Introducción a Python Avanzado - Parte 1

Fuente: RealPython

Luis A. Muñoz (2024)

Funciones definidas por el usuario

Las funciones son codigos autónomos que implementan soluciones puntuales. Son parte del rompecabezas que puede ser un proyecto de programación final. Consisten en un bloque de codigo que toma valores de entrada (parametros) para ser parte del proceso y retornar un valor o varios como resultado.

In [ ]:
```
def factores_primos(n):
    '''factores primos(n)   Funcion que retorna los factores primos de un numero

    Parametros:
        - n: int

    Uso:
        factores_primos(12)  -> [2, 2, 3]'''
    if n < 1 or not isinstance(n, int):
        return []

    div = 2
    factores = []
    # Lazo que encuentra los divisores del numero (factores)
    while n != 1:    # is not
        # Si se tiene un factor, se guarda en la lista de salida
        if n % div == 0:
            factores.append(div)
            n //=  div   # n = n // div
        else:
            # ...de lo contrario se pasa a considerar el siguiente posible divisor
            div += 1     # div = div + 1
```

```
    return factores
```

In [ ]:
```
print(factores_primos(360))
```

In [ ]:
```
help(factores_primos)
```

Type Hints

Python es un lenguaje de tipado dinámico, esto es que los tipos de datos se asignan de forma dinámica al momento de asignar un valor a una variable. Esto convierte a Python en un lenguaje de prototipado rápido. Sin embargo, cuando se trabaja en equipos de trabajo o cuando se quiere documentar correctamente un script, tener alfun mecanismo de control de tipos de datos es útil.

Considere el siguiente ejemplo:

```
def circunferencia(radio, point):
    pass
```

¿El parámetro `point` es un valor, es un string, es una tupla o una lista, y en caso de ser estos últimos, cual es su formato?

Esta es la razón por la que Python introdujo el Type Hinting (anotación de tipos) en la versión 3.5 y ha ido agregando más facilidades. Al mismo tiempo, librerías como `pydantic` y `FastAPI` utilizan el Type Hinting para el control de tipos de datos de forma nativa, por lo que utilizar el Type Hinting no solo resulta útil sino cada vez más necesario.

In [ ]:
```
nombre: str = "Elvio Lado"
altura: float = 1.75
peso: int = 56

print("Nombre:", nombre)
print("Altura:", altura, "m")
print("Peso:", peso, "kg")
```

In [ ]:
```
pares: list[num] = [num for num in range(20) if num % 2 == 0]
meses: dict[num, str] = dict(zip([1, 2, 3, 4, 5, 6], ['ene', 'feb', 'mar', 'abr', '
nombres: set[str] = {"Elvio", "Dina", "Elmer", "Elvio"}

print(pares)
print(meses)
print(nombres)
```

Aunque en términos generales es una buena práctica, donde se recomienda su uso es en la definición de las funciones y clases:

In [ ]:
```
def factorial(n: int) -> int:
    '''factorial(n)     Retorna el factorial de n (forma recursiva)
    '''
    if not isinstance(n, int):
        raise TypeError("El valor 'n' debe ser un 'int'")
```

```
    if n < 0:
        raise ValueError("El valor 'n' debe ser un 'int' >= 0")

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

In [ ]:
```
factorial?
```

In [ ]:
```
factorial(5)
```

Se pueden definir colecciones de diferentes tipos de datos:

In [ ]:
```
def promedio_notas(notas: list[int, float]) -> float:
    return sum(notas) / len(notas)
```

In [ ]:
```
print(f"Promedio: {promedio_notas([14, 12.5, 13, 16.8]):.2f}")
```

También se puede especificar el tipo `Optional` para indicar que puede ser de tipo `None` (será necesario importar la clase de la librería `typing`):

In [ ]:
```
from typing import Optional

def valores_comun(fila: list[int], columna: list[int]) -> Optional[set]:
    if len(set(fila) & set(columna)) > 0:
        return set(fila) & set(columna)
    else:
        return None
```

In [ ]:
```
print(valores_comun([1, 2, 3, 4], [3, 6, 9]))
```

In [ ]:
```
print(valores_comun([1, 2, 3, 4], [8, 6, 9]))
```

También se puede especificar si un parámetro puede ser de diferentes tipos con `Union` o a partir de Python 3.10, con el operador `|`:

In [ ]:
```
from typing import Union

