Herencia

La **herencia** es la capacidad que tiene una clase de heredar los atributos y métodos de otra, algo que nos permite **reutilizar código**.

Partiremos de una clase sin herencia con muchos atributos y la iremos descomponiendo en otras clases más simples que nos permitan trabajar mejor con sus datos.

Ejemplo sin herencia

Si partimos de una clase que contenga todos los atributos, quedaría más o menos así:

Código

```
class Producto:
    def __init__ (self, referencia, tipo, nombre, descripcion,precio,productor="",
                  distribuidor="", isbn="", autor=""):
        self.referencia = referencia
        self.tipo = tipo
        self.nombre = nombre
        self.descripcion = descripción
        self.precio = precio
        self.productor = productor
        self.distribuidor = distribuidor
        self.isbn = isbn
        self.autor = autor
```

Resultado

Obviamente esto es un despropósito, así que veamos cómo aprovecharnos de la herencia para mejorar el planteamiento.

Superclases

Así pues, la idea de la herencia es identificar una **clase base** (la **superclase**) con los atributos comunes y luego crear las demás clases heredando de ella (las **subclases**) **extendiendo** sus campos específicos. En nuestro caso esa clase sería el **Producto** en sí mismo.

```
class Producto:
    def __init__ (self,referencia,nombre,descripcion,precio):
        self.referencia = referencia
        self.nombre = nombre
        self.descripcion = descripción
        self.precio = precio

    def __str__ (self):
        return "Producto: {} - {} - {} -
   {}".format(self.referencia,self.nombre,self.descripcion,self.precio)

    def rebajar_producto(self,rebaja):
        self.precio = self.precio - rebaja
```

Subclases

Para heredar los atributos y métodos de una clase en otra sólo tenemos que pasarla entre paréntesis en la definición:

```
class Adorno(Producto):
    pass

adorno = Adorno(2034, "Vaso adornado", 15, "Vaso de porcelana")
print(adorno)
```

```
REFERENCIA      2034
NOMBRE          Vaso adornado
PVP             15
DESCRIPCIÓN     Vaso de porcelana
```

Como se puede apreciar es posible utilizar el comportamiento de una superclase sin definir nada en la subclase.

Ejemplo para importar contenido de otras clases

Se importará el contenido de otro archivo .py agregando en la cabecera del archivo **from** (nombre del archivo sin la extensión .py) **import** (Clase a importar).

Python nos permite utilizar un acceso directo a la superclase llamado ***super()***. Hacerlo de esta forma además nos permite llamar cómodamente los métodos o atributos de la superclase sin necesidad de especificar el self, pero ojo, **sólo se aconseja utilizarlo cuando tenemos una única superclase**:

Código

```
from producto import Producto

class Adorno(Producto):
    def __init__(self,referencia,nombre,descripcion,precio,estilo):
        super().__init__(referencia, nombre, descripcion, precio)
        self.estilo = estilo

    def __str__ (self):
        return "{} - {} - {} -
   {}".format(self.referencia,self.nombre,self.descripcion,self.estilo)
```

Respecto a las demás subclases como se añaden algunos atributos, podríamos definirlas de esta forma:

```
from producto import Producto


class Libro(Producto):

    def __init__(self,referencia,nombre,descripcion,precio,isbn,autor):
        super().__init__(referencia, nombre, descripcion,precio)
        self.isbn = isbn
        self.autor = autor

    def __str__ (self):
        return "{} - {} - {} - {} - {} -
   {}".format(self.referencia,self.nombre,self.descripcion,self.precio,self.isbn,self.autor)
```

Ahora para utilizarlas simplemente tendríamos que establecer los atributos después de crear los objetos:

```
alimento = Alimento(2035, "Botella de Aceite de Oliva", 5, "250 ML")
alimento.productor = "La Aceitera"
alimento.distribuidor = "Distribuciones SA"
print(alimento)

libro = Libro(2036, "Cocina Mediterránea",9, "Recetas sanas y buenas")
libro.isbn = "0-123456-78-9"
libro.autor = "Doña Juana"
print(libro)
```

```
REFERENCIA      2035
NOMBRE          Botella de Aceite de Oliva
PVP             5
DESCRIPCIÓN     250 ML
PRODUCTOR       La Aceitera
DISTRIBUIDOR    Distribuciones SA

REFERENCIA      2036
NOMBRE          Cocina Mediterránea
PVP             9
DESCRIPCIÓN     Recetas sanas y buenas
ISBN            0-123456-78-9
AUTOR           Doña Juana
```

Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Definición de la función main()

Si bien, en un apartado anterior, ya hemos trabajado este tema, desarrollemos un poco más algo que está íntimamente ligado al modo de funcionamiento del intérprete Python:

Cuando el intérprete lee un archivo de código, **ejecuta todo el código global que se encuentra en él**. Esto implica crear objetos para toda función o clase definida y variables globales.

Todo módulo (archivo de código) en Python tiene un atributo especial llamado __name__ que define el espacio de nombres en el que se está ejecutando. Es usado para identificar de forma única un módulo en el sistemade importaciones.

Por su parte "__main__" es el nombre del ámbito en el que se ejecuta el códigode nivel superior (tu programa principal).

El intérprete pasa el valor del atributo __name__ a la cadena '__main__' si el módulo se está ejecutando como programa principal (cuando lo ejecutas llamando al intérprete en la terminal con python my_modulo.py, haciendo dobleclic en él, ejecutándolo en el intérprete interactivo, etc.).

Si el módulo no es llamado como programa principal, sino que es importado desde otro módulo, el atributo __name__ pasa a contener el nombre del archivo en sí.

Ventajas de usar def main()

Otros lenguajes (como C y Java) tienen una función main() que se llama cuando se ejecuta el programa. Utilizando este if, podemos hacer que Python se comporte como ellos, lo cual es más familiar para muchas personas.

El código será más fácil de leer y estará mejor organizado.

Será posible ejecutar pruebas en el código.

Podemos importar ese código en un shell de python y probarlo/depurarlo/ejecutarlo.

Variables dentro def main() son **locales**, mientras que las que están afuerason **globales**. Esto puede introducir algunos errores y comportamientos inesperados.

Permite ejecutar la función si se importa el archivo como un módulo.

Código (main.py)

```
from producto import Producto
from alimento import Alimento
from adorno import Adorno
from libro import Libro

def main():
    producto = Producto(2033,"Producto Genérico","1 kg",50)
    alimento = Alimento(2035, "Botella de Aceite de Oliva","250
    ML",50,"Marca","Distribuidor")
    adorno = Adorno(2034, "Vaso adornado", "Vaso de porcelana",34,"De Mesa")
    libro = Libro(2036, "Cocina Mediterránea", "Recetas buenas",75,"0-123456-
    789","Autor")

if __name__ == "__main__":
    main()
```

Luego en los ejercicios mostraremos como podemos sobrescribir el constructor de una forma eficiente para inicializar campos extra, por ahora veamos cómo trabajar con estos objetos de distintas clases de forma común.

Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Trabajando en conjunto

Gracias a la flexibilidad de Python podemos manejar objetos de distintas clases masivamente de una forma muy simple.

Vamos a empezar creando una lista con nuestros tres productos de subclases distintas:

Código

```
productos = [adorno, alimento]
productos.append(libro)
```

Ahora si queremos recorrer todos los productos de la lista podemos usar un bucle *for*:

Código

```
for producto in productos:
    print(producto)
```

También podemos acceder a los atributos, siempre que sean compartidos entre todos los objetos:

Código

```
for producto in productos:
    print(producto.referencia, producto.nombre)
```

Si un objeto no tiene el atributo al que queremos acceder nos dará error:

Código

```
for producto in productos:
    print(producto.autor)
```

Por suerte podemos hacer una comprobación con la función *isinstance()* para

determinar si una instancia es de una determinado clase y así mostrar unos atributos u otros:

Código

```
for producto in productos:
    if(isinstance(producto, Adorno)):
        print(producto.referencia, producto.nombre)
    elif(isinstance(producto, Alimento)):
        print(producto.referencia, producto.nombre, producto.productor)
    elif(isinstance(producto, Libro)):
        print(producto.referencia, producto.nombre, producto.isbn)
```

Aunque esta no será la forma que utilizaremos a futuro ya que nos valdremos de una propiedad muy importante en objetos: Poliformismo.

Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Polimorfismo

El polimorfismo es una propiedad de la herencia por la que objetos de distintas subclases pueden responder a una misma acción.

La polimorfia es implícita en Python, ya que todas las clases son subclases de una superclase común llamada **Object**.

Por ejemplo la siguiente función aplica una rebaja al precio de un producto, ubicar en **producto.py**:

```
def rebajar_producto(self,rebaja):
    self.precio = self.precio - rebaja
```

Gracias al polimorfismo no tenemos que comprobar si un objeto tiene o no el atributo *precio*,simplemente intentamos acceder y si existe premio:

Código

```
for producto in productos:
    producto.rebajar_producto(10)
    print(producto)
```

Por cierto, como podés ver en el ejemplo, cuando modificamos un atributo de un objeto dentro de una función éste cambia en la instancia. Esto es por aquello que comentamos del paso por valor y referencia.

Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Herencia múltiple

Finalmente hablemos de la herencia múltiple: la capacidad de una subclase de heredar de múltiples superclases.

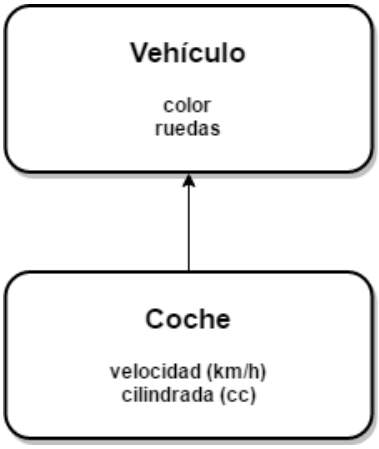
Esto conlleva un problema, y es que si varias superclases tienen los mismos atributos ométodos, la subclase sólo podrá heredar de una de ellas.

En estos casos Python dará prioridad a las clases **más a la izquierda** en el momento de ladeclaración de la subclase:

Código

```
(a.py)
class A:

    def a(self):
        print("Este método lo heredo de A")

    def b(self):
        print("Este método lo heredo de A")

(b.py)
class B:

    def b(self):
        print("Este método lo heredo de B")

(c.py)
from a import A
from b import B

class C(B,A):

    def __init__(self):
        print("Soy de la clase C")

    def c(self):
        print("Este método es de C")

(main.py)
from a import A
from b import B
from c import C

def main():
    c = C()
    c.a()
    c.b()
    c.c()

if __name__ == "__main__":
    main()
```

Hasta ahora sabemos que una clase heredada puede fácilmente extender algunas funcionalidades, simplemente añadiendo nuevos atributos y métodos, o sobrescribiendolos ya existentes. Como en el siguiente ejemplo:



Ejercicio

```
class Vehiculo():

    def_init_(self, color, ruedas):
        self.color  =  color
        self.ruedas = ruedas

    def __str__(self):
        return "Color {}, {} ruedas".format(self.color,self.ruedas)

class Coche(Vehiculo):

    def __init__(self, color, ruedas, velocidad, cilindrada):
        self.color = color
        self.ruedas = ruedas
        self.velocidad = velocidad
        self.cilindrada = cilindrada

    def __str__(self):
        return "color {}, {} km/h, {} ruedas, {} cc".format( self.color, self.velocidad,
    self.ruedas, self.cilindrada )

coche = Coche("azul", 150, 4, 1200)
print(coche)
```

