Repaso

- Como saber si han introducido un número negativo

```
str = "-2137"
result = str.lstrip('-').isdigit()
```

- Como dividir una cadena usando un separador
  - `lista = cadena.split(“ “)`
- Como fundir una lista en una cadena usando un separador
  - `cadena = separador.join(lista)`
  - `cadena = “\n”.join(lista)`
  - `cadena = “;”.join(lista)`

Practica P04

Accede a esta web (https://www.tutorialpython.com/)
escoge y copia 4 párrafos de texto y pégalos a un fichero que se llame : texto2.txt.

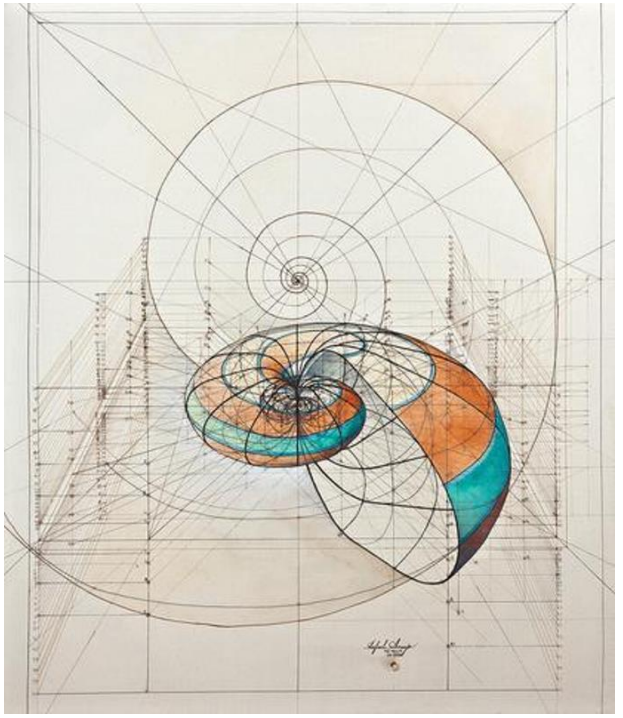
Crea un programa que lea dicho archivo y calcule:

- Número total de líneas:
- Número de palabras por línea :

```
# Solución practica P04

import io

try :
    nombre_fichero = "dat\\texto2.txt"
    f = open(nombre_fichero, encoding="utf-8")
    lineas = f.readlines()
    f.close()

    print ("Num total de lineas : " , len(lineas))

    for i, l in enumerate (lineas) :
        palabras = l.split(" ")
        print ("linea:", i, " palabras : " , len(palabras))
except IOError:
    print ("No existe el archivo : ", nombre_fichero )

except Exception as e:
    print (e)
```

Unió Europea
Fons social europeu

Funciones
…continuación



7. Funciones integradas
8. Continua argumentos en funciones
9. Funciones recursivas

8. Argumentos de funciones

Argumentos indeterminados

Quizá en alguna ocasión no sabemos de antemano cuantos elementos vamos a enviar a una función. En estos casos podemos utilizar los parámetros indeterminados por posición y por nombre.

Por posición

Para recibir un número indeterminado de parámetros por posición, debemos crear una lista dinámica de argumentos (una tupla en realidad) definiendo el parámetro con un asterisco:

```
def indeterminados_posicion(*args):
    for arg in args:
        print(arg)

indeterminados_posicion(5,"Hola",[1,2,3,4,5])
```

Para evitar ese problema, las funciones pueden admitir una cantidad indeterminada de valores, como muestra el ejemplo siguiente

```
import io

def guarda_usuarios (*args) :

    f = open ("dat\\usuarios.dat", "a", encoding='utf-8')

    for arg in args :
        f.write(arg + '\n')
    f.close()

#-------------------------- Inici
guarda_usuarios ("Rosa", "Marta", "Joan", "Pere")
guarda_usuarios ("Elena", "Carles")
```

Abanzado : En realidad args no deja de ser una lista que podemos pasar como tal a otra función

```
import io

def guarda_usuarios (*args) :

    f = open ("dat\\usuarios.dat", "a", encoding='utf-8')
    f.write('\n'.join(args) + '\n')
    f.close()

#------------------------- Inici
guarda_usuarios ("Rosa", "Marta", "Joan", "Pere")
guarda_usuarios ("Elena", "Carles")
```

Normalmente estos argumentos de cantidad variables son los últimos en la lista de parámetros

```
def junta_items(nombre_archivo, separador, *args) :
    archivo = open(nombre_archivo, "a")
    archivo.write(separador.join(args)+"\n")
#
#----------------------proceso principal
#
junta_items("pepe.txt", ";", "a","b","c","d")
```

Este ejemplo crea un archivo se que llama pepe.txt, y graba en él todos los datos que se pasan como parámetros. Los graba separados por ";"

Compruébalo.