def IMC(peso: Union[int, float], altura:Union[int, float]) -> float:
    return peso / altura ** 2
```

In [ ]:
```
print(f"IMC: {IMC(80, 1.76):.2f}")
```

In [ ]:
```
def IMC(peso: int | float, altura: int | float) -> float:
    return peso / altura ** 2
```

In [ ]:
```
print(f"IMC: {IMC(80, 1.76):.2f}")
```

Bloque try... except

Cuando un script llame a la función `factorial` puede hacer un "intento" con la instrucción `try` y si es que esta función genera una excepción, esta es capturada con la instrucción

`except` y se puede mostrar un error propio del script.

In [ ]:
```
num = 1.5        # Prueba float
# num = -5       # Prueba negativo

try:
    factorial(num)
except TypeError:
    print("Error: 'num' debe ser entero")
except ValueError:
    print("Error: 'num' debe ser mayor o igual a 0")
```

Los mensajes de error de la función original se pueden mantener extrayendolos con `except Exception as e`, que captura el mensaje de la excepción generada.

In [ ]:
```
num = 1.5        # Prueba float
# num = -5       # Prueba negativo

try:
    factorial(num)
except Exception as e:
    print(e)
```

*args y **kwargs

En algunas ocasiones se requiere que una función tenga un número de parametros aleatorios, ya sea en forma de argumentos posicionales o por `keywords`. Esto se logra con los symbolos especiales `*` (desempaquetado en tuplas) y `**` (desempaquetado en diccionarios) y normalmente se especifican de la forma `*args` y `**kwargs`.

In [ ]:
```
from typing import Any

def foo(*args: Any, **kwargs: Any) -> Any:
    for arg in args:
        print(arg)

    for k, v in kwargs.items():
        print(k, '->', v)
```

In [ ]:
```
foo('A')
```

In [ ]:
```
foo('A', 'B', 'C')
```

In [ ]:
```
foo(num1=1)
```

In [ ]:
```
foo(num1=1, num2=2, num3=3)
```

In [ ]:
```
foo('A', 'B', 'C', num1=1, num2=2, num3=30)
```

Esta especificación permite reducir la definición de los argumentos de una función y se utiliza mucha en la documentación de Python. Un buen ejemplo es el método `format` de la clase `str` que soporta diferentes argumentos y keywords:

In [ ]:
```
help(str.format)
```

Generadores con yield

Una función puede ser un *generador*, esto es un objeto que no retorna una secuencia de elementos sino un *motor* que genere valores según la regla de la función y que será controlado por un iterador como un lazo `for`. Para esto, en lugar de retornar una lista o una tupla, la función retorna los valores por separado utilizando la instrucción `yield` en lugar de `return`:

In [ ]:
```
def range_letters(ini: str='A', end: str='Z', case: str='upper', reverse: bool=Fals
    if not isinstance(ini, str) or not isinstance(end, str):
        raise TypeError

    if all([ini.isalpha(), end.isalpha()]):     # all: True si [True, True]
        if ini.upper() < end.upper():
            if not reverse:
                letter = ord(ini.upper()) - 1
                while letter < ord(end.upper()):
                    letter += 1
                    if case == 'upper':
                        yield chr(letter).upper()
                    elif case == 'lower':
                        yield chr(letter).lower()
                    else:
                        raise AttributeError
            else:
                letter = ord(end.upper()) + 1
                while letter > ord(ini.upper()):
                    letter -= 1
                    if case == 'upper':
                        yield chr(letter).upper()
                    elif case == 'lower':
                        yield chr(letter).lower()
                    else:
                        raise AttributeError
        else:
            raise ValueError
    else:
        raise ValueError
```

Si se llama a la función lo que retornará es un objeto generador:

In [ ]:
```
print(range_letters('A', 'C'))
```

Se puede utilizar la función `next` para que retorne el siguiente valor del generador. `next` ira retornando los valores del generador hasta que genere una excepción por haber agotado los valores de salida.

In [ ]:
```
gen = range_letters('A', 'C')
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))   # Esta linea genera un Excepcion StopIteration
```

¿Ahora entiende porque la función `range` al ser impresa no retorna el rango de números sino la especificación del rango a generar? Esto es porque `range` es un generador y si se especifica que se quieren generar 10, 100 o 1000 números, el uso de los recursos es el mismo pues la función solo retorna un número por vez y no una lista de números.

In [ ]:
```
print(range(1, 11))
```

In [ ]:
```
import sys

rango = range(1, 1000)
print("Bytes:", sys.getsizeof(rango))