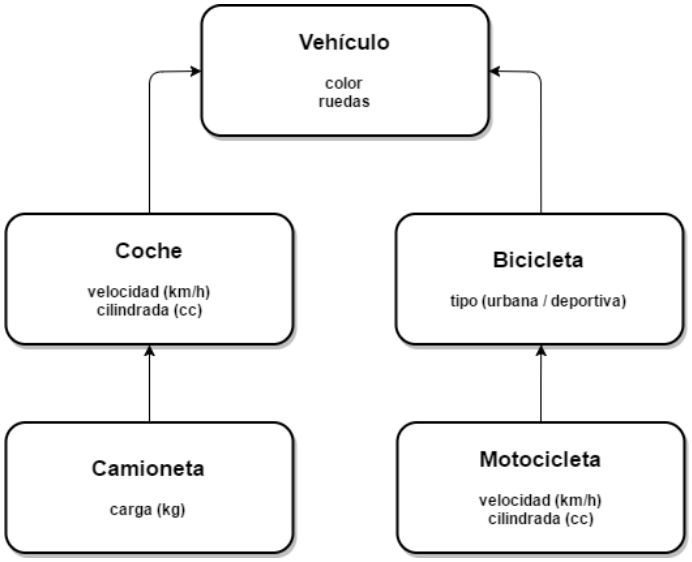
El inconveniente más evidente de ir sobrescribiendo es que tenemos que volver a escribirel código de la superclase y luego el específico de la subclase. Para evitarnos escribir código innecesario, podemos utilizar un truco que consiste en llamar el método de la superclase y luego simplemente escribir el código de la clase:

Ejercicio

```
(vehiculo.py)
class Vehiculo():

    def __init__(self, color, ruedas):
        self.color  =  color
        self.ruedas = ruedas

    def __str__(self):
        return "color {}, {} ruedas".format(self.color, self.ruedas)

(coche.py)
from vehiculo import Vehiculo
class Coche(Vehiculo):

    def __init__(self, color, ruedas, velocidad, cilindrada):
        Vehiculo.__init__(self, color, ruedas)
        self.velocidad = velocidad
        self.cilindrada = cilindrada

    def __str__(self):
        return Vehiculo._str_(self) + ", {} km/h, {} cc".format(self.velocidad,
self.cilindrada)

(main.py)
from coche import Coche

def main():
    c = Coche("azul", 4, 150, 1200)
    print(c)

if __name__ == "__main__":
    main()
```

Enunciado

Utilizando esta nueva técnica extiende la clase Vehiculo y realiza la siguiente implementación:



Crea al menos un objeto de cada subclase y agregalos a una lista llamadavehículos.

Realiza una función llamada **catalogar()** que reciba la lista de vehículos y los recorra mostrando el nombre de su clase y sus atributos.

Modifica la función **catalogar()** para que reciba un argumento optativo **ruedas**, haciendo que muestre únicamente los que su número de ruedas concuerde con el valor del argumento. También debe mostrar un mensaje **"Se han encontrado {} vehículos con {} ruedas:"** únicamente si se envía el argumento ruedas. Pon a prueba con 0, 2 y 4 ruedas como valor.

Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Clases Abstractas

Un concepto importante en programación orientada a objetos es el de las clases abstractas.Son clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instanciadas directamente. Solamente se pueden usar para construir subclases.Permitiendo así tener una única implementación de los métodos compartidos, evitando la duplicación de código.

Propiedades de las clases abstractas

La primera propiedad de las clases abstractas es que no pueden ser instanciadas. Simplemente proporcionan una interfaz para las subclases derivadas evitando así la duplicación de código.

Otra característica de estas clases es que no es necesario que tengan una implementaciónde todos los métodos necesarios. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración, pero no una implementación detalladade las funcionalidades.

Las clases derivadas de las clases abstractas debe implementar necesariamente todos losmétodos abstractos para poder crear una clase que se ajuste a la interfaz definida. En el caso de que no se defina alguno de los métodos no se podrá crear la clase.

Resumiendo, las clases abstractas definen una interfaz común para las subclases. Proporcionan atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código. Imponiendo además los métodos que deber ser implementados para evitar inconsistencias entre las subclases

Creación de clases abstractas en Python

Para poder crear clases abstractas en Python es necesario importar la clase ABC y el decorador **abstractmethod** del módulo **abc** (Abstract Base Classes). Un módulo que se encuentra en la **librería estándar** del lenguaje, por lo que no es necesario instalarlo. Así para definir una clase abstracta solamente se tiene que crear una clase heredada de **ABC**con un método abstracto.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass
```

Ahora si se intenta crear una instancia de la clase animal, Python no lo permitirá indicandoque no es posible. Es importante notar que, si la clase no hereda de ABC o contiene por lo menos un método abstracto, Python permitirá instancias de las clases.