Los parámetros anteriores al indeterminado, deben ser obligatorios, de lo contrario ….

Práctica S01

Ejercicio 1. Estudia esta función. El separador (opcional) se sitúa como parámetro final, el programa funciona ?:

```
def concatenar(*args, sep="-" ):
    return sep.join(args)

#--------------------proceso principal
#
print (concatenar ("Tierra", "Luna", "Sol"))
```

```
In [28]:

Tierra-Luna-Sol
```

a) Qué pasaría si hubieras definido la función así :

```
def concatenar(sep="-",*args )
```

y se llamara de la misma forma que antes ?

```
print (concatenar ("Tierra", "Luna", "Sol"))
```

Ejercicio 2. Observa esta función con varios parámetros algunos de los cuales tienen valores por defecto. Realiza en tu ordenador algún ejemplo equivalente, o copia el código que está en la página siguiente …

```
def circuito (tension,
              estado="cerrado",
              accion="puerta bloqueada",
              tipo="JA-U7") :
    #
    #-------- Imprime datos del circuito
    print("-"*30)
    print(">> Tipo circuito :", tipo)
    print(">> Este circuito tiene ", accion)
    print(">> si le aplicas", tension , "voltios")
    print(">> queda en estado", estado)
#fin
#----------------------proceso principal
#
circuito(110, estado="abierto", tipo="X23")
circuito(500, estado="reiniciando")
circuito(110, tipo="X23")
```

Prueba variando los parámetros

```
def circuito (tension,
              estado="cerrado",
              accion="puerta bloqueada",
              tipo="JA-U7") :
    #
    #-------- Imprime datos del circuito
    print("-"*30)
    print(">> Tipo circuito :", tipo)
    print(">> Este circuito tiene ", accion)
    print(">> si le aplicas", tension , "voltios")
    print(">> queda en estado", estado)

#fin funcion circuito

#----------------------proceso principal
#
circuito(110, estado="abierto", tipo="X23")
circuito(500, estado="reiniciando")
circuito(110, tipo="X23")
```

No olvides adaptar las comillas, que con el copia-pega pueden haber quedado mal.

Expresiones Lambda

Expresiones Lambda
Las expresiones lambda se usan idealmente cuando necesitamos hacer algo simple y estamos más interesados en hacer el trabajo rápidamente en lugar de nombrar formalmente la función. Las expresiones lambda también se conocen como funciones anónimas.

Las **funciones Lambda** se comportan como funciones normales declaradas con la palabra clave def. Resultan útiles cuando se desea definir una función pequeña de forma concisa. Pueden contener solo una expresión, por lo que no son las más adecuadas para funciones con instrucciones de flujo de control.

Sintaxis de una función Lambda

```
square = lambda x: x ** 2
print(square(3))
```

lambda argumentos: expresión
Equivale a decir **lambda** parámetros : retorno

Funciones anónimas: Lambda

```
def doblar(num):
    return num*2
```

Todavía más, podemos escribirlo todo en una sola línea:

```
def doblar(num): return num*2
```

Esta notación simple es la que una función lambda intenta replicar, fijaros, vamos a convertir la función en una función anónima:

```
lambda num: num*2
```

Lo único que necesitamos hacer para utilizarla es guardarla en una variable y utilizarla tal como haríamos con una función normal:

```
doblar = lambda num: num*2

doblar(2)
```

Gracias a la flexibilidad de Python podemos implementar infinitas funciones simples. Por ejemplo comprobar si un número es impar:

```
impar = lambda num: num%2 != 0

impar(5)
```

Darle la vuelta a una cadena utilizando slicing:

```
revertir = lambda cadena: cadena[::-1]

revertir("Hola")
```

