



unab

UNIVERSIDAD NACIONAL
GUILLERMO BROWN

Funciones - Modularidad

Algóritmos y Estructuras de Datos

-00184-

Dr. Diego Agustín Ambrossio

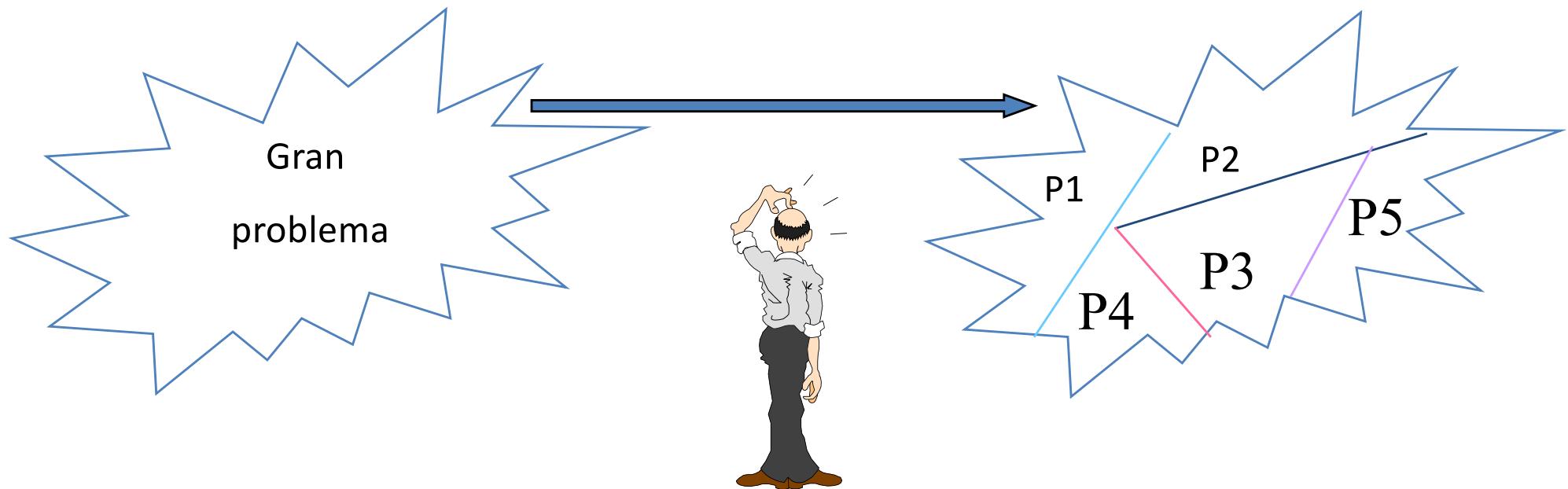
Anl. Sis. Angel Leonardo Bianco

Overview:

Modularidad:

Cuando nos encontramos frente a un gran problema, lo mejor que podemos hacer para solucionarlo es dividirlo en pequeños problemas.

DIVIDE Y VENCERÁS



Procedimientos y Funciones:

Disponemos de dos herramientas básicas para realizar programación descendente:

- los procedimientos; y
- las funciones (**def**)

Estos módulos también son referidos genéricamente con el término de subprogramas.

Procedimientos:

- Deben realizar una tarea específica.
- Pueden recibir cero o más valores del programa que llama y devolver cero o más valores a dicho programa llamador.
- Se los invoca como una sentencia.

En Python sólo utilizaremos el concepto de función!!
(no hay procedimientos)

Funciones:

- También realizan una tarea específica
- Devuelven un valor a la unidad de programa que los referencia.
- Se los invoca utilizando alguna sentencia que nos permita recibir el valor devuelto.

Funciones:

Para definir una función usamos la palabra reservada **def**, luego escribimos la **signatura** de la función, esto es:

- el **nombre** de dicha función
- los **parámetros** que recibe.