```
class Animal(ABC):
    def mover(self):
        print("El animal se mueve")

animal = Animal()
```

Métodos en las subclases

Las subclases tienen que implementar todos los métodos abstractos, en el caso de que faltealguno de ellos Python no permitirá instancias tampoco la clase hija

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def comer(self):
        print('El animal come')
```

Por otro lado, desde los métodos de las subclases podemos llamar a las implementacionesde la clase abstracta con el comando super() seguido del nombre del método. La palabra **pass** permite no definir el contenido de un método.

```
(animal.py)
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def comer(self):
        print('Animal come')

(gato.py)
from animal import Animal

class Gato(Animal):

    def mover(self):
        print('Mover gato')

    def comer(self):
        super().comer()
        print('Gato come')

(main.py)
from gato import Gato

def main():
    g = Gato()
    g.mover()
    g.comer()

if __name__ == "__main__":
    main()
```

Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Diagrama de Clases

Es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relacionesentre los objetos.

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), se coloca uno de los siguientes signos delante de ese miembro:

+	Público
-	Privado
#	Protegido

Ámbitos

UML especifica dos tipos de ámbitos para los miembros: instancias y clasificadores y estosúltimos se representan con nombres subrayados.

Los miembros **clasificadores** se denotan comúnmente como “**estáticos**” en muchos lenguajes de programación. Su ámbito es la propia clase.

Los valores de los atributos son los mismos en todas las instancias

La invocación de métodos no afecta al estado de las instancias

Los miembros **instancias** tienen como ámbito una instancia específica.

Los valores de los atributos pueden variar entre instancias

La invocación de métodos puede afectar al estado de las instancias(es decir,cambiar el valor de sus atributos).

Para indicar que un miembro posee un ámbito de **clasificador**, hay que subrayar su nombre. De lo contrario, se asume por defecto que tendrá ámbito de **instancia**.

Fuente:

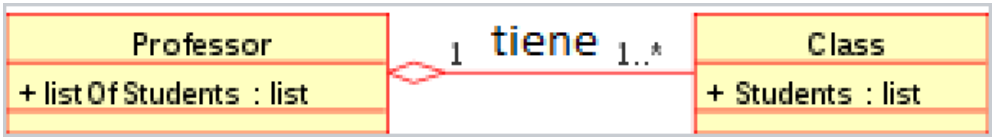
https://www.hektorprofe.net/

https://www.analyticslane.com/

Agregación y composición

Relaciones a nivel de instancia

Agregación



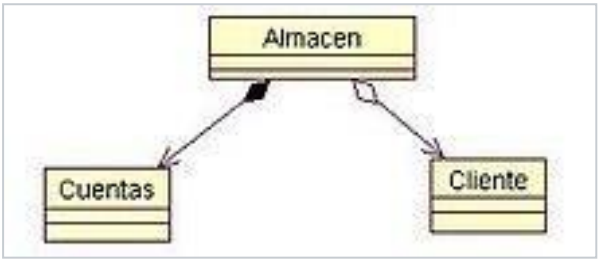
Como se puede ver en la imagen del ejemplo (en inglés), un Profesor **'tiene una o más clases'** clase a las que enseña.

Una agregación puede tener un nombre e indicaciones de cardinalidad en los extremos de lalínea. Sin embargo, una agregación no puede incluir más de dos clases; debe ser una asociación binaria.

Una ***agregación*** se puede dar cuando una clase es una colección o un contenedor de otras clases, pero a su vez, el tiempo de vida de las clases contenidas **no tienen una *dependencia fuerte del tiempo de vida*** de la clase contenedora. Es decir, el contenido de la clase contenedora **no se destruye** automáticamente cuando desaparece dicha clase.

Se representa gráficamente con un ***rombo hueco*** junto a la clase contenedora con una línea que lo conecta a la clase contenida. Todo este conjunto es, semánticamente, un objeto extendido que es tratado como una única unidad en muchas operaciones, aunque físicamenteestá hecho de varios objetos más pequeños.