Incluso podemos enviar varios valores, por ejemplo para sumar dos números:

```
sumar = lambda x,y: x+y

sumar(5,2)
```

Ejemplo : Vamos a crear varias funciones lambda

```
# Función lambda para calcular el cuadrado de un número
square = lambda x: x ** 2

# Dado un texto y una palabra devuelve el número de veces
contar = lambda texto, palabra : texto.count(palabra)


# Pasa un carácter numérico a su letra [a-z]
numletra = lambda cnum : chr(ord(cnum)+48)


#---------------- Probando las funciones
print(square(3))
print (contar ("es aplicable a multitud de situaciones", "es"))
print(numletra("2"))
```

Avanzado : Realiza algún ejercicio de esta página o de esta web

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/funciones-lambda/

Lamda con condicional

```
calc10 = lambda x: True if x**2 >= 10 else False
calc10(3) # Retorna False
calc10(4) # Retorna True
```

Otros ejemplo de lamda que devuelve True o False

```
#  Generamos una lista con los 20 primeros números
mi_lista = [x for x in range(20)]

# Creamos un filtro para localizar los impares

filtrado = filter(lambda x: x % 2 != 0, mi_lista)

# Aplicamos el filtro e imprimimos
print (list(filtrado))
```

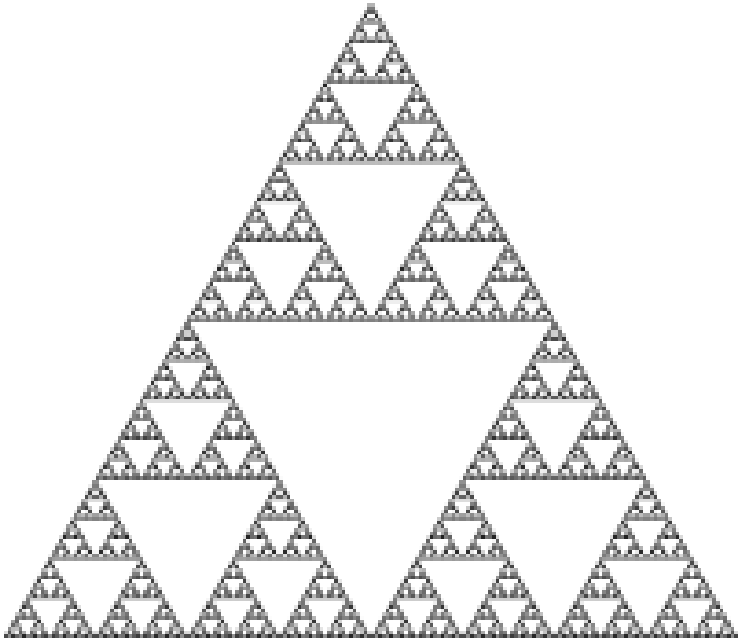
Practica S02

Escribe las funciones lambda que se especifica más abajo, y pruébalas.

- x = lamda con parámetros a, b , que devuelve el producto de a * b
- y = lamda con parámetros a,b,c que devuelve la suma de los 3 parámetros
- z = lamda con parámetros a, b , que devuelve True si a**b > b**a
- y False en otro caso.

9. Funciones recursivas

- Hay algunos problemas que se resuelven de forma iterativa (con un bucle for .. O con while)
- Pero en algunas ocasiones existe otra manera de resolver estos problemas
  - descomponiendo el problema en sub-problemas más pequeños
  - y llamando sucesivas veces a la misma función



Análisis de Costes

Instrucciones fuera de ciclos :

Coste -> constante (K) se desprecia

Bucles : Coste -> n

Bucle 2 niveles : Coste -> n exp 2 ejemplo : 7exp2 = 49

Triple 3 niveles : Coste -> n exp 3 ejemplo : 7exp3 = 343

```
coste = 0
for i in range (0,7):
    for j in range (0, 7):
        coste +=1
print (coste)
```

```
fichas = []
for i in range (0, 7):
    for j in range (0, 7):
        fichas.append("{}-{}".format(i,j))

random.shuffle(fichas)
```