rango = range(1, 10000000)
print("Bytes:", sys.getsizeof(rango))
```

Iterables con iter

¿Se entiende ahora por qué el lazo `for` en Python es tan extraño por no utilizar indices o controles de fin de iteración? Se puede entender como un lazo `while` con la instrucción `next` en el interior, y la extracción de datos se mantiene hasta que se genera la exception `StopIteration` (siempre y cuando el elemento a iterar sea un generador).

In [ ]:
```
for letter in range_letters('C', 'H', reverse=True):
    print(letter)
```

Esto es gracias a que la instrucción `for` es un *iterador* y lo que hace es barrer los elementos de un objeto *iterable*. Como se ve en el ejemplo anterior, un generador es un iterable, pero no al reves: un iterador es un concepto más general asociado a los objetos de Python. La instrucción `for` convierte un objeto en un iterador para luego barrer los elementos. Esto también se puede conseguir con la función `iter()`:

In [ ]:
```
texto = "hola mundo"
iterable = iter(texto)
print(next(iterable))
print(next(iterable))
print(next(iterable))
print(next(iterable))
```

La librería collections

La libreria `collections` viene a complementar las colecciones de datos estandar de Python (tuplas, listas, diccionarios, conjuntos) por objetos con mayor versatilidad o que permiten realizar tareas más especializadas. Se exponen algunos elemetos útiles de la librería `collections`:

Counter

La clase `Counter` permite contar elementos dentro de una colección de una forma sencilla sin recurrir a lazos de repetición:

In [ ]:
```
from collections import Counter
from random import randint

chars = [chr(randint(ord('a'), ord('z'))) for _ in range(100)]
```

```
count = Counter(chars)
print(count)
```

In [ ]:
```
count['a']
```

In [ ]:
```
count.most_common(3)
```

deque

La clase `deque` permite construir una cola, la sea LIFO (Last In First Out) o FIFO (First In First Out):

In [ ]:
```
from collections import deque

buffer = deque(['b', 'c', 'd', 'e', 'f', 'g'])
print(buffer)
```

In [ ]:
```
buffer.append('h')
print(buffer)
```

In [ ]:
```
buffer.appendleft('a')
print(buffer)
```

In [ ]:
```
val = buffer.pop()
print(buffer, val, sep=' -> ')
```

In [ ]:
```
val = buffer.popleft()
print(buffer, val, sep=' -> ')
```

Se puede definir el tamaño de una cola para que actue de forma automática con la gestión de los elementos:

In [ ]:
```
buffer = deque(maxlen=10)
buffer.extend([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(buffer)
```

In [ ]:
```
buffer.append(11)
print(buffer)
```

In [ ]:
```
buffer.appendleft(0)
print(buffer)
```

namedtuples

Se puede construir una clase sencilla que solo contemple propiedades utilizando `namedtuples` (pero ojo, es una construcción inmutable).

In [ ]:
```
from collections import namedtuple

Alumno = namedtuple('Alumno', ['nombre', 'apellido', 'codigo'])
```

In [ ]:
```
alumno1 = Alumno('Elvio', 'Lado', 'a81277222')
print(alumno1.nombre)
print(alumno1.codigo)
print(alumno1)
```

¡Y tiene su propio `__repr__` !

---

time y datetime

La gestión del tiempo en Python es un dolor de cabeza. Esto porque hay diferentes maneras de hacer los mismo con diferentes módulos. Asi que para encontrar un orden, se puede tomar la siguiente guia de referencia:

- `time` : información del tiempo (horas, minuto segundo)
- `datetime` : manipulacion de fechas(operaciones con el tiempo)

time

El módulo `time` permite obtener información del tiempo. Por ejemplo, se puede obtener una *estructura* del tiempo con `time.localtime` de tal forma que los diferentes campos se puede extraer con el operador `.` :

In [ ]:
```
import time

time_now = time.localtime()
print(time_now)

