

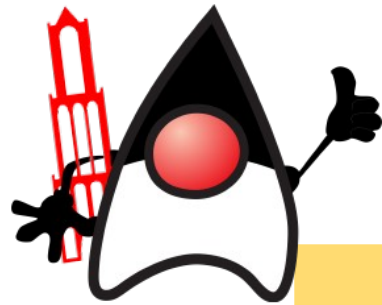
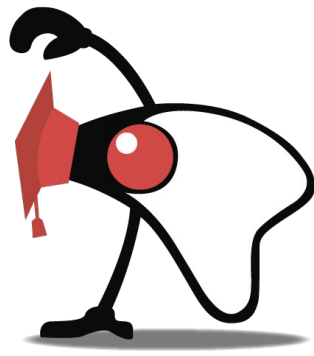
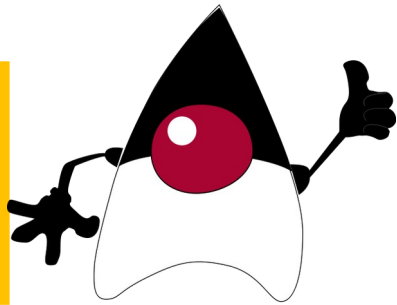


<codoa  
codo/>



# Curso FullStack Python

Codo a Codo 4.0



# Python

## ***Parte 8***



# Errores y excepciones

En algunas ocasiones nuestros programas pueden fallar ocasionando su **detención**. Ya sea por errores de sintaxis o de lógica, tenemos que ser capaces de detectar esos momentos y tratarlos debidamente para prevenirlos.

## Errores

Los errores detienen la ejecución del programa y tienen varias causas. Para poder estudiarlos mejor vamos a provocar algunos intencionadamente y ver lo que nos muestra la terminal.

### Errores de sintaxis

Identificados con el código **SyntaxError**, son los que podemos apreciar repasando el código, por ejemplo al dejarnos de cerrar un paréntesis:

```
print("Hola"
```

```
File "<ipython-input-1-8bc9f5174855>", line 1
    print("Hola"
          ^
SyntaxError: unexpected EOF while parsing
```

terminal



*errores.py*

*El intérprete reproduce la línea responsable del error y muestra una pequeña "flecha" que apunta al primer lugar donde se detectó el error.*

## Errores de nombre

Se producen cuando el sistema interpreta que debe ejecutar alguna función, método... pero no lo encuentra definido. Devuelven el código **NameError**:

```
pint("Hola")
```

```
<ipython-input-2-155163d628c2> in <module>()  
----> 1 pint("Hola")  
  
NameError: name 'pint' is not defined
```

terminal

*La mayoría de errores sintácticos y de nombre los identifican los editores de código antes de la ejecución, pero existen otros tipos que pasan más desapercibidos.*

## Errores semánticos

Estos errores son muy difíciles de identificar porque van ligados al sentido del funcionamiento y **dependen de la situación**. Algunas veces pueden ocurrir y otras no.

La mejor forma de prevenirlos es programando mucho y aprendiendo de tus propios fallos, la experiencia es la clave. Veamos un par de ejemplos:

### Ejemplo pop() con lista vacía:

Si intentamos sacar un elemento de una lista vacía, algo que no tiene mucho sentido, el programa dará fallo de tipo `IndexError`. Esta situación ocurre sólo durante la ejecución del programa, por lo que los editores no lo detectarán:

```
valores = []  
valores.pop()
```

```
IndexError: pop from empty list
```

terminal

## Errores semánticos (continuación)

Para prevenir el error deberíamos comprobar que una lista tenga como mínimo un elemento antes de intentar sacarlo, algo factible utilizando la función **len()**:

```
valores = []  
if len(valores) > 0:  
    valores.pop()
```

## Ejemplo lectura de cadena y operación sin conversión a número

Cuando leemos un valor con la función **input()**, este **siempre** se obtendrá como una **cadena de caracteres**. Si intentamos operarlo directamente con otros números tendremos un fallo **TypeError** que tampoco detectan los editores de código:

```
n = input("Ingrese un número: ")  
m = 4  
print("{} / {} = {}".format(n, m, n/m))
```

Introduce un número: 4

terminal

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-12-85bb893ab3e3> in <module>()  
----> 1 print("{} / {} = {}".format(n, m, n/m))  
  
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

## Errores semánticos (continuación)

Como ya sabemos este error se puede prevenir transformando la cadena a entero o flotante:

```
n = float(input("Ingresa un número: "))
m = 4
print("{} / {} = {}".format(n, m, n/m))
```

```
Ingresa un número: 12
12.0/4 = 3.0
```

terminal

Sin embargo no siempre se puede prevenir, como cuando se introduce una cadena que no es un número.