Composición



El rombo negro muestra una **relación de composición**: el almacén está **compuesto** de cuentas, si se elimina el almacén las cuentas por si solas no tienen sentido como una entidad separada del almacén y **se eliminan también**. El rombo sin rellenar muestra unarelación de agregación: el almacén tiene clientes, si el almacén cierra los

clientes irán a otro, su razón de existir sigue teniendo sentido sin el almacén.

La representación de una relación de composición es mostrada con una figura de diamante rellenado del lado de la clase contenedora, es decir al final de la línea que conecta la clasecontenido con la clase contenedor.

Diferencias entre Composición y Agregación

Relación de Composición

1) Cuando intentamos representar un todo y sus partes. Ejemplo, un motor esuna parte de un coche.

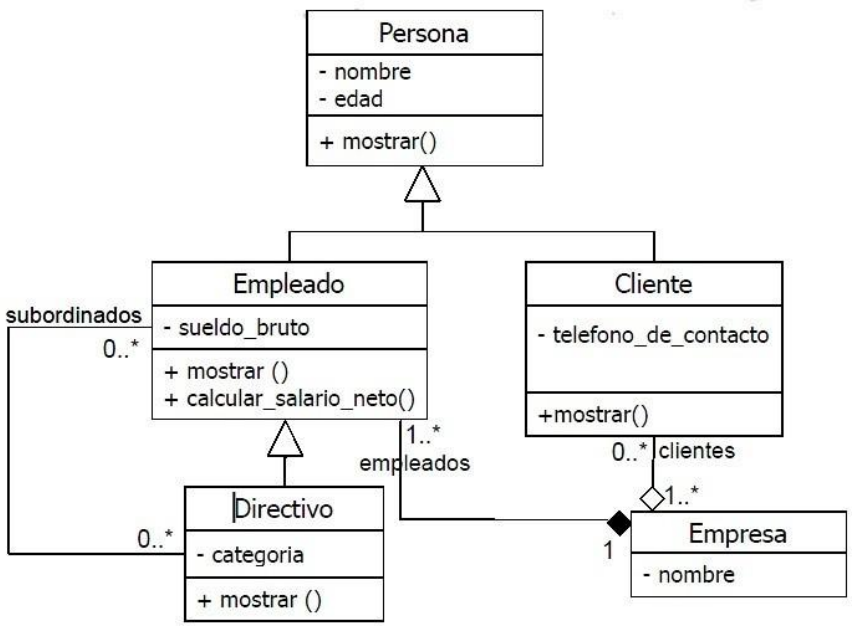
2) Cuando se elimina el contenedor, el contenido también es eliminado.Ejemplo, si eliminamos una universidad eliminamos igualmente sus departamentos.

Relación de Agregación

1) Cuando representamos las relaciones en un software o base de datos. Ejemplo, el modelo de motor MTR01 es parte del coche MC01. Como tal, el motor MTR01 puede ser parte de cualquier otro modelo de coche, es decir sieliminamos el coche MC01 no es necesario eliminar el motor pues podemosusarlo en otro modelo.

2) Cuando el contenedor es eliminado, el contenido usualmente no es destruido. Ejemplo, un profesor tiene estudiantes, cuando el profesor muere los estudiantes no mueren con él o ella.

Así, una relación de agregación es a menudo "clasificar" o "catalogar" contenido para distinguirlo del todo "físico" del contenedor.



Fuente:

https://www.hektorprofe.net/

https://www.analyticslane.com/

Errores

Los errores detienen la ejecución del programa y tienen varias causas. Para poderestudiarlos mejor vamos a provocar algunos intencionadamente.