Como siempre, debemos terminar la línea con ':' para indicar que a continuación habrá un bloque de código. Obviamente, luego respetar la indentación.

```
def nombreFunción(arg_1 , ... , arg_r) :
```

```
{código de la función}
```

```
...
```

```
{más código de la función}
```

Al igual que cualquier bloque de código indentado, debe contener al menos una linea, en otro caso deberemos escribir **pass** luego de los ':' .

Funciones:

Ejemplo:

```
def Helloworld1():
    print("Hello world")
```

Algunas reglas basicas para definir funciones :

- El **nombre** de la función debe ser representativo (al igual que con los nombres de variable).
- Puede aceptar tantos argumentos como sea necesario, incluso ninguno.
- Siempre que llamamos una función debemos añadir los *parentesis ()*, aunque la función no reciba parámetros.
- Si no proveemos la cantidad correcta de argumentos, la función retornara un **error** del tipo ****TypeError****.

Algunas reglas basicas para definir funciones :

- Los argumentos no tienen un tipo definido. Para controlar que los argumentos sean del tipo esperado deberemos, tendremos que validar su tipo y en caso de error, utilizar el comando `assert` informar que se ha producido un **nuevo** error del tipo de error **TypeError**.
- También podemos utilizar el comando `raise`, el cual nos permite lanzar (o invocar) un **error** (formamente excepciones) informar que se ha producido un error, y el “**tipo de error**”.
- Para que la función retorne un valor **v** usaremos, usaremos el comando `return v`.
- Luego del comando `return` es ejecutado, la función **termina**.
- Si no usamos el comando `return` en el cuerpo de la función, entonces el valor de la función será del tipo ****NoneType****.

Parámetros por referencia:

- Cuando se pasa una variable a una función como parámetro **por referencia**, los cambios que se efectúan sobre dicha variable dentro de la función se mantienen, incluso después de que haya finalizado la función.
- Los cambios producidos en parámetros por referencia son permanentes, ya que no se pasa a la función el valor de hay dentro de la variable, sino la dirección de memoria de la variable.

Parámetros por valor:

- Los parámetros por valor son diferentes a los parámetros por referencia, cuando se pasa un parámetro por valor a una función se guarda en memoria una *copia temporal* de la variable, dentro de la función solo se utiliza la copia, la original nunca se toca.

Modificación de los parametros en Python:

Dependiendo de si el argumento que le pasamos a la función es **mutable** o **inmutable**, es posible modificar su contenido (o no) dentro de la función.

No-mutable: pasaje de parametros "por copia".

Mutable: pasaje de parametros "por referencia".

Argumentos posicionales y argumentos asignados:

Podemos definir los argumentos de dos formas:

- (argumento) posicional: el orden de los argumentos es importante.
- (argumento) asignado: podemos asignar valores, ya sea por defecto, o mediante asignacion directa en la llamada de la función.

```
print('c','g',end=" ** ",sep="*") # 'c' es un argumento posicional, mientras que,  
# 'end'('endofthe[...]printline') y 'sep' son argumentos asignados.  
print('g','c')
```

Argumentos por defecto:

```
def func(posarg_1, ... , posarg_r , kwarg_1 = vk_1 , ..., kwarg_s = vk_s) :  
    {código de la función}
```

...

{más código de la función}

- Los argumentos posarg_1, ... , posarg_r serán **argumentos posicionales** (sin valores por defecto), mientras que vk_1, ... , vk_s tendrán **valores por defecto**.

Argumentos por defecto:

```
def Displayingarguments2(a,b,c,kw1='defaultkw1',kw2='defaultkw2',kw3='defaultkw3'):# Un ejemplo con ambos.

    "Muestra los argumentos uno por linea"
    print("argumento positional a es ",a)
    print("argumento positional b es ",b)
    print("argumento positional c es ",c)
    print("argumento asignado kw1 es ",kw1)
    print("argumento asignado kw2 es ",kw2)
    print("argumento asignado kw3 es ",kw3)
```

Alcanze (scope, ámbito) de la una variable:

El **alcanze** (o **ámbito**) de una variable **v** es el conjunto de lineas de código, en donde la variable es comprendida. Es decir, donde el nombre de variable **v** esta definido.