Como podemos suponer, es difícil prevenir fallos que ni siquiera nos habíamos planteado que podían existir. Por suerte para esas situaciones existen las **excepciones**.

```
Introduce un número: aaa
```

terminal

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-14-c0e7fd4a26a9> in <module>()
----> 1 n = float(input("Introduce un número: "))
      2 m = 4
      3 print("{} / {} = {}".format(n, m, n/m))

ValueError: could not convert string to float: 'aaa'
```

# Excepciones

Las **excepciones** son bloques de código que nos permiten continuar con la ejecución de un programa **pese a que ocurra un error**.

Continuando con el ejemplo anterior, teníamos el caso en que leíamos un número por teclado, pero el usuario no introduce un número:

```
n = float(input("Ingrese un número: "))
m = 4
print("{} / {} = {}".format(n, m, n/m))
```

Introduce un número: aaa

terminal

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-14-c0e7fd4a26a9> in <module>()
----> 1 n = float(input("Introduce un número: "))
      2 m = 4
      3 print("{} / {} = {}".format(n, m, n/m))

ValueError: could not convert string to float: 'aaa'
```

# Tipos de excepciones

## Exception

ArithmeticError

AssertionError

AttributeError

BaseException

BufferError

ChildProcessError

ConnectionAbortedError

ConnectionError

ConnectionRefusedError

ConnectionResetError

DeprecationWarning

EOFError

EnvironmentError

FileExistsError

FileNotFoundError

FloatingPointError

IOError

ImportError

IndentationError

IndexError

InterruptedError

KeyError

KeyboardInterrupt

ModuleNotFoundError

NameError

NotImplementedError

PermissionError

RecursionError

ReferenceError

RuntimeError

StopIteration

TypeError

UnboundLocalError

UnicodeDecodeError

UnicodeEncodeError

UnicodeError

UnicodeTranslateError

UnicodeWarning

UserWarning

ValueError

Warning

ZeroDivisionError



# Excepciones: Bloques try - except

Para prevenir el fallo debemos poner el código propenso a errores en un bloque **try** y luego encadenar un bloque **except** para tratar la situación excepcional mostrando que ha ocurrido un fallo:

```
try:
    n = float(input("Ingrese un número: "))
    m = 4
    print("{} / {} = {}".format(n, m, n / m))
except:
    print("Ha ocurrido un error, introduzca un número")
```

```
Ingrese un número: hola
Ha ocurrido un error, introduzca un número
```

terminal

*Como vemos esta forma nos permite controlar situaciones excepcionales que generalmente darían error y en su lugar mostrar un mensaje o ejecutar una pieza de código alternativo.*



*excepciones-try-except.py*

Podemos aprovechar las **excepciones** para forzar al usuario a introducir un número haciendo uso de un bucle **while**, repitiendo la lectura por teclado hasta que lo haga bien y entonces romper el bucle con un **break**:

```
while(True):  
    try:  
        n = float(input("Ingrese un número: "))  
        m = 4  
        print("{} / {} = {}".format(n,m,n/m))  
        break # Importante romper la iteración si todo ha salido bien  
    except:  
        print("Ha ocurrido un error, introduzca un número")
```

```
Ingrese un número: a  
Ha ocurrido un error, introduzca un número  
Ingrese un número: hola  
Ha ocurrido un error, introduzca un número  
Ingrese un número: 12s  
Ha ocurrido un error, introduzca un número  
Ingrese un número: 10  
10.0/4 = 2.5
```

terminal

## Bloque else

Es posible encadenar un bloque **else** después del **except** para comprobar el caso en que **todo funcione correctamente** (*no se ejecuta la excepción*).

El bloque **else** es un buen momento para romper la iteración con *break* si todo funciona correctamente:

```
while(True):  
    try:  
        n = float(input("Ingrese un número: "))  
        m = 4  
        print("{} / {} = {}".format(n,m,n/m))  
    except:  
        print("Ha ocurrido un error, introduzca un número")  
    else:  
        print("Todo ha funcionado correctamente")  
        break # Importante romper la iteración si todo ha salido bien
```

```
Ingrese un número: 15  
15.0/4 = 3.75  
Todo ha funcionado correctamente
```

terminal

## Bloque finally

Por último es posible utilizar un bloque **finally** que se ejecute al final del código, **ocurra o no ocurra un error**:

```
while(True):  
    try:  
        n = float(input("Ingrese un número: "))  
        m = 4  
        print("{} / {} = {}".format(n,m,n/m))  
    except:  
        print("Ha ocurrido un error, introduzca un número")  
    else:  
        print("Todo ha funcionado correctamente")  
        break # Importante romper la iteración si todo ha salido bien  
    finally:  
        print("Fin de la iteración") # Siempre se ejecuta
```

```
Ingrese un número: hola  
Ha ocurrido un error, introduzca un número  
Fin de la iteración  
Ingrese un número: 18  
18.0/4 = 4.5  
Todo ha funcionado correctamente  
Fin de la iteración
```

terminal

# Excepciones: Otros ejemplos

La mayoría de las excepciones no son gestionadas por el código, y resultan en mensajes de error como los mostrados aquí.

