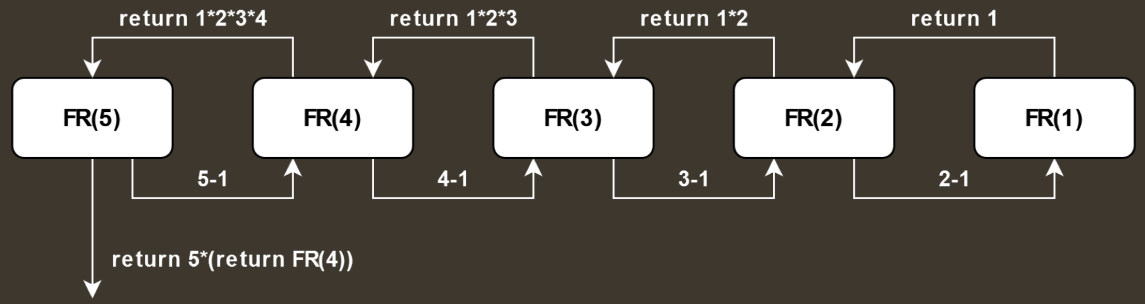
1. El problema debe poder escribirse o plantearse de forma recursiva.
2. El problema debe de tener una condición de fin.

5. Recursión

Recursividad: *técnica que consiste en que una función se invoque a si misma.*

- Es muy importante que la función recursiva tenga una condición de parada, ya que en caso contrario puede existir un bucle infinito.

- Permite dividir las tareas en subtareas más pequeñas que son fáciles de resolver.




```
[88]: def factorial(n):
          resultado = 1
          i = 2
          while(i <= n):
              resultado = resultado * i
              i = i + 1
          return resultado

      factorial(5)

[88]: 120
```

```
[87]: def factorial_recursivo(n):
          if n == 1:
              return 1
          else:
              return n*factorial_recursivo(n-1)

      factorial_recursivo(5)

[87]: 120
```

5.1. Iterativo vs Recursivo

- Iterativo:
  - Los bucles se crean a partir de instrucciones while y for.
  - Puede ser muy difícil encontrar solución iterativa a algunos problemas concretos.
  - El código es más largo y complejo de leer, siendo en algunas ocasiones difícil su comprensión.
  - Carga de computación menor al almacenar los resultados en variables con diferentes estados en cada iteración.

- Recursivo:
  - Se crean usando llamadas decrecientes o divididas a la misma función.
  - Pueden tener problemas de redundancia al realizar llamadas repetidas con el mismo resultado.
  - El código recursivo usualmente suele ser más sencillo y fácil de leer.
  - La carga computacional y de memoria es mucho mayor al tener que almacenar todas las llamadas recurrentes.

5.1. Iterativo vs Recursivo

- Iterativo:

```
[1]: import time
     import sys

     def factorial_iterativo(n):
         resultado = 1
         i = 2
         while(i <= n):
             resultado = resultado * i
             i = i + 1
         return resultado

     inicio = time.time()
     print("Factorial 10: ",factorial_iterativo(10))
     fin = time.time()
     print("Memoria: ",sys.getsizeof(factorial_iterativo)," bytes")
     print("Tiempo: "+str(fin-inicio))

     Factorial 10:  3628800
     Memoria:  136  bytes
     Tiempo: 0.0009975433349609375
```

- Recursivo:

```
[1]: import time
     import sys

     def factorial_recursivo(n):
         if n == 1:
             return 1
         else:
             return n * factorial_recursivo(n-1)

     inicio = time.time()
     print("Factorial 10: ",factorial_recursivo(10))
     fin = time.time()
     print("Memoria: ",sys.getsizeof(factorial_recursivo)*10," bytes")
     print("Tiempo: "+str(fin-inicio))

     Factorial 10:  3628800
     Memoria:  1360  bytes
     Tiempo: 0.0009965896606445312
```

5.1. Iterativo vs Recursivo

- Iterativo:

```
[9]:
import time
import sys

def suma_iterativa(n):
    suma = 0
    i = 0
    while (i < n):
        suma = suma + 1
        i = i + 1
    return suma

inicio = time.time()
print("Suma 2500: ",suma_iterativa(2500))
fin = time.time()
print("Memoria: ",sys.getsizeof(suma_iterativa)," bytes")
print("Tiempo: "+str(fin-inicio))
```

```
Suma 2500:  2500
Memoria:  136  bytes
Tiempo: 0.0
```

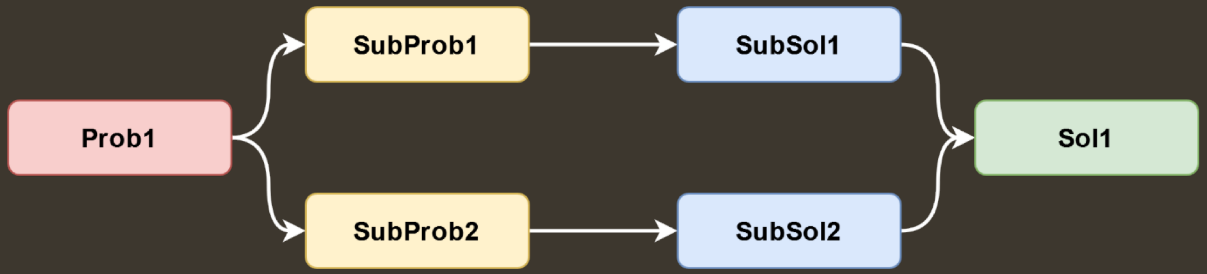
- Recursivo:

```
[8]:
import time
import sys

def suma_recursiva(n):
    if n == 1:
        return 1
    else:
        return 1 + suma_recursiva(n-1)

inicio = time.time()
print("Suma 2500: ",suma_recursiva(2500))
fin = time.time()
print("Memoria: ",sys.getsizeof(suma_recursiva)*2500," bytes")
print("Tiempo: "+str(fin-inicio))
```

```
Suma 2500:  2500
Memoria:  340000  bytes
Tiempo: 0.0
```