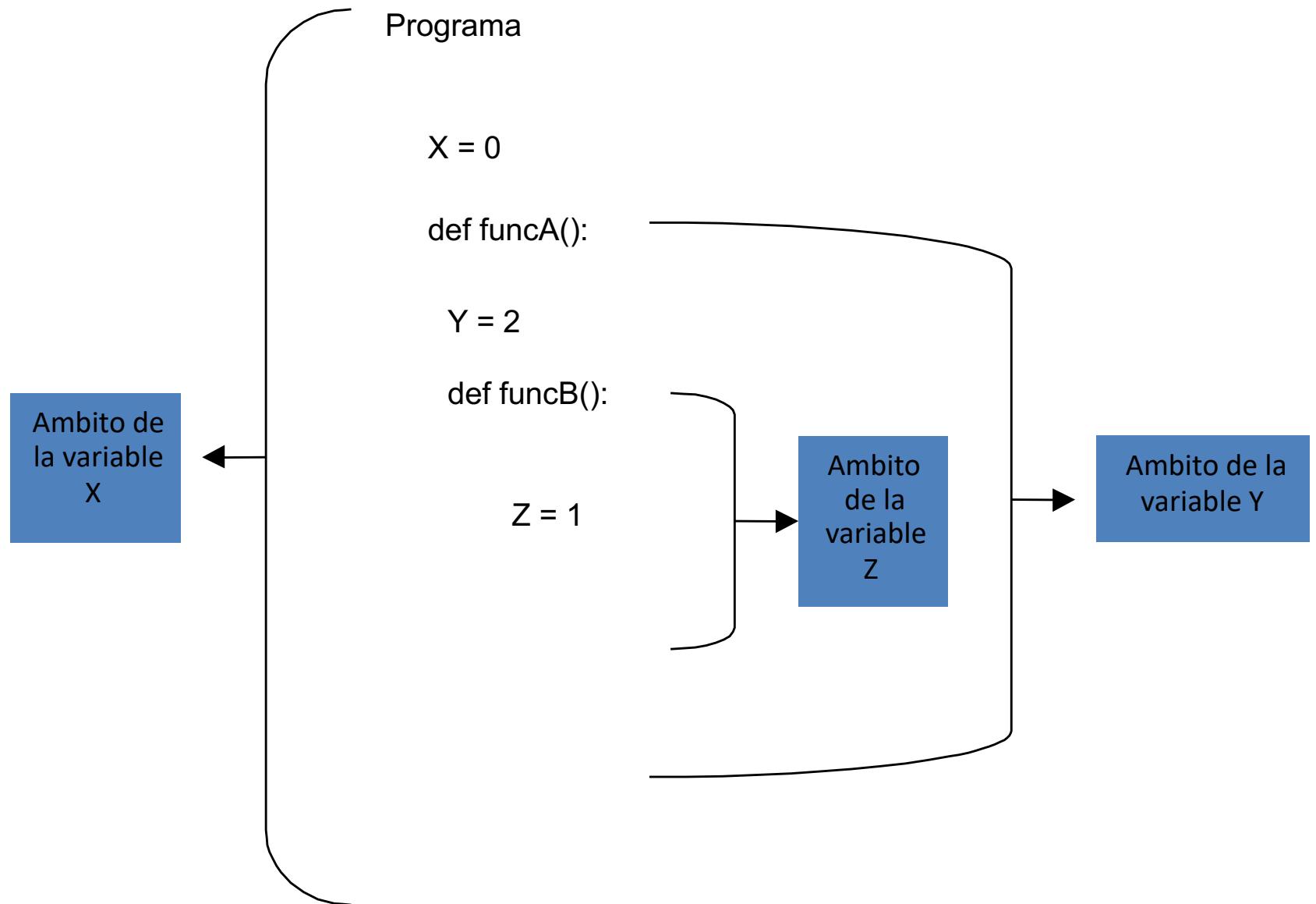
- A priori, es simple:

"si la variable **v** esta definida en la linea **n**, el alcance de la variable **v** será cualquier linea **m > n**".