La última línea de los mensajes de error indica qué ha sucedido. Hay excepciones de diferentes tipos, y el tipo se imprime como parte del mensaje: los tipos en el ejemplo son:

**ZeroDivisionError, NameError y TypeError.**

```
print(10 * (1/0))
```

```
ZeroDivisionError: division by zero
```

terminal

```
print(4 + spam*3)
```

```
NameError: name 'spam' is not defined
```

terminal

```
print('2' + 2)
```


```
TypeError: can only concatenate str (not "int") to str
```

terminal

La cadena mostrada como tipo de la excepción es el nombre de la excepción predefinida que ha ocurrido. Esto es válido para todas las excepciones predefinidas del intérprete, pero no tiene por qué ser así para excepciones definidas por el usuario (aunque es una convención útil). Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee información basado en el tipo de la excepción y qué la causó.

# Excepciones: Otros ejemplos



```
1  def f(x, y):  
2      return x / y  
3  
4  def g(x, y):  
5      return f(x, y)  
6  
7  z = g(5, 0)
```

```
Traceback (most recent call last):  
  File "programa.py", line 7, in <module>  
    z = g(5, 0)  
  File "programa.py", line 5, in g  
    return f(x, y)  
  File "programa.py", line 2, in f  
    return x / y  
ZeroDivisionError: division by zero
```

**terminal**



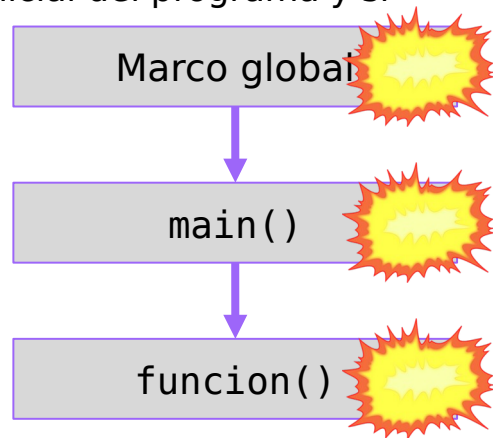
*excepciones-2.py*

# Propagación de excepciones

Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no la maneja, la excepción se **propaga** hacia la función que la invocó, si esta otra tampoco la maneja, la excepción continua propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa.

```
1 def funcion(x, y):
2     print("antes")
3     div = x/y
4     print("después")
5     return div
6
7 def main():
8     x = float(input('x: '))
9     y = float(input('y: '))
10    print(funcion(x, y))
11    print("listo")
12
13 main()
```

*Si introducimos un 0 como divisor nos devolverá un error de división por 0 que se propaga a las demás funciones.*



```
Traceback (most recent call last):
  File ... line 13, in <module> main()
  File ... line 10, in main print(funcion(x, y))
  File ... line 3, in función div = x/y
ZeroDivisionError: float division by zero
```


**terminal**



**excepciones-3-  
propagacion.py**

# Propagación de excepciones

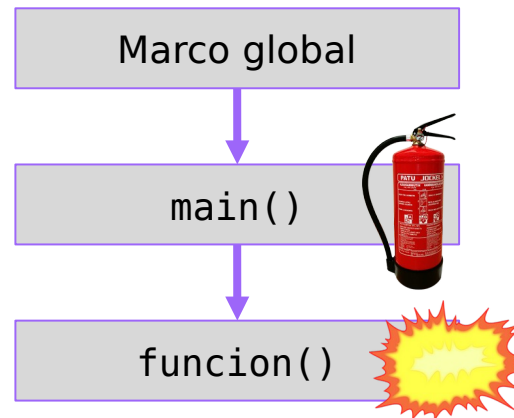
La forma de resolver el problema es incorporando un bloque **try...except** de modo tal que si se introduce un 0 como divisor aparecerá *"Algo salió mal"*:



```
1 def funcion(x, y):
2     print("antes")
3     div = x/y
4     print("después")
5     return div
6
7 def main():
8     x = float(input('x: '))
9     y = float(input('y: '))
10    try:
11        print(funcion(x, y))
12    except:
13        print("Algo salió mal")
14        print("listo")
15
16 main()
```

```
x: 2
y: 0
antes
Algo salió mal
listo
```

**terminal**



*excepciones-4.py*



# Excepciones múltiples

En una misma pieza de código pueden ocurrir **muchos errores distintos** y quizá nos interese actuar de forma diferente en cada caso.