print(f"{time_now.tm_hour}:{time_now.tm_min}")
print(f"Han pasado {time_now.tm_yday} dias desde el 1 de Enero de {time_now.tm_year
```

La información del tiempo se puede obtener (en inglés) como una cadena de impresión con `time.asctime` ("asc" de ASCII):

In [ ]:
```
time.asctime()    # Por defecto es el tiempo actual. Prueba como argumento time_now
```

Sin embargo, una buena practica de programación no es usar el tiempo local sino el tiempo universal (UTC). Esto es, información de tiempo referenciado a un meridiano terreste (UTC-0). Esto permite generar un valor de tiempo que será reconocido y convertido a tiempo local en la zona de consulta. Para esto se necesita un tiempo de referencia donde todos los husos horarios consideren "el inicio del tiempo". Eso existe, es el "UNIX Time" referenciado a un "epoch": 1 de enero de 1970.

In [ ]:
```
time_now = time.time()
print(f"{time_now:,}")
```

Ese es el número de segundos que han transcurrido desde el 1 de enero de 1970. Si queremos retornar esto a una estructura de tiempos debemos llamar al método `time.gmtime()` ("gm" de "Greenwich Mean Time" o "GMT"):

In [ ]:
```
time.gmtime(time_now)
```

Estas estructuras de tiempo se pueden mostrar con un formato personalizado (ya que al usar `asctime` obtendremos una respuesta en inglés y con un formato fijo) gracias al método `time.strftime()`, donde `f` especifica *format*, es decir que retorna un `str` con la información de tiempo con las siguientes especificaciones de formato:

(Extraido de la ayuda del método `time.strftime`)

```
strftime(...)
    strftime(format[, tuple]) -> string

    Convert a time tuple to a string according to a format
specification.
    See the library reference manual for formatting codes. When the
time tuple
    is not present, current time as returned by localtime() is
used.

    Commonly used format codes:

    %Y  Year with century as a decimal number.
    %m  Month as a decimal number [01,12].
    %d  Day of the month as a decimal number [01,31].
    %H  Hour (24-hour clock) as a decimal number [00,23].
    %M  Minute as a decimal number [00,59].
    %S  Second as a decimal number [00,61].
    %z  Time zone offset from UTC.
    %a  Locale's abbreviated weekday name.
    %A  Locale's full weekday name.
    %b  Locale's abbreviated month name.
    %B  Locale's full month name.
    %c  Locale's appropriate date and time representation.
    %I  Hour (12-hour clock) as a decimal number [01,12].
    %p  Locale's equivalent of either AM or PM.

    Other codes may be available on your platform.  See
documentation for
    the C library strftime function.
```

In [ ]:
```
time.strftime("%I:%M:%S %p", time.localtime())
```

Lo contrario se puede obtener con `time.strptime()`, donde `p` especifica `parse`, por lo que identifica una cadena y corta las secciones según la especificación del formato para convertirlo en una estructura de tiempo:

In [ ]:
```
time.strptime("10:20:15", "%H:%M:%S")
```

Otro uso típico de la librería `time` es la inclusión de la función `sleep` para hacer pausas en la ejecución de un script:

In [ ]:
```
import os
from IPython.display import clear_output
```

```
num = 10
while num >= 0:
    time.sleep(0.5)
    print(num)
    num -= 1

    clear_output(wait=True)
    #os.system('cls')

print("--- BOOM! ---")
```

datetime

El módulo `datetime` se utiliza para hacer operaciones con las fechas. Esto resulta muy útil pues evita tener que hacer calculos complejos para calcular una fecha en el futuro o en el pasado, tomando en consideración los años bisiestos, los meses de 30 o 31 dias, etc.

Para obtener la fecha de hoy como un objeto DateTime llamamos al método `datetime.datetime.now`:

In [ ]:
```
import datetime

time_now = datetime.datetime.now()
print(time_now)
```

Podemos definir una fecha cualquiera cargando los valores de año, mes, dia, hora, minuto y segundo (los datos de hora que falten se cargan con los valores 0):

In [ ]:
```
time_past = datetime.datetime(2001, 1, 1, 13, 30)    # 1 de Enero 2001, 00:00 horas
print(time_past)
```

También se puede definir un objeto `datetime` a partir de un `str` utilizando el método `strptime` con el formato de correcto:

In [ ]:
```
time_from_str = datetime.strptime("3/2/2024 10:45", "%d/%m/%Y %H:%M")
print(time_from_str)
```

Si hacemos operaciones de suma o resta con los objetos `datetime`, obtendremos un nuevo objeto: `timedelta`, una diferencia de tiempo:

In [ ]:
```
time_delta = time_now - time_past     # Cuanto tiempo ha transcurido del siglo XXI?
print(time_delta)
```

Se puede definir un `timedelta` para calcular una fecha (se define con información de segundos, minutos, horas o dias). Por ejemplo, que fecha tendremos 100 días en el futuro:

In [ ]:
```
time_delta = datetime.timedelta(days=100)
time_future = datetime.datetime.now() + time_delta
print(time_future)
```

f-strings

Un *f-string* es una cadena de texto con formato incluido y es la forma preferida de generar cadenas de texto con formato desde la version 3.6 de Python. Se utiliza el caracter `f` antes

de definir una cadena con comodines de formato `{}` y en el interior se colocan los valores a ser asignados.

In [ ]:
```
peso = 80
altura  = 1.60
imc = peso / altura**2