Errores de sintaxis

```
print("Hola"
```

Resultado

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

Errores de nombre

Se producen cuando el sistema interpreta que debe ejecutar alguna función, método... perono lo encuentra definido. Devuelven el código **NameError**:

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

Resultado

```
<ipython-input-2-155163d628c2> in <module>()
----> 1 pint("Hola")

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

La mayoría de errores sintácticos y de nombre los identifican los editores de código antesde 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:

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

Resultado

```
<ipython-input-6-9e6f3717293a> in <module>()
----> 1 l.pop()

IndexError: pop from empty list
```

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

```
l = []

if len(l) > 0:
    l.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 tendremosun fallo **TypeError** que tampoco detectan los editores de código:

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

Resultado

```
Introduce un número: 4

---------------------------------------------------------------------------
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'
```

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

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

Resultado

```
Introduce un número: 10
10.0/4 = 2.5
```

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

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

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

Resultado

```
Introduce un número: aaa

---------------------------------------------------------------------------
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'
```

Fuente:
https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

Excepciones

Las excepciones son bloques de código que nos permiten continuar con la ejecución de un programa pese a que ocurra un error.
Siguiendo con el ejemplo de la lección anterior, teníamos el caso en que leíamos un númeropor teclado, pero el usuario no introduce un número:

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

Resultado

```
Introduce un número: aaa

---------------------------------------------------------------------------
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'
```

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("Introduce un número: "))
    m = 4
    print("{}/{} = {}".format(n,m,n/m))
except:
    print("Ha ocurrido un error, introduce bien el número")
```

Resultado

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
```

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.

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 yentonces romper el bucle con un break:

```
while(True):
    try:
        n = float(input("Introduce 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, introduce bien el número")
```

Resultado

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: sdsd
Ha ocurrido un error, introduce bien el número
Introduce un número: 10
10.0/4 = 2.5
```

Bloque else

Es posible encadenar un bloque else después del except para comprobar el caso en quetodo 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("Introduce un número: "))
        m = 4
        print("{}/{} = {}".format(n,m,n/m))
    except:
        print("Ha ocurrido un error, introduce bien el número")
    else:
        print("Todo ha funcionado correctamente")
```

```
        break  # Importante romper la iteración si todo ha salido bien
```

Resultado

```
Introduce un número: 10
10.0/4 = 2.5
Todo ha funcionado correctamente
```

Bloque finally

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

```python
while(True):
    try:
        n = float(input("Introduce un número: "))
        m = 4
        print("{}/{} = {}".format(n,m,n/m))
    except:
        print("Ha ocurrido un error, introduce bien el 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
```

Resultado

```
Introduce un número: aaa
Ha ocurrido un error, introduce bien el número
Fin de la iteración
Introduce un número: 10
10.0/4 = 2.5
Todo ha funcionado correctamente
Fin de la iteración
```

Fuente:

https://www.hektorprofe.net/

https://python-para-impacientes.blogspot.com/

Excepciones múltiples

En una misma pieza de código pueden ocurrir **muchos errores distintos** y quizá nosinterese 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("Introduce 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__)
```

Resultado

```
Introduce un número: 10
Ha ocurrido un error =>  TypeError
```

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

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

Resultado

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

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({}))
```

Resultado

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

Como vemos siempre nos indica eso de "**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__)
```

Resultado

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

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

```
try:
    n = float(input("Introduce 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__)
```

Resultado

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

Fuente:
https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

Invocación de excepciones

En algunas ocasiones quizá nos interesa llamar un error manualmente, ya que un printcomún 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()
```

Resultado

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

Instrucción raise

Gracias a **raise** podemos lanzar un error manual 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()
```

Resultado

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

Ejercicios resueltos

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

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

Resultado

```
Error:  No es posible dividir entre cero
```

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

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

Resultado

```
Error:  El índice se encuentra fuera del rango
```

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

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

Resultado

```
Error:  La clave del diccionario no se encuentra
```

Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

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

Resultado

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

Fuente:

https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

Módulos y Paquetes en Python

Módulos

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()
```

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**.