Para esas situaciones algo que podemos hacer es asignar una excepción a una variable. De esta forma es posible analizar el tipo de error que sucede gracias a su identificador:

```
try:
    n = input("Ingrese un número: ") # No transformamos a número
    5/n
except Exception as e: # Guardamos la excepción como una variable e
    print("Ha ocurrido un error =>", type(e).__name__)
```

```
Ingrese un número: 12
Ha ocurrido un error => TypeError
```

terminal

Cada error tiene un identificador único que curiosamente se corresponde con su tipo de dato. Aprovechándonos de eso podemos mostrar la clase del error, dentro de **except**, utilizando la sintaxis:

```
print(type(e))
```

```
<class 'TypeError'>
```

terminal

Es similar a conseguir el tipo (o clase) de cualquier otra variable o valor literal:

```
print(type(1))  
print(type(3.14))  
print(type([]))  
print(type(()))  
print(type({}))
```

```
<class 'int'>  
<class 'float'>  
<class 'list'>  
<class 'tuple'>  
<class 'dict'>
```

terminal

Como vemos siempre nos indica "**class**" delante. Eso es porque en Python todo son clases. Lo importante ahora es que podemos mostrar solo el nombre del tipo de dato (la clase) consultando su propiedad especial **name** de la siguiente forma:

```
print(type(e).__name__)  
print(type(1).__name__)  
print(type(3.14).__name__)  
print(type([]).__name__)  
print(type(()).__name__)  
print(type({}).__name__)
```

```
TypeError  
int  
float  
list  
tuple  
dict
```

terminal

Gracias a los identificadores de errores podemos crear múltiples comprobaciones, siempre que dejemos en último lugar la excepción por defecto ***Exception*** que engloba cualquier tipo de error (si la pusiéramos al principio las demás excepciones nunca se ejecutarán):

```
try:
    n = float(input("Ingrese un número divisor: "))
    5/n
except TypeError:
    print("No se puede dividir el número entre una cadena")
except ValueError:
    print("Debes introducir una cadena que sea un número")
except ZeroDivisionError:
    print("No se puede dividir por cero, prueba otro número")
except Exception as e:
    print("Ha ocurrido un error no previsto", type(e).__name__)
```

```
Ingrese un número divisor: 0
No se puede dividir por cero, prueba otro número
```

terminal

```
Ingrese un número divisor: hola
Debes introducir una cadena que sea un número
```

terminal



*excepciones-multiples.py*

# Invocación de excepciones

En algunas ocasiones quizá nos interesa llamar a un error manualmente, ya que un print común no es muy elegante:

```
def mi_funcion(algo=None):  
    if algo is None:  
        print("Error! No se permite un valor nulo (con un print)")  
  
mi_funcion()
```

Error! No se permite un valor nulo (con un print)

terminal

## Instrucción raise

Gracias a **raise** podemos *lanzar una excepción* pasándole el identificador. Luego simplemente podemos añadir un **except** para tratar esta excepción que hemos lanzado:

```
def mi_funcion(algo=None):  
    try:  
        if algo is None:  
            raise ValueError("Error! No se permite un valor nulo")  
    except ValueError:  
        print("Error! No se permite un valor nulo (desde la excepción)")  
  
mi_funcion()
```

Error! No se permite un valor nulo (desde la excepción)

terminal

## Instrucción raise (continuación)

- Se utiliza para crear una excepción.
- Si no se detalla el tipo de excepción, se relanza la última excepción producida.
- Como desarrollador, se puede elegir lanzar una excepción si se produce una condición.
- Para “lanzar” una excepción, debemos usar la palabra clave **raise**.

```
1  x = -1
2  if x < 0:
3      raise Exception("No se aceptan números menores a cero.")
```

- Puede definir qué tipo de error generar y el texto para imprimir al usuario.
- **Ejemplo:** lanzar una excepción del tipo **TypeError** si x no es un entero:

```
1  x = "Hola Mundo"
2  if not type(x) is int:
3      raise TypeError("Sólo se admiten valores enteros.")
```



*raise.py*

**Fuente:** [https://www.w3schools.com/python/gloss\\_python\\_raise.asp](https://www.w3schools.com/python/gloss_python_raise.asp)