# Uso del método format de los strings
print("Para una persona de {:.2f} m y {} kg, el IMC es de {:.1f}".format(altura, pe
```

In [ ]:
```
# Uso de un f-string
print(f"Para una persona de {altura:.2f} m y {peso} kg, el IMC es de {imc:.1f}")
```

Los f-strings son muy versatiles. Por ejemplo, si se incluye el caracter `=` se pueden incluir el nombre de las variables. Esto puede ser muy util para un código de depuración:

In [ ]:
```
from random import randint

for i in range(5):
    num = randint(1, 100)
    print(f"{i=:2} | {num=}")
```

Así también, se puede hacer conversiones de tipos de datos:

In [ ]:
```
for i in range(16):
    print(f"{i:2} = 0x{i:0x}")
```

In [ ]:
```
for i in range(16):
    print(f"{i:2} = b{i:0b}")
    #print(f"{i:2} = b{f'{i:0b}':0>4}")
```

Se pueden formatear los números para que contengan separadores de miles:

In [ ]:
```
num = 3200000   # 3_200_000
print(f"Monto: {num:,.2f} PEN")
```

También permite formatear la impresión de objetos `datetime`:

In [ ]:
```
from datetime import datetime

now = datetime.now()
print(f"Fecha: {now:%d/%m/%Y} | Hora: {now:%H:%M:%S}")
```

Decoradores

Los decoradores son funciones que toman una función como parametro de entrada y modifica su funcionamiento. Para entender los decoradores, hay que tener claro que las funciones son objetos de primera clase (*first classs objects*). Esto significa que se pueden tratar como un objeto o una variable y se pueden asignar de la misma forma:

In [ ]:
```
def suma(a, b):
    return a  + b
```

In [ ]:
```
S = suma
```

In [ ]:
```
print(type(S))
```

In [ ]:
```
print(S(10, 3))
```

Teniendo este concepto claro, primero definamos una función genérica que permite medir el tiempo de ejecución de la función `factores_primos`, utilizando una función de la librería time: `perf_counter`:

In [ ]:
```
from time import perf_counter

def test_performance(num: int) -> int:
    start_time = perf_counter()
    result = factores_primos(num)
    end_time = perf_counter()
    print(f"Tiempo requerido: {end_time - start_time} seg")
    return result
```

In [ ]:
```
test_performance(1928272636352)
```

Ahora, quisiera hace una versión genérica de la función anterior, esto es que le pueda pasar la función `factores_primos` como un parámetro. En este caso, la función `test_performance` va a tomar una función como parámetro de entrada y va a retornar a su vez una función como salida, que será una versión modificada de la función de entrada que llamaremos `wrapper`:

In [ ]:
```
def test_performance(func:Callable[..., Any]) -> Callable[..., Any]:
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        start_time = perf_counter()
        result = func(*args, **kwargs)
        end_time = perf_counter()
        print(f"Tiempo requerido: {end_time - start_time} seg")
        return result

    return wrapper
```

In [ ]:
```
wrapper = test_performance(factores_primos)
value = wrapper(18272672562)
print(value)
```

Como se observa, ahora tenemos una función, `test_performance` que puede probar la performance en el tiempo de cualquier función. En Python, existe una notación que permite ahorrar la celda de código anterior, que es la notación *at* (@) de los decoradores, esto es que "decora" la función original con una nueva funcionalidad:

In [ ]:
```
@test_performance
def factores_primos(n: int) -> list[int]:
    '''factores primos(n)   Funcion que retorna los factores primos de un numero

    Parametros:
        - n: int

    Uso:
        factores_primos(12)  -> [2, 2, 3]'''
    if n < 1 or not isinstance(n, int):
```