coste desconocido

1. El coste de los algoritmos

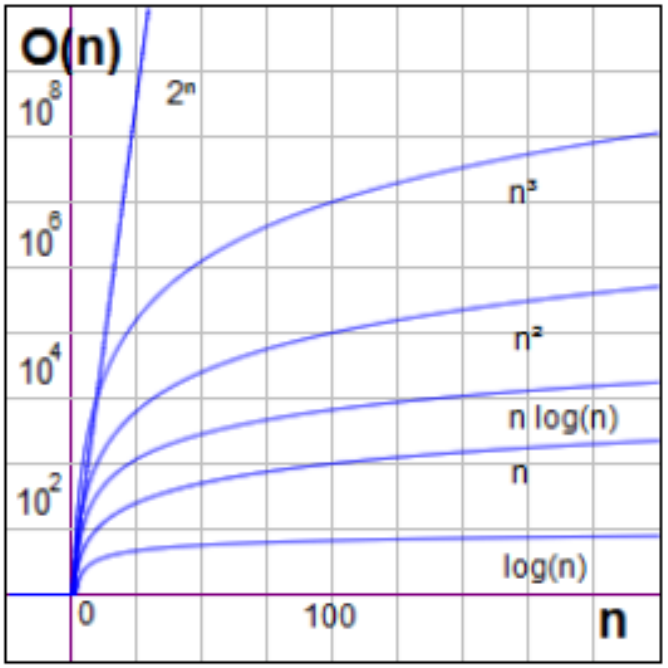


Figura 1. Coste de los algoritmos

Los algoritmos recursivos van reduciendo el tamaño del problema en cada llamada. Si tenemos un problema de tamaño n puede que lleguemos a la solución con un coste linealmente proporcional a n y lo representamos como $O(n)$, que leemos como *"en el orden de n"*. Uno cuadrático sería $O(n^2)$ por lo que tendría un coste superior, al igual que uno exponencial como $O(2^n)$. En cambio uno con $O(\log n)$ tendría un coste inferior al lineal (entenderemos log como logaritmos en base 2).

En la *Figura 1* puede ver la representación de las funciones de coste más frecuentes: $log(n)$, n, $n\,log(n)$, n^2, n^3 y 2^n.

Entender el coste de los recursivos sirve para buscar formas de plantearlos que supongan una reducción de ese coste.

Funciones recursivas

Se trata de funciones que se llaman a sí mismas durante su propia ejecución. Funcionan de forma similar a las iteraciones, pero debemos encargarnos de planificar el momento en que dejan de llamarse a sí mismas o tendremos una función rescursiva infinita.

Suele utilizarse para dividir una tarea en subtareas más simples de forma que sea más fácil abordar el problema y solucionarlo.

```
def busqueda_binaria_recursiva(arreglo, busqueda, izquierda, derecha):
    if izquierda > derecha:
        return -1
    indiceDelElementoDelMedio = (izquierda + derecha) // 2
    elementoDelMedio = arreglo[indiceDelElementoDelMedio]
    if elementoDelMedio == busqueda:
        return indiceDelElementoDelMedio
    if busqueda < elementoDelMedio:
        return busqueda_binaria_recursiva(arreglo, busqueda, izquierda, indiceDelElementoDelMedio - 1)
    else:
        return busqueda_binaria_recursiva(arreglo, busqueda, indiceDelElementoDelMedio + 1, derecha)
```

Ejemplo sin retorno

Cuenta regresiva hasta cero a partir de un número:

Código Resultado

```
def cuenta_atras(num):
    num -= 1
    if num > 0:
        print(num)
        cuenta_atras(num)
    else:
        print("Booooooooom!")
    print("Fin de la función", num)

cuenta_atras(5)
```

```
def cuenta_atras(num):
#------------ ejemplo de recursiva
    num -= 1
    if num > 0:
        print(num)
        cuenta_atras(num)
    else:
        print("BOOOOOOOOM !!!!!!")
    print(">>Fin función ", num)

#--------------------- inicio
#
cuenta_atras(5)
```

Ejemplo con retorno

El factorial de un número corresponde al producto de todos los números desde 1 hasta el propio número. Es el ejemplo con retorno más utilizado para mostrar la utilidad de este tipo de funciones:

https://docs.hektorprofe.net/python/programacion-de-funciones/funciones-recursivas/

¿Qué es la función factorial?