## Instrucción assert

- La declaración **assert** es una forma de generar una excepción si no ocurre la afirmación esperada.
- La excepción **AssertionError** se genera cuando una declaración assert no se cumple.
- **Ejemplo:** crear una función para validar el ingreso de números naturales.

```
def ingresarNatural(msj):  
    while True:  
        try:  
            valorReal= float(input(msj))  
            valor= int(valorReal)  
            assert (valor == valorReal), "Error: debe ingresar un valor entero."  
            assert (valor > 0), "Error: debe ingresar un valor positivo."  
            break  
        except AssertionError as error:  
            print(error)  
        except (ValueError):  
            print("Error: ha ingresado un valor que no es numérico.")  
        except:  
            print("Error inesperado.")  
    return valor
```

sigue...

```
# Programa principal  
def __main__():  
    msj= "Ingrese un número natural: "  
    num= ingresarNatural(msj)  
    print(num)  
  
if __name__ == '__main__':  
    __main__()
```



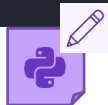
*assert.py*

## Instrucción `sys.exc_info()`

- La última cláusula **except** puede omitir el nombre de la excepción, a modo de “comodín”.
- Usar esto con extremo cuidado, ya que es muy fácil enmascarar un error de programación de esta manera.
- También se puede usar para imprimir un mensaje de error emitido por el sistema y luego volver a generar (re-lanzar) la excepción (permitiendo que quien llama también maneje la excepción).

```
import sys

try:
    arch= open('miArch.txt')
    txt = arch.readline()
    num = int(txt.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Error: no se pudo convertir el valor a entero.")
except:
    print("Error inesperado:", sys.exc_info()
[0]) # Me informa el tipo de excepción (class)
    raise # Re-lanza la excepción
```



`sys.exc_info().py`

# Ejercicios resueltos

Localizar el error en los siguientes bloques de código. Crear una excepción para evitar que el programa se bloquee y además explicar en un mensaje al usuario la causa y/o solución:

## Ejercicio 1

```
resultado = 10/0
```

```
try:
    resultado = 10/0
except:
    print("No es posible dividir entre cero")
```

Solución

```
No es posible dividir entre cero
```

terminal

## Ejercicio 2

```
lista = [1, 2, 3, 4, 5]
lista[10]
```

```
lista = [1, 2, 3, 4, 5]
try:
    lista[10]
except IndexError:
    print("El índice se encuentra fuera de rango")
```

Solución

```
El índice se encuentra fuera de rango
```

terminal



*ejercicios\_excepciones..py*



# Ejercicios resueltos

Localizar el error en los siguientes bloques de código. Crear una excepción para evitar que el programa se bloquee y además explicar en un mensaje al usuario la causa y/o solución:

## Ejercicio 3

```
colores = { 'rojo':'red', 'verde':'green', 'negro':'black' }  
colores['blanco']
```

```
colores = { 'rojo':'red', 'verde':'green', 'negro':'black' }, Solución  
try:  
    colores['blanco']  
except KeyError:  
    print("La clave del diccionario no se encuentra")
```

La clave del diccionario no se encuentra

terminal

# Ejercicios resueltos

Localizar el error en los siguientes bloques de código. Crear una excepción para evitar que el programa se bloquee y además explicar en un mensaje al usuario la causa y/o solución:

## Ejercicio 4

```
resultado = 15 + "20"
```

```
try:
    resultado = 15 + "20"
except TypeError:
    print("Sólo es posible sumar datos del mismo tipo")
```

Solución

```
Sólo es posible sumar datos del mismo tipo
```

terminal

# Excepciones - Resumen



```
try:
    <instrucciones>
except:
    <instrucciones>
```

```
try:
    <instrucciones>
except <tipo1>:
    <instrucciones>
except <tipo2>:
    <instrucciones>
except:
    <instrucciones>
```

```
try:
    <instrucciones>
except <tipo1> as <nombre>:
    <instrucciones>
except <tipo2> as <nombre>:
    <instrucciones>
except:
    <instrucciones>
```

```
try:
    <instrucciones>
except <tipo1> as <nombre>:
    <instrucciones>
except <tipo2> as <nombre>:
    <instrucciones>
except:
    <instrucciones>
finally:
    <instrucciones>
```

- Podemos capturar excepciones para evitar que el programa finalice por un error. **Bloque:** try... except
- Se ejecuta el **bloque try**. Si no ocurre ninguna excepción, el bloque except se saltea.
- Si ocurre una excepción durante la ejecución del bloque try, el resto del bloque se saltea. Si su tipo coincide con la excepción, se ejecuta el **bloque except**.
- Una declaración try puede tener más de un except.
- El último except puede omitir el tipo de dato (evitar su uso) se ejecuta ante cualquier error que ocurra en el bloque y no esté gestionado anteriormente.
- else: es opcional en un bloque try except. Sólo será ejecutada cuando todas las instrucciones del bloque try se ejecutaron en forma “normal”.
- finally: garantiza que un bloque de código siempre se ejecute, sin importar si hubo o no errores.

