Informática 2º Bto

Bloque : Programación

Programando en Python

IES Fco. Grande Covián
Informática II - Versión 2.0.1

1. Introducción

El curso anterior dimos nuestros primeros pasos dentro del mundo de la programación con Python. Aprendimos a:

- Utilizar y operar con los distintos tipos de datos.
- Crear y utilizar variables y listas.
- Mostrar información por pantalla utilizando la función print.
- Pasar valores al programa a través de la función input.
- Utilizar bloques condicionales if.
- Realizar procesos en bucle con las funciones for y while.

Este curso vamos continuaremos introduciendo nuevos conceptos que nos ayudarán a realizar nuevas y más complejas tareas.

Aprenderemos a trabajar con funciones y módulos, aprenderemos a manejar diccionarios y archivos externos.

Como bloque final del tema introduciremos un nuevo paradigma de programación, al que llamaremos Programación Orientada a Objetos, que constituye el modo de programación más utilizado en la actualidad.

Aprender a programar es una tarea que nunca termina. Espero que estos apuntes te sirvan para continuar aprendiendo y sobre todo que despierten en ti el interés por esta actividad de tal forma que te animes a seguir aprendiendo por tu cuenta.

2. Concepto de subrutina

A veces, en un determinado programa es necesario efectuar una misma tarea en distintos puntos del código. En otras ocasiones, diferentes programas necesitan efectuar una misma tarea.

Para evitar tener que volver a escribir una y otra vez las mismas instrucciones, los lenguajes de programación permiten agrupar porciones de código y reutilizarlas, ya sea en el mismo programa donde se han definido o en otros distintos. Estos bloques se llaman subrutinas.

> **Subrutina**: Bloque de instrucciones que puede ser llamado para su ejecución en un programa por medio de una orden sencilla.

Ventajas:

- **Ahorran trabajo**: Una vez creadas, no es necesario volver a programar ese bloque de código una y otra vez.
- **Facilitan el mantenimiento**: Si corregimos errores o implementamos mejores en una subrutina, esta modificación afectará a todos los puntos donde se aplique, sin tener que revisar todos los sitios en que utilicemos el algoritmo.
- **Simplifican el código**: La longitud y complejidad del programa se reducen porque las tareas repetitivas ya no aparecen en el cuerpo del programa.
- **Facilitan la creación de programas**: el trabajo se puede dividir entre varios programadores (unos escriben las subrutinas, otros el cuerpo principal del programa, etc.).

Las subrutinas se pueden definir y utilizar de dos formas diferentes:

- Subrutina a utilizar en un único programa: Se suele definir dentro del propio programa que la va a utilizar. En Python se suelen llamar **funciones**.
- Subrutina a utilizar en varios programas: Normalmente se incluirá la subrutina en un fichero aparte al que los programas que la necesiten pueden acceder. Estos ficheros reciben el nombre de **bibliotecas** (del inglés library) o **módulos**.

3. Funciones

> **Función**: Bloque de instrucciones que al ser llamado por el programa principal realiza la acción definida en su código.

3.1 Definición de una función

Se puede crear una función en cualquier punto del programa. Por claridad se suelen escribir al comienzo del programa.

> Otra razón para ello es que la función tiene que haber sido creada antes de poder ser utilizada. Crearlas al comienzo del código asegura que las podamos utilizar en cualquier punto del programa.

El procedimiento de creación de una función se llama **definición** y es el siguiente:

- **La primera línea** de la definición contendrá:
  - La palabra reservada def
  - El nombre de la función (la guía de estilo de Python recomienda escribir todos los caracteres en minúsculas separando las palabras por guiones bajos).
  - Paréntesis (pueden incluir los argumentos de la función, lo veremos más adelante)
  - Símbolo de dos puntos.
- **Bloque de instrucciones**:
  - Se escriben con sangría con respecto a la primera línea.
- **Finalización del bloque:**
  - El bloque de la función continuará mientras mantengamos la indentación.
  - Se puede finalizar la función con la palabra reservada return (la veremos más adelante), aunque no es obligatorio.

Veamos el siguiente ejemplo:

```
def licencia():
    print("Copyright 2016 IES Fco. Grande Covián")
    print("Licencia CC-BY-SA 4.0")
    return

print("Este programa no hace nada interesante.")
licencia()
print("Programa terminado.")
```

Generará la salida:

```
Este programa no hace nada interesante.
Copyright 2013 IES Fco. Grande Covián
Licencia CC-BY-SA 4.0
Programa terminado.
```

3.2 Utilizando variables en funciones

3.2.1 *Conflictos de nombres de variables*

Normalmente una función utilizará variables en su funcionamiento. Esto genera dos situaciones:

- Si la subrutina que utilizamos en un programa utiliza alguna variable auxiliar para algún cálculo intermedio y resulta que el programa principal utilizaba una variable con el mismo nombre, los cambios en la variable que se hagan en la subrutina podrían afectar al resto del programa de forma imprevista. Para evitar este problema nos interesaría que la variable de la función no tuviera sentido fuera de ella.
- En otros casos nos interesará que una subrutina pueda modificar variables que estén definidas en otros puntos del programa.