- Esto se complica cuando introducimos funciones, estas pueden (o no) cambiar el alcance de una variable. Esta situación se complica aún más cuando tenemos *funciones anidadas*, es decir, funciones dentro de otras funciones.

Alcanze (scope, ámbito) de la una variable:

Ejemplo:



Alcanze (scope, ámbito) de la una variable:

La razón principal por la cual los variables tienen un alcance predefinido es para evitar **efectos secundarios**. Es decir, cambiar los valores de variables que pasamos por argumento. Por defecto, el alcance de una variable será **dentro del cuerpo de la función**.

Las variables “**invocadas que no están en el cuerpo de la función no existiran**”.

De esta manera podemos "proteger" las variables fuera de la función.

Alcanze (scope, ámbito) de la una variable:

Es posible cambiar este comportamiento usando el comando **global** o **nonlocal**. La diferencia entre ambos es sutil:

- Sea una variable **v**:
- **global v** : la variable **v** será usada como una variable global (externa a la función), perteneciente al código con mayor alcance.
- **nonlocal v**, la variable **v** será aquella cual alcance es de **un nivel superioir**, es decir, en funciones anidadas, aquella que tenga mayor alcance.

Debemos notar que al utilizar **nonlocal** en funciones recursivas, generara un error del tipo ****SyntaxError****. (lo veremos a continuación)

Definición:

Es una técnica de programación en la cual una función se llama a si misma.

Ejemplo: El factorial de el entero no negativo n es el producto de todos los enteros desde el n hasta 1.

- Formalmente:

$$n! = \prod_{k=1}^{k=n} k$$

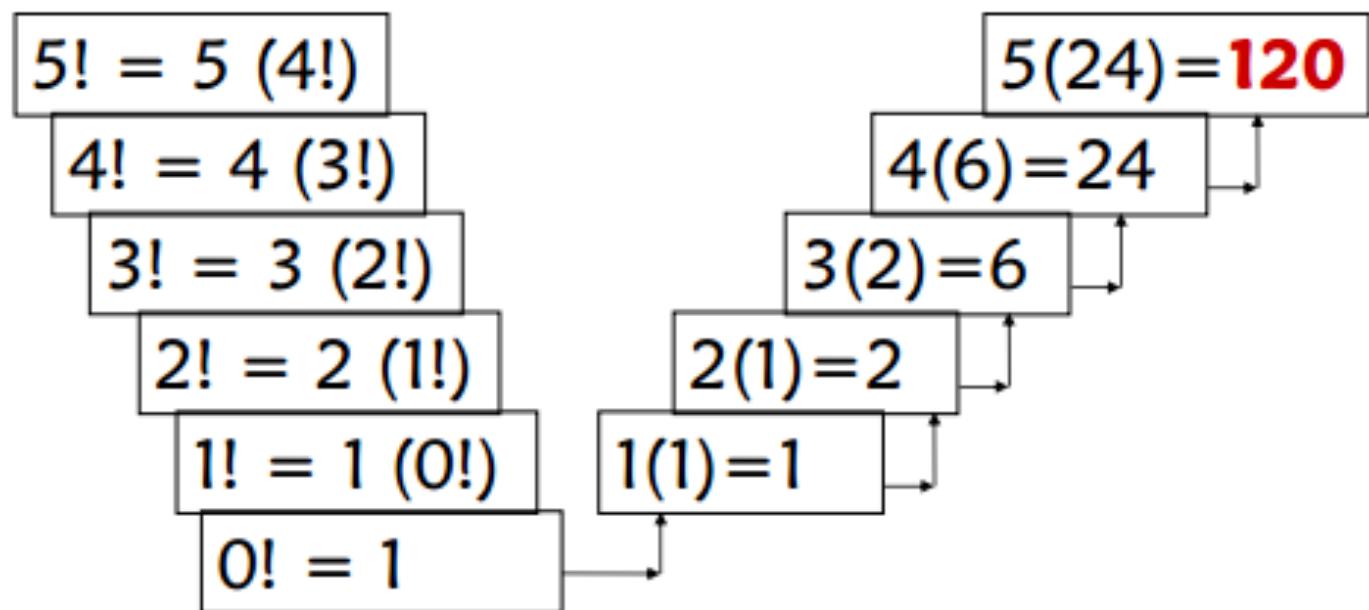
- De este producto es fácil ver que el factorial de n es también