# Excepciones - Más ejemplos

Crear una función para validar el ingreso de un número positivo:

```
def ingresarPositivo(msj):  
    err= True  
    while err:  
        try:  
            num= float(input(msj))  
            if num>=0:  
                err=False  
            else:  
                print("Error: debe ingresar un valor positivo o igual a 0.")  
        except (ValueError):  
            print("Error: ha ingresado un valor no numérico.")  
        except:  
            print("Error inesperado.")  
        else:  
            if not err:  
                break  
    return num  
  
ingresarPositivo("Ingrese un número positivo: ")
```

**terminal**

```
Ingrese un número positivo: -1  
Error: debe ingresar un valor positivo o  
igual a 0.  
Ingrese un número positivo: hola  
Error: ha ingresado un valor no numérico.  
Ingrese un número positivo: 3
```



*excepciones-ingresarPositivo.py*

# Módulos

Los módulos son ficheros que contienen definiciones que se pueden importar en otros scripts para reutilizar sus funcionalidades.

Anteriormente vimos que crear un módulo en Python es tan sencillo como crear un script, sólo tenemos que añadir alguna función a un fichero con la extensión `.py`, por ejemplo **saludos.py**:

```
def saludar():  
    print("Hola, te estoy saludando desde la función saludar()")
```

Luego ya podremos utilizarlo desde otro script, por ejemplo **script.py**, en el mismo directorio haciendo un `import` y el nombre del módulo:

```
import saludos  
  
saludos.saludar()
```

También podemos importar funciones directamente, de esta forma ahorraríamos memoria. Podemos hacerlo utilizando la sintaxis **from import**:

```
from saludos import saludar  
  
saludar()
```



*Ver carpeta “modulos” y script.py*

Para importar ***todas las funciones*** con la sintaxis **from import** debemos poner un **asterisco**:



```
from saludos import *  
  
saludar()
```

Dicho esto, aparte de funciones también vimos que podemos reutilizar clases:

```
class Saludo():  
    def __init__(self):  
        print("Hola, te estoy saludando desde el __init__")
```

Igual que antes, tendremos que llamar primero al módulo para referirnos a la clase:

```
from saludos import Saludo  
  
s = Saludo()
```

El problema ocurre cuando queremos utilizar nuestro módulo desde un directorio distinto por ejemplo **test/script.py**.

```
ModuleNotFoundError: No module named 'saludos'
```

terminal

# Paquetes

Utilizar **paquetes** nos ofrece varias ventajas. En primer lugar nos permite unificar distintos módulos bajo un mismo nombre de paquete, pudiendo crear jerarquías de módulos y submódulos, o también subpaquetes.

Por otra parte nos permiten distribuir y manejar fácilmente nuestro código como si fueran librerías instalables de Python. De esta forma se pueden utilizar como módulos estándar desde el intérprete o scripts sin cargarlos previamente.

Para crear un paquete lo que tenemos que hacer es crear un fichero especial **init** vacío en el directorio donde tengamos todos los módulos que queremos agrupar. De esta forma cuando Python recorra este directorio será capaz de interpretar una jerarquía de módulos:

```
script.py  
paquete/
```

```
    __init__.py  
    saludos.py
```



*Ver carpeta “paquete” y script.py*

Ahora, si utilizamos un script **desde el mismo directorio** donde se encuentra el paquete podemos acceder a los módulos, pero esta vez refiriéndonos al paquete y al módulo, así que debemos hacerlo con **from import**:

```
from paquete.saludos import Saludos = Saludo()
```

*script.py*

Esta jerarquía se puede expandir tanto como queramos creando subpaquetes, pero siempre añadiendo el fichero **\_\_init\_\_** en cada uno de ellos:

```
script.py
paquete/
    __init__.py
    adios/
        __init__.py
        despedidas.py
    hola/
        __init__.py
        saludos.py
```

Este es el contenido de paquete/hola/saludos.py

```
def saludar():
    print("Hola, te estoy saludando desde la función saludar() " \
          "del módulo saludos")

class Saludo():
    def __init__(self):
        print("Hola, te estoy saludando desde el __init__ " \
              "de la clase Saludo")
```



Este es el contenido de paquete/adios/despedidas.py

```
def despedir():  
    print("Adiós, me estoy despidiendo desde la función despedir() " \  
          "del módulo despedidas")  
  
class Despedida():  
    def __init__(self):  
        print("Adiós, me estoy despidiendo desde el __init__ " \  
              "de la clase Despedida")
```