La función factorial se representa con un signo de exclamación "**!**" detrás de un número. Esta exclamación quiere decir que hay que multiplicar todos los números enteros positivos que hay entre ese número y el 1.

Por ejemplo:

$$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$$

A este número, **6!** le llamamos generalmente "**6 factorial**", aunque también es correcto decir "**factorial de 6**".

En tu calculadora podrás ver una tecla con "n!" o "x!". Esta tecla te servirá para calcular directamente el factorial del número que quieras.

Algunos ejemplos de factoriales

Vamos a ver algunos ejemplos más de factoriales:

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

$$10! = 1 \times 2 \times \cdots \times 9 \times 10 = 3628800$$

$$100! = 1 \times 2 \times 3 \times \cdots \times 98 \times 99 \times 100 \approx 9{,}33 \times 10^{157}$$

Como ves, 100! es enorme...

Y, ¿qué hacemos con los números más pequeños? 1 factorial es, lógicamente, 1, ya que multiplicamos 1 x 1:

$$1! = 1$$

Pero, ¿cómo podemos calcular el 0 factorial? Bueno, esto no tiene sentido cuando aplicamos la norma de que hay que multiplicar todos los números enteros positivos entre el 0 y el 1, ya que 0 x 1 es 0.

Al final, por convenio se ha acordado que lo más útil es que el 0 factorial sea igual a 1. Así que recuerda:

$$0! = 1$$

¿Para qué podemos utilizar los factoriales?

Los números factoriales se utilizan sobre todo en **combinatoria**, para calcular combinaciones y permutaciones. A través de la combinatoria, los factoriales también se suelen utilizar para calcular probabilidades.

Vamos a ver un ejemplo sencillo de problema en el que podemos aplicar los factoriales:

Pepa ha sacado los 4 ases de una baraja.
Va a colocarlos en fila encima de la mesa.
¿De cuántas maneras distintas podría colocarlos?

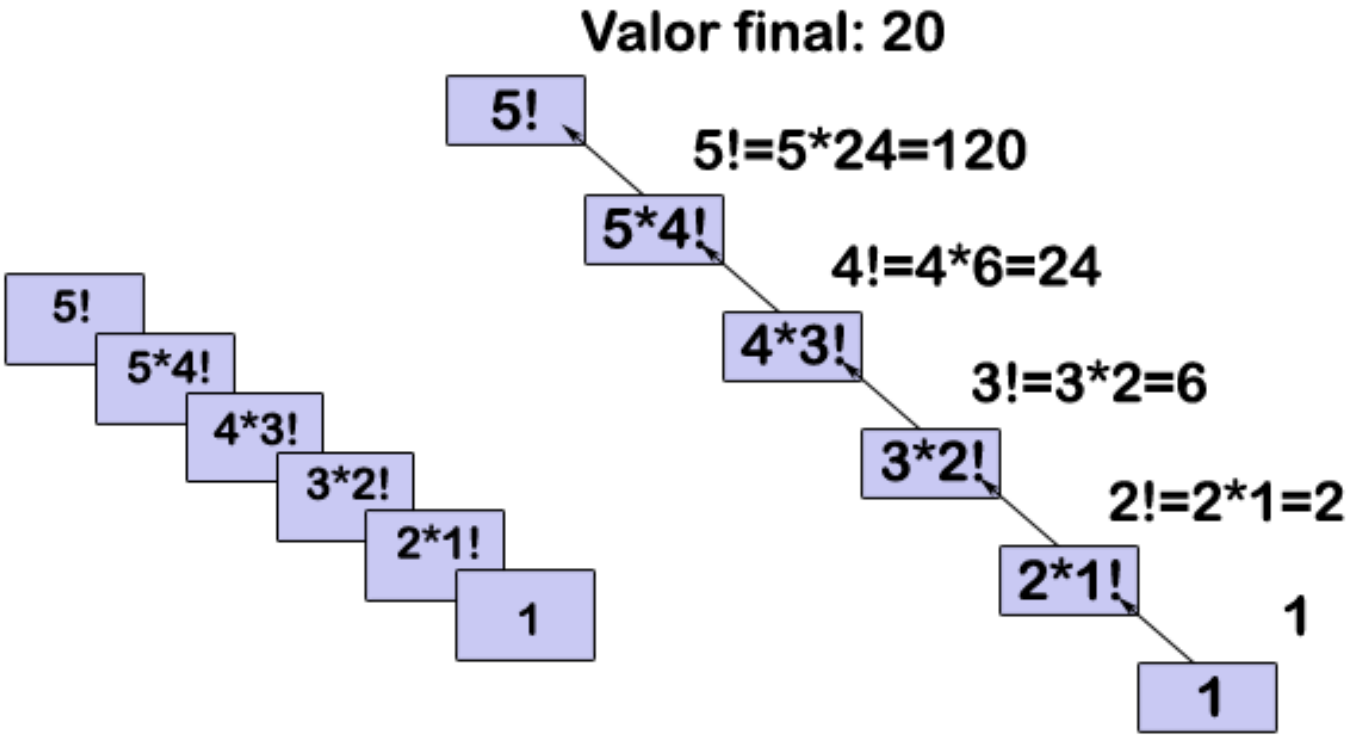
En este problema nos están pidiendo lo que se llama una **permutación**, es decir, que averigüemos todas las maneras posibles en las que estas 4 cartas se pueden combinar **teniendo en cuenta el orden** en el que las colocamos.