$$n! = n * (n - 1)!$$

Ejemplo:

Ejemplo: El factorial de el entero no negativo n es el producto de todos los enteros desde el n hasta 1.

Ejecución:

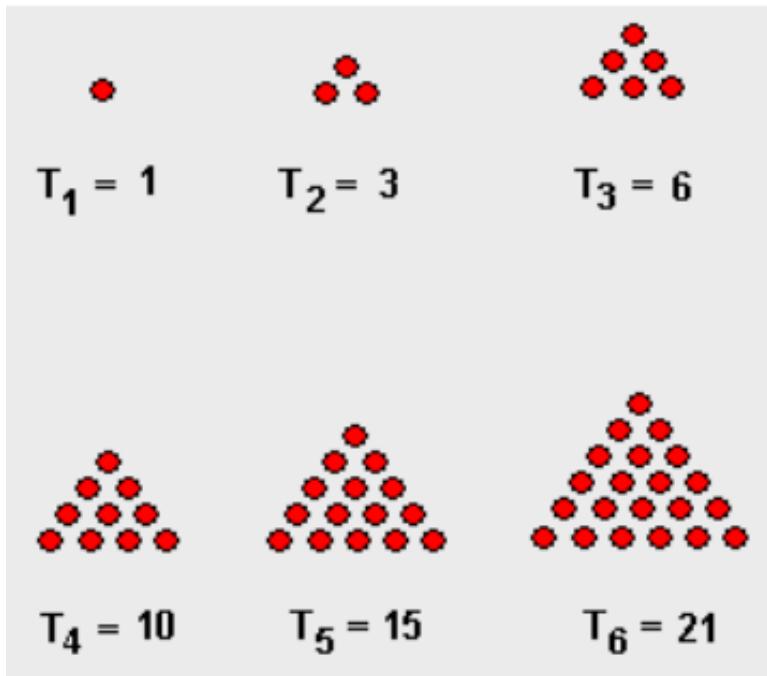


(Otro) Ejemplo:

Dícese que los Pitagóricos sentían una conexión mística con la serie de números 1, 3, 6, 10, 15, 21, ...

Puedes encontrar el próximo miembro de esta serie?