Ahora de una forma bien sencilla podemos ejecutar las funciones y métodos de los módulos de cada subpaquete desde el archivo **script.py** :

```
from paquete.hola.saludos import saludar  
from paquete.adios.despedidas import Despedida  
  
saludar()  
Despedida()
```

*script.py*

```
Hola, te estoy saludando desde la función saludar() del módulo saludos  
Adiós, me estoy despidiendo desde el __init__ de la clase Despedida
```

**terminal**

Más información en: <https://docs.hektorprofe.net/python/modulos-y-paquetes/paquetes/>

# Módulos esenciales



- **copy**: Se utiliza para crear copias de variables referenciadas en memoria, como colecciones y objetos.
- **collections**: Cuenta con diferentes estructuras de datos.
- **datetime**: Maneja tipos de datos referidos a las fechas/horas.
- **html, xml y json**: Permiten manejar cómodamente estructuras de datos html, xml y json. Son muy utilizados en el desarrollo web.
- **math**: Uno de los módulos más importantes de cualquier lenguaje, incluye varias funciones para trabajar matemáticamente.
- **random**: Permite generar contenidos aleatorios, escoger aleatoriamente valores y este tipo de cosas que hacen que un programa tenga comportamientos al azar. Es muy útil en el desarrollo de videojuegos y en la creación de pruebas.
- **sys**: Nos permite conseguir información del entorno del sistema operativo o manejarlo en algunas ocasiones, se considera un módulo avanzado e incluso puede ser peligroso utilizarlo sin conocimiento.
- **threading**: Se trata de otro módulo avanzado que sirve para dividir procesos en subprocesos gracias a distintos hilos de ejecución paralelos.
- **tkinter**: Es el módulo de interfaz gráfica de facto en Python.

# Módulo copy: copia de objetos

El módulo estándar **copy** permite crear copias de distintos objetos de Python, generalmente colecciones mutables (como las listas y los diccionarios) e instancias de clases, también mutables. Por ejemplo: consideremos la siguiente lista:

```
a = list(range(5))  
print(a) # [0, 1, 2, 3, 4]
```

Si por el motivo que fuese requiero crear una lista igual a la anterior, intuitivamente haría lo siguiente:

```
b = a  
print(b) #[0, 1, 2, 3, 4]
```

El problema con esta solución es que **no estamos creando dos listas** con los mismos elementos, sino que la lista es siempre una y a ella se puede acceder a través de dos nombres diferentes (**a** y **b**). Eso se comprueba sencillamente viendo cómo alterando los elementos de un objeto se refleja en el otro.

```
b.append(5)  
print(a) #[0, 1, 2, 3, 4, 5]  
del a[2]  
print(b) #[0, 1, 3, 4, 5]
```



*modulo-copy.py*

Sumado a que el operador **is** nos confirma que efectivamente se trata del mismo objeto:

```
print(a is b) #True
```

Pero en ocasiones realmente queremos crear dos objetos iguales aunque independientes el uno del otro, es decir, cada uno con su espacio asignado en la memoria. Allí es donde nos auxilia la función `copy()`.

```
from copy import copy

a = list(range(5))
b = copy(a)
print(a is b) #False

b.append(5)
del a[0]

print(a) #[0, 1, 2, 3, 4]
print(b) #[0, 1, 2, 3, 4, 5]
```

**Para ampliar este ejemplo:** <https://recursospython.com/guias-y-manuales/modulo-estandar-copy/>

# Módulo collections: colecciones de datos

El módulo integrado de colecciones nos provee otros tipos o mejoras de las colecciones clásicas.

## Contadores

Counter es una subclase de diccionario integrada en Python que se utiliza para realizar cuentas o conteos sobre listas, palabras... Podríamos resumir la función Counter como un contador en Python.

Lo primero que tenemos que hacer siempre que queramos utilizar **Counter** es importar el módulo:

```
from collections import Counter
```

Ahora podríamos contar los números dentro de una lista:

```
numeros = [1,2,3,4,1,2,3,1,2,1]
print(Counter(numeros)) #Counter({1: 4, 2: 3, 3: 2, 4: 1})
```

También nos sirve para contar las letras que hay dentro de una palabra:

```
print(Counter("palabra"))
#Counter({'a': 3, 'p': 1, 'l': 1, 'b': 1, 'r': 1})
```



Para contar las palabras que hay dentro de un conjunto de palabras debemos usar la función **split()**, que nos separa ese conjunto de palabras en una lista de palabras:

```
coches= "mercedes ferrari bmw bmw ferrari bmw"
print(Counter(coches.split()))
#Counter({'bmw': 3, 'ferrari': 2, 'mercedes': 1})
```

**most\_common()** nos permite obtener una lista ordenada por repeticiones:

```
animales= "perro gato jirafa jirafa gato jirafa"
a= Counter(animales.split())
print(a.most_common(1)) #Elemento más repetido
print(a.most_common(2)) #Las dos palabras más repetidas
print(a.most_common()) #Ordenado por número de repeticiones
```

```
[('jirafa', 3)]
[('jirafa', 3), ('gato', 2)]
[('jirafa', 3), ('gato', 2), ('perro', 1)]
```

termina

**Fuente:** <https://estadisticamente.com/modulo-collections-python-contadores/>

# Módulo datetime

Este módulo contiene las clases **time** y **datetime** esenciales para manejar tiempo, horas y fechas.