```
        return []

    div = 2
    factores = []
    # Lazo que encuentra los divisores del numero (factores)
    while n != 1:    # is not
        # Si se tiene un factor, se guarda en la lista de salida
        if n % div == 0:
            factores.append(div)
            n //=  div   # n = n // div
        else:
            # ...de lo contrario se pasa a considerar el siguiente posible divisor
            div += 1     # div = div + 1

    return factores
```

In [ ]:
```
factores = factores_primos(187828272)
print(factores)
```

¿Qué tan útiles pueden ser los decoradores? Depende de la complejidad del código. En general, no se recomienda su uso para agregar funcionalidades complejas a funciones existentes, pues la facilidad en la sintaxis (colocar @ y el nombre de la función que agrega funcionalidades nuevas) oculta lo que esta haciendo esta función, lo que hace que el código pierda legibilidad.

Quiza su uso más popular es como modificador de funciones a partir de otras librerías (como en el caso de las Frameworks Web como Flask o Django) o como control de atributos en las clases (como @property).

La idea principal en la definición de un decorador es la capacidad de agregar una funcionalidad nueva a una función sin tener que cambiar el código de la función original. Por ejemplo, si se tiene una función `get_mean_voltages` que retorna el valor promedio de una cadena de caracteres con un formato especifico que esta siendo recibido por el puerto serial (por ejemplo, de un sensor conectado a un Arduino), un decorador le puede agregar la funcionalidad de ver la data recibida para depuración.

In [ ]:
```
from typing import Callable

def show_voltages(func: Callable[[str], float]) -> Callable:
    def wrapper(str_data: str):
        print(f"Data In: {str_data}")
        result = func(str_data)
        return result

    return wrapper
```

In [ ]:
```
@show_voltages
def get_mean_voltage(values: str) -> float:
    voltages = tuple(map(lambda x: float(x), values.split(",")))
    return sum(voltages) / len(voltages)
```

In [ ]:
```
get_mean_voltage("12.4,14.2,14.3,13.4")
```

logging

Indicar eventos y errores con una impresión en el terminal es una costumbre usual y útil, aunque no necesariamente práctica. Esto es una forma de registro (*logging* en inglés) para depurar código. La forma recomendada de realiza esta tarea es utilizando el módulo `logging`.

In [ ]:
```
import logging

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(me
```

El estilo de formato es bastante antiguo aunque no hay que preocuparse mucho al respecto. Es solo la definición inicial del formato del registro. Lo importante a recordar es que cuando sucede un evento y esta se registra, se crea un objeto `LogRecord` que contiene información sobre el evento. La función `basicConfig()` permite especificaar que detalles sobre el objeto `LogRecord` serán visibles.

In [ ]:
```
def factorial(n: int) -> int:
    '''factorial(n)     Retorna el factorial de n (forma recursiva)
    '''
    logging.debug("Inicio de la funcion")

    if not isinstance(n, int):
        raise TypeError("El valor 'n' debe ser un 'int'")

    if n < 0:
        raise ValueError("El valor 'n' debe ser un 'int' >= 0")

    if n == 0:
        logging.debug("Recursividad: Caso Base factorial(0) = 1")
        return 1
    else:
        logging.debug(f"Recursividad: factorial({n}) * factorial({n-1})")
        return n * factorial(n-1)
```

In [ ]:
```
factorial(5)
```

Los detalles mostrados en los mensajes de logging son mucho más detallados que un print personalizado. Además, se puede llenar un programa de lineas `logging.debug` sin preocupaciones ya que luego se pueden desahabilitar todas con una simple instrucción (mientras que por otro lado se tendría que eliminar todas las funciones print que muestran mensajes de depuración). El print debe de reservarse para los mensajes a los usuarios; los logging para los programadores.

Los niveles de logging funcionan de manera jerárquica:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

Si se especifica en el formato de registro `logging.basicConfig(level=logging.ERROR)` solo se mostrarán los eventos de novel ERROR y CRITIAL, más no los de nivel inferior. La instrucción `logging.disable()` permite deshabilitar los registros a partir de un nivel, asi

que para deshabilitar todos los mensajes de logging se puede llamar a la operación anterior con el nivel más bajo:

In [ ]:
```
logging.disable(logging.DEBUG)
```

In [ ]:
```
factorial(5)
```

Uno de los usos más útiles del módulo `logging` es el registro sobre un archivo. Para esto será necesaria la siguiente especificación en la definición del registro:

```
logging.basicConfig(filename='events.log', level=logging.DEBUG,
format='%(asctime)s - %(levelname)s - %(message)s')
```

- NOTA: El registro de eventos está asociado a la sesión son el kernel, por lo que su uso en un ambiente como Jupyter Notebook no esta recomendado ya que en todo momento se mantiene la sesión del kernel activa, a diferencia de un script que se ejecuta en un terminal.

In [ ]:
```

```