El enésimo término se consigue añadiendo n al término previo

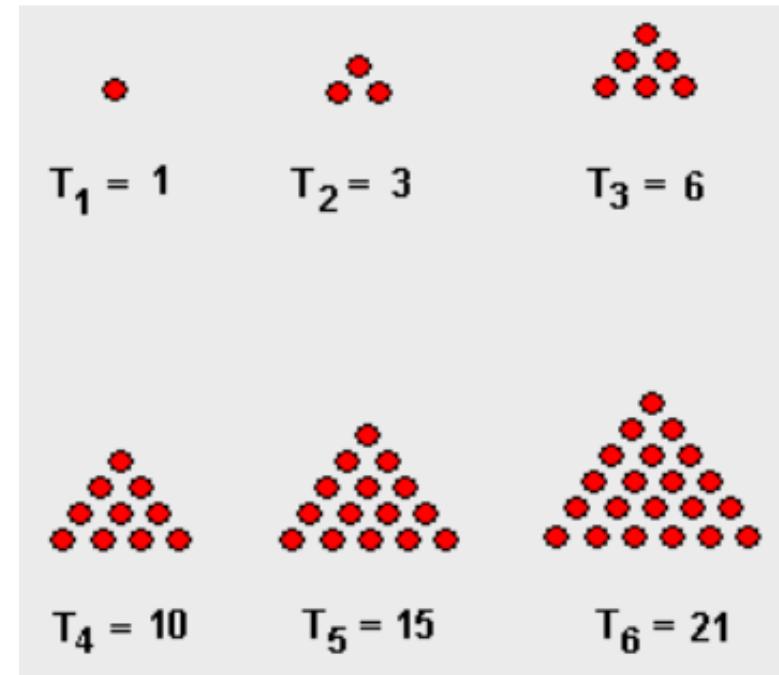


Los números en esta serie se llaman **triangulares** porque pueden ser visualizados de forma triangular

(Otro) Ejemplo:

Pseudo-código:

- Si introducimos 4,
- $4 > 1$ por lo tanto ejecuta $4 + \text{triangulo}(3)$
- Entra la función con argumento 3
- $3 > 1$ por lo tanto $3 + \text{triangulo}(2)$
- Entra la función con argumento 2
- $2 > 1$ por lo tanto $2 + \text{triangulo}(1)$
- Entra la función con argumento 1
- $1 = 1$ por lo tanto retorna 1
- Vuelve a la función con argumento 2
- $\text{triangulo}(1) = 1$, retorna $2 + 1 = 3$
- Vuelve a la función con argumento 3
- $\text{triangulo}(2) = 3$, retorna $3 + 3 = 6$
- Vuelve a la función original, con argumento 4
- $\text{triangulo}(3) = 6$, retorna $4 + 6 = 10$



Funciones Recursivas en Python:

Python nos permite una declaración sencilla de **funciones recursivas**, solamente debemos "*llamar a la función, dentro del cuerpo de la (misma, función)*".

Una definición correcta de una función recursiva, deberá tener el cuenta lo siguiente:

- Existencia de un **caso base** (o inicial) de manera *especial*, es decir cuando la función retorna un valor (no hay llamada recursiva), ya que sino, la función prodria nunca terminar.

Funciones Recursivas en Python:

Manejo standard del caso base:

- Utilizando un `if`, como veremos en los ejemplos.

Luego podremos:

- Terminar la función
- Terminar y devolver un valor que indicará un error en la ejecución de la función
- Opcionalmente podemos informar del error, utilizando los errores predefinidos provistos por Python utilizando el comando `raise` .

Funciones Recursivas en Python:

El comando `raise` es invocado de la siguiente forma:

```
raise AssertionError ( string )
```

Ejemplo:

```
# Cálculo recursivo del factorial de un número n.
def factorial(n):
    if type(n) is not int:
        raise ValueError("N debe ser un numero entero")
    elif n <0 :
        raise ValueError("N debe ser mayor a 0")
    elif n<=1:
        return 1
    else:
        return factorial(n-1)* n
```

Funciones Recursivas en Python:

Ejemplo:

```
# Secuencia de Fibonacci.  
# [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, . . . ]  
def Fibo(n):  
  
    "Definición: F(n)=F(n-1)+F(n-2), F(1)=F(0)=1"  
  
    if n<0:  
        raise ValueError("Fibonacci terms begin at 0") # sin este error, Fibo(-1) entrara en un ciclo eterno.  
    if n==0:  
        return 1 # Primer caso base  
    elif n==1:  
        return 1 # Segundo caso base  
    else:  
        return Fibo(n-1)+Fibo(n-2) ## ESTA ES LA LLAMADA RECURSIVA
```

Algoritmos y Estructuras de Datos

unq



Algoritmos y Estructuras de Datos



Algoritmos y Estructuras de Datos



Algoritmos y Estructuras de Datos



Algoritmos y Estructuras de Datos