**Clase datetime:** Esta clase permite crear objetos para manejar fechas y horas:

```
from datetime import datetime

dt= datetime.now()           # Fecha y hora actual

print(dt)                   # Imprime fecha y hora actual

print("Año:", dt.year)      # Año
print("Mes:", dt.month)     # Mes
print("Dia:", dt.day)       # Dia

print("Hora:", dt.hour)     # Hora
print("Minuto:", dt.minute) # Minuto
print("Segundo:", dt.second) # Segundo
print("Microsegundo:", dt.microsecond) # Microsegundo


print("{}: {}: {}".format(dt.hour, dt.minute, dt.second))
print("{} / {} / {}".format(dt.day, dt.month, dt.year))
```

terminal

```
2021-06-29
15:56:31.130223
Año: 2021
Mes: 6
Dia: 29
Hora: 15
Minuto: 56
Segundo: 31
Microsegundo: 130223
15:56:31
29/6/2021
```



modulo-  
datetime.py



Es posible crear un datetime manualmente pasando los parámetros (year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None). Sólo son **obligatorios** el **año**, el **mes** y el **día**.

```
dt= datetime(2021,9,28, 11,23)
print(dt) #2021-09-28 11:23:00
```


No podemos cambiar un atributo simplemente asociándole a un valor porque son de **solo lectura**.

```
dt.year = 3000 # AttributeError: attribute 'year' of 'datetime.date'
               objects is not writable
```

Debemos usar el método **replace()**:

```
dt = dt.replace(year=3000)
print(dt) #3000-09-28 11:23:00
```

**Para ampliar:** <https://docs.hektorprofe.net/python/modulos-y-paquetes/modulo-datetime/>





# Módulo math

Este módulo contiene un buen puñado de funciones para manejar números, hacer redondeos, sumatorios precisos, truncamientos, además de constantes.

```
import math # Importamos el módulo math
```

## Redondeos

```
print(math.floor(3.99)) # Redondeo a la baja (suelo)
print(math.ceil(3.01)) # Redondeo al alta (techo)
```

```
3
4 terminal
```

## Sumatoria mejorada

```
numeros = [0.9999999, 1, 2, 3]
print(math.fsum(numeros)) # 6.9999999
```

## Truncamiento

```
print(math.trunc(123.45)) # 123
```

## Potencias y raíces

```
print(math.pow(2, 3)) # Potencia con flotante
print(math.sqrt(9)) # Raíz cuadrada (square root)
```

```
8.0
3.0 terminal
```

## Constantes

```
print(math.pi) # Constante pi
print(math.e) # Constante e
```

```
3.141592653589793
2.718281828459045 terminal
```



*modulo-math.py*

# Módulo random

## Aleatoriedad

Este módulo contiene funciones para generar números aleatorios:

```
import random

# Flotante aleatorio >= 0 y < 1.0
print(random.random())

# Flotante aleatorio >= 1 y <10.0
print(random.uniform(1,10))

# Entero aleatorio de 0 a 9, 10 excluído
print(random.randrange(10))

# Entero aleatorio de 0 a 100
print(random.randrange(0,101))

# Entero aleatorio de 0 a 100 cada 2 números, múltiplos de 2
print(random.randrange(0,101,2))

# Entero aleatorio de 0 a 100 cada 5 números, múltiplos de 5
print(random.randrange(0,101,5))
```

```
0.024469998027175754
9.309153792304896
3
2
50
70
```

terminal



*modulo-random.py*

## Muestras

También tiene funciones para tomar muestras:

```
# Letra aleatoria
print(random.choice('Hola mundo'))

# Elemento aleatorio
random.choice([1,2,3,4,5])

# Dos elementos aleatorios
random.sample([1,2,3,4,5], 2)
```

```
a
5
[1, 2]
```

terminal

## Mezclas

Y para mezclar colecciones:

```
# Barajar una lista, queda guardado
lista = [1,2,3,4,5]
random.shuffle(lista)
print(lista)
```

```
[2, 4, 3, 1, 5]
```

terminal