5.2. Divide / Decrementa y Vencerás

- La estrategia de Divide y Vencerás consiste en subdividir tantas veces como sea necesario un problema complejo en versiones más sencillas del mismo problema que sí que son posibles de resolver de manera no muy compleja.
- A partir de la resolución de estos subproblemas más sencillos se reconstruye la solución global del problema complejo.
- En computación representa uno de los paradigmas de diseño de algoritmos.
- Las implementaciones recursivas a la hora de resolver problemas están fundamentadas en la estrategia de Divide y Vencerás junto con el principio de inducción matemática.



5.3. Inducción matemática

- La inducción matemática es un método de demostración finito que se utiliza cuando se trata de establecer la veracidad de una lista infinita de proposiciones.
- El método puede ser utilizado en computación a la hora de ver si un algoritmo funciona como se espera.
- La recursión está fundamentada en el principio de inducción matemática ya que se compone del caso base o de menor valor y un caso arbitrario n como hipótesis de partida para probar su siguiente valor n+1.
- Si en ambos se demuestra que es cierto, se probará la afirmación o algoritmo y por tanto el correcto funcionamiento del mismo.

5.3. Inducción matemática

$$1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

- Base de la inducción: Para el $n = 1$: $\quad 1 = \frac{1 \cdot (1+1)}{2}$

- Paso inductivo: Suponemos valida para n = k: $\quad 1 + 2 + \cdots + k = \frac{k \cdot (k+1)}{2}$ (Hipótesis)

  Comprobamos validez para n=k+1:

$$(1 + 2 + \cdots + k) + (k+1) = \frac{k \cdot (k+1)}{2} + k + 1 = \frac{k^2 + 3k + 2}{2} = \frac{(k+1) \cdot (k+2)}{2} = \frac{(k+1) \cdot ((k+1)+1)}{2}$$

5.4. Estructura recursiva

- Estructura:

  - Caso base: puede ser único o múltiple, pero debe definirse siempre. Debe contener una sentencia de retorno o parada que lance la ejecución ordenada de la pila de llamadas.

  - Caso recursivo: Puede ser único o múltiple, pero debe garantizar la aplicación de la técnica de Divide y Vencerás o la actualización de la variable que desencadena en el caso base. Espera la respuesta y la combina con la de la llamada actual.

```
[1]: import time
     import sys

     def factorial_recursivo(n):
         if n == 1:
             return 1
         else:
             return n * factorial_recursivo(n-1)

     inicio = time.time()
     print("Factorial 10: ",factorial_recursivo(10))
     fin = time.time()
     print("Memoria: ",sys.getsizeof(factorial_recursivo)*10," bytes")
     print("Tiempo: "+str(fin-inicio))

     Factorial 10:  3628800
     Memoria:  1360  bytes
     Tiempo: 0.0009965896606445312
```

5.5. Ejemplos

$$1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

```
[3]: def sum_n_iterativo(n):
         suma = 0
         i = 0
         while (i <= n):
             suma = suma + i
             i = i + 1
         return suma

     print("Suma (10 terminos): ",sum_n_iterativo(10))

     Suma (10 terminos):  55
```

```
[6]: def sum_n_recursivo(n):
         if n == 1:
             return 1
         else:
             return n + sum_n_recursivo(n-1)

     print("Suma (10 terminos): ",sum_n_recursivo(10))

     Suma (10 terminos):  55
```

```
[5]: def sum_n_serie(n):
         return n*(n+1)/2

     print("Suma (10 terminos): ",sum_n_serie(10))

     Suma (10 terminos):  55.0
```

5.5. Ejemplos

- Multiplicación de dos números:

```
[8]: def mul_iterativa(a,b):
         i = 0
         res = 0
         while i < b:
             res = res + a
             i = i + 1
         return res

     print("Multiplicación: ",mul_iterativa(5,7))

     Multiplicación:  35
```

```
[11]: def mul_recursivo(a,b):
          if b == 1:
              return a
          else:
              return a + mul_recursivo(a,b-1)

      print("Multiplicación: ",mul_recursivo(5,7))

      Multiplicación:  35
```

```
[9]: def mul_clasica(a,b):
         return a*b

     print("Multiplicación: ",mul_iterativa(5,7))

     Multiplicación:  35
```

5.5. Ejemplos

- Sucesión de Fibonacci:

$$1, 1, 2, 3, 5, 8, 13, \ldots, F_{n-2}, F_{n-1}, F_n = (F_n + F_{n-1})$$

```
[25]: def fibonacci_iterativo(n):
          f_n1 = 1
          f_n2 = 1
          i = 2
          while i <= n:
              f_nn = f_n1 + f_n2
              f_n1 = f_n2
              f_n2 = f_nn
              i = i + 1
          return f_nn

      print("Fibonacci (término 10): ",fibonacci_iterativo(10))

      Fibonacci (término 10):  89
```

```
[20]: def fibonacci_recursivo(n):
          if (n == 0) or (n == 1):
              return 1
          else:
              return fibonacci_recursivo(n-1) + fibonacci_recursivo(n-2)

      print("Fibonacci (término 10): ",fibonacci_recursivo(10))

      Fibonacci (término 10):  89
```