Por lo tanto, todas las combinaciones posibles serán 4 x 3 x 2 x 1.

O lo que es lo mismo, 4! = 24

Para ello vamos a estudiar

- como se puede DESCOMPONER el factorial de un número,
- como se puede calcular de forma iterativa
- como se puede calcular de forma recursiva



En forma iterativa

```
def factorial (n) :


   fact = 1
   for i in range(2, n+1) :
     fact = fact * i
     print ("vuelta : " , i, fact)
   return (fact)

#--------------------------- Main

n = int (input ("Introduzca un número: "))

print ( "resultado " ,factorial (n))
```

```
Introduzca un número: 5
vuelta :  2 2
vuelta :  3 6
vuelta :  4 24
vuelta :  5 120
resultado  120
```

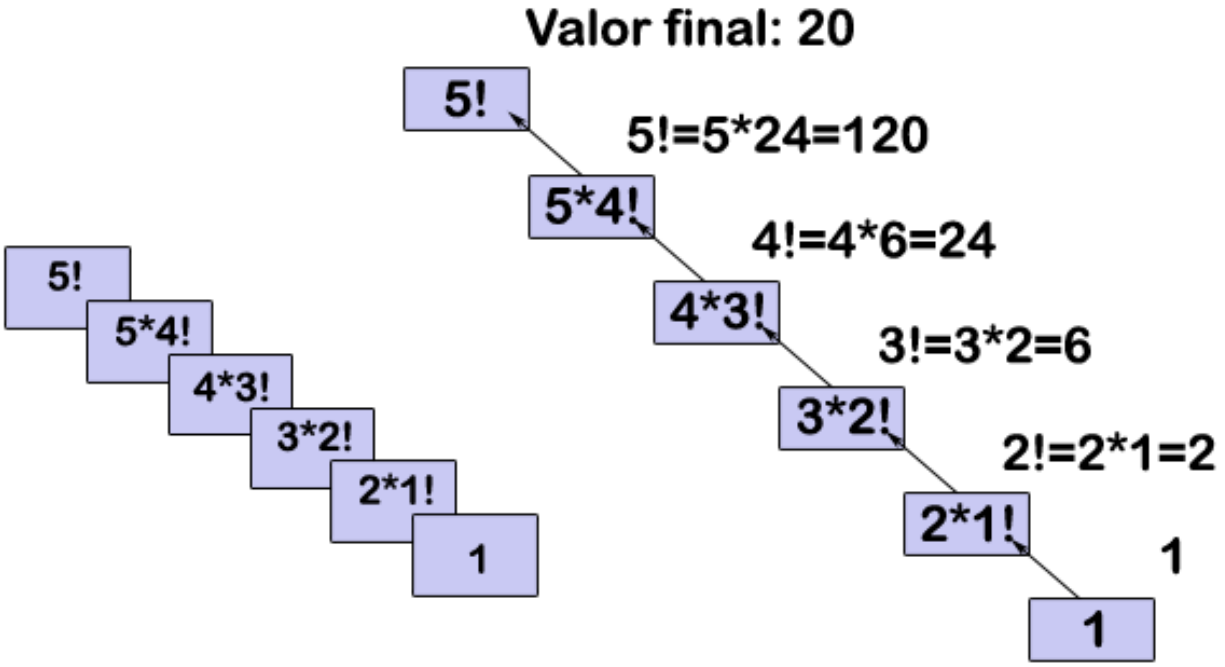
En forma recursiva

```
def factorial (n) :
   if n == 0 :
      n = 1
   else :
      n= n * factorial (n-1)
   return (n)

#--------------------------- Main

n = int (input ("Introduzca un número: "))

print ( "resultado " ,factorial (n))
```

```
Introduzca un número: 5
resultado  120
```

```python
def factorial (n) :
    if n == 0 :
        n = 1
    else :
        n= n * factorial (n-1)
    return (n)

#---------------------------- Main

n = int (input ("Introduzca un número: "))

print ( "resultado " ,factorial (n))
```

```
Introduzca un número: 5
resultado  120
```