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 estándardesde el

intérprete o scripts sin cargarlos previamente.
Para crear un paquete lo que tenemos que hacer es crear un fichero especial **init** vacío enel 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:

```
paquete
|-__init__.py
|-saludos.py
|-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, asíque debemos hacerlo con from import:

```
from paquete.saludos import Saludo

s = Saludo()
```

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
|-hola
    |-__init__.py
    |-saludos.py
|-adios
   |-__init__.py
   |-despedidas.py
```

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")
```

Ahora de una forma bien sencilla podemos ejecutar las funciones y métodos de los módulosde cada subpaquete:

script.py

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

saludar()
Despedida()
```

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

paquetes/paquetes/

Fuente:
https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

Módulos estándar en Python

Algunos de los módulos esenciales de Python:

copy: Se utiliza para crear copias de variables referenciadas en memoria, comocolecciones y objetos.

collections: Cuenta con diferentes estructuras de datos.

datetime: Maneja tipos de datos referidos a las fechas/horas.

html, **xml** y **json**: También quiero comentar estos tres módulos, que aunque no los vamos a trabajar, permiten manejar cómodamente estructuras de datos html, xml y json. Son muy utilizados en el desarrollo web.

math: Posiblemente uno de los módulos más importantes de cualquier lenguaje, yaque incluye un montón de funciones para trabajar matemáticamente. Lo veremos más a fondo en esta misma unidad.

random: Este es el cuarto y último módulo que veremos en esta unidad, y sirve paragenerar 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 puedeser 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. La programación de hilos es compleja y he considerado que es demasiado para un curso básico-mediocomo éste.

tkinter: Tkinter esel módulo de interfaz gráfica de facto en Python.

Fuente:

https://www.hektorprofe.net/

https://python-para-impacientes.blogspot.com/

Módulo copy: copia de Objetos

Para realizar una copia a partir de sus valores podemos utilizar la función ***copy*** del módulo con el mismo nombre:

```
from copy import copy

class Test:
    pass

test1 = Test()
test2 = copy(test1)
```

Fuente:
https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

Módulo collections

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

Contadores

La clase **Counter** es una subclase de diccionario utilizada para realizar cuentas:

Ejemplo

```
from collections import Counter


lista = [1,2,3,4,1,2,3,1,2,1]
Counter(lista)
```

Resultado

```
Counter({1: 4, 2: 3, 3: 2, 4: 1})
```

Ejemplo

```
from collections import Counter

Counter("palabra")
```

Resultado

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

Ejemplo

```
from collections import Counter
animales = "gato perro canario perro canario perro"
c = Counter(animales.split())
print(c)

print(c.most_common(1)) # Primer elemento más repetido
print(c.most_common(2))  # Primeros dos elementos más repetidos
```

Resultado

```
Counter({'canario': 2, 'gato': 1, 'perro': 3})
[('perro', 3)]
[('perro', 3), ('canario', 2)]
```

Fuente:

https://www.hektorprofe.net/

https://python-para-impacientes.blogspot.com/

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)
print(dt.year)         # año
print(dt.month)        # mes
print(dt.day)          # día

print(dt.hour)         # hora
print(dt.minute)       # minutos
print(dt.second)       # segundos
print(dt.microsecond)  # microsegundos

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

Resultado

```
datetime.datetime(2016, 6, 18, 21, 29, 28, 607208)
2016
6
18
21
29
28
607208
21:29:28
18/6/2016
```

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**.

```
from datetime import datetime

dt = datetime(2000,1,1)
print(dt)
```

Resultado

```
datetime.datetime(2000, 1, 1, 0, 0)
```

No se puede cambiar un atributo al vuelo. Hay que utilizar el método **replace**:

```
dt = dt.replace(year=3000)

print(dt)
```

Resultado

```
datetime.datetime(3000, 1, 1, 0, 0)
```

Fuente:
https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

Módulo math

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

Redondeos

```
import math

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

Sumatoria mejorada

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

Resultado

```
6.9999999
```

Truncamiento

```
math.trunc(123.45)
```

Resultado

```
123
```

Potencias y raíces

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

Constantes

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

Fuente:

https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/

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 excluido
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))
```

Resultado

```
0.12539542779843138
6.272300429556777
7
14
68
25
```

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)
```

Resultado

```
o
3
[3, 4]
```

Mezclas

Y para mezclar colecciones:

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

Resultado

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

Fuente:
https://www.hektorprofe.net/
https://python-para-impacientes.blogspot.com/