Python's default **recursion limit** is 1000, meaning that **Python** won't let a function call on itself more than 1000 times

How do you find the recursive limit in Python?

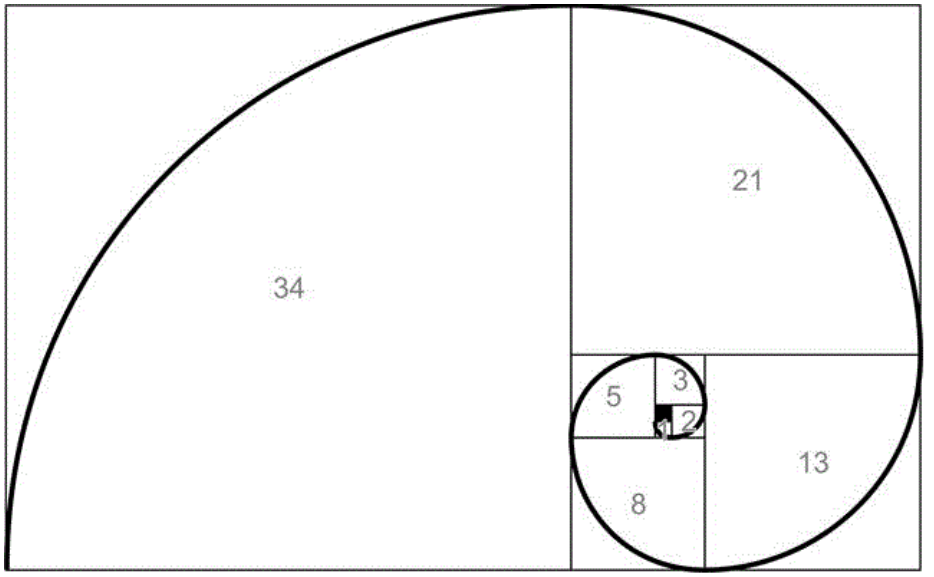
The **Python** interpreter **limits** the **recursion limit** so that infinite recursions are avoided. The " sys " module in **Python** provides a function called setrecursionlimit() to modify the **recursion limit in Python**. It takes one parameter, the value of the new **recursion limit**. By default, this value is usually 10^4. 19 jul. 2019

Recursión múltiple: La función de Fibonacci

- La sucesión de Fibonacci es: 0,1,1,2,3,5,8,13,21,. Sus dos primeros términos son 0 y 1 y los restantes se obtienen sumando los dos anteriores.
- `(fibonacci n)` es el `n`-ésimo término de la sucesión de Fibonacci. Por ejemplo,

```
# Fibonacci version recursiva
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

$$\text{fib}(n) = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{Si } n > 1 \end{cases}$$



Práctica P01

-Estudia, pasa a tu cuaderno y modifica la versión de fibonacci recursiva.

-- Avanzado : Visita esta web para trazar el fractal de Fibonacci

https://www.geeksforgeeks.org/python-plotting-fibonacci-spiral-fractal-using-turtle/

Practica P02

- Avanzado : Repasa estos apartados

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/operadores-encadenados/

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/comprension-de-listas/

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/funciones-decoradoras/

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/funciones-generadoras/

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/funcion-filter/

https://docs.hektorprofe.net/python/funcionalidades-avanzadas/funcion-map/