Para poder hacer frente a estas dos situaciones, los lenguajes de programación limitan lo que se llama el **ámbito de las variables**.

Python distingue tres tipos de variables:

- **Variables locales:** Pertenecen al ámbito de la subrutina.
- **Variables libres:** A su vez pueden ser:
  - **Variables globales**: Pertenecen al ámbito del programa principal.
  - **Variables no locales:** Pertenecen a un ámbito superior al de la subrutina, pero no son globales.

Si el programa contiene solamente funciones que no contienen a su vez funciones, todas las variables libres son variables globales.

Si el programa contiene una función que a su vez contiene una función, las variables libres de esas "subfunciones" pueden ser globales (si pertenecen al programa principal) o no locales (si pertenecen a la función).

Para identificar explícitamente las variables globales y no locales se utilizan las palabras reservadas global y nonlocal. Las variables locales no necesitan identificación.

A continuación, se detallan las reglas y situaciones posibles, acompañadas de ejemplos.

3.2.2 *Variables locales*

Si no se han declarado como globales o no locales, las variables **a las que se asigna valor** en una función se consideran variables **locales**, es decir, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre. Por ejemplo:

```
def subrutina():
    a = 2
    print(a)
    return

a = 5
subrutina()
print(a)
```

Mostrará por pantalla:

```
2
5
```

La función subrutina asigna el valor 2 a la variable a y muestra su valor por pantalla.

El código principal asigna a la variable a el valor 5, llama a la función subrutina(), que escribirá un 2 y por último escribe el valor de la variable a que al estar fuera de la función continúa teniendo el valor 5 ya que el valor 2 sólo se mantiene dentro de la función.

Las variables **locales** sólo existen en la propia función, ***no son accesibles desde niveles superiores***.

Por ejemplo:

```
def subrutina():
    a = 2
    print(a)
    return

subrutina()
print(a)
```

La orden print(a) generará un mensaje de error ya que intenta escribir el valor de una variable que no existe fuera de la subrutina.

```
2
Traceback (most recent call last):
File "D:\Code\8. Ejercicios while 2\src\ensayo.py", line 7, in <module>
print(a)
NameError: name 'a' is not defined
```

Si en el interior de una función **se asigna valor** a una variable que no se ha declarado como global o no local, esa variable es **local** a todos los efectos. Por ello el siguiente programa da error:

```
def subrutina():
    print(a)
    a = 2
    print(a)
    return

a = 5
subrutina()
print(a)
```

Genera:

```
Traceback (most recent call last):
File "D:\Code\8. Ejercicios while 2\src\ensayo.py", line 8, in <module>
subrutina()
File "D:\Code\8. Ejercicios while 2\src\ensayo.py", line 2, in subrutina
print(a)
UnboundLocalError: local variable 'a' referenced before assignment
```

La primera instrucción de la función produce un mensaje de error debido a que quiere imprimir el valor de la variable "a" definida como local dentro de la función (por el simple hecho de que en la línea siguiente se la va a asignar un valor), pero a la que todavía no se le ha dado valor dentro de la función (el valor de la variable "a" del programa principal no cuenta pues se trata de variables distintas, aunque se llamen igual).

3.2.3 *Variables libres globales o no locales*

Si a una variable **no se le asigna valor** en ninguna parte de una función, Python la considera **libre** y busca su valor en los niveles superiores de esa función, empezando por el inmediatamente superior y continuando hasta el programa principal. Si a la variable se le asigna valor en algún nivel intermedio la variable se considera **no local** y si se le asigna en el programa principal la variable se considera **global**.

En el siguiente ejemplo la variable libre "a" de la función subrutina() se considera global porque obtiene su valor del programa principal:

```
def subrutina():
    print(a)
    return

a = 5
subrutina()
print(a)
```

Produciendo el resultado:

```
5
5
```

En la función subrutina(), la variable "a" es **libre** puesto que no se le asigna valor. Su valor se busca en los niveles superiores, por orden. En este caso, el nivel inmediatamente superior es el programa principal. Como en él hay una variable que también se llama "a", Python coge de ella el valor (en este caso, 5) y lo imprime. Para la función subrutina(), la variable "a" es una **variable global**, porque su valor proviene del programa principal.

En el ejemplo siguiente, la variable libre "a" de la función sub_subrutina() se considera no local porque obtiene su valor de una función intermedia:

```
def subrutina():
    def sub_subrutina():
        print(a)
        return
    a = 3
    sub_subrutina()
    print(a)
    return

a = 4
subrutina()
print(a)
```

Mostrando:

```
3
3
4
```

En la función sub_subrutina(), la variable "a" es libre puesto que no se le asigna valor. Su valor se busca en los niveles superiores, por orden. En este caso, el nivel inmediatamente superior es la función subrutina(). Como en ella hay una variable local que también se llama "a", Python coge de ella el valor (en este caso, 3) y lo imprime. Para la función sub_subrutina(), la variable "a" es una variable no local, porque su valor proviene de una función intermedia.

Si a una variable que Python considera libre (porque no se le asigna valor en la función) tampoco se le asigna valor en niveles superiores, Python dará un mensaje de error.

3.2.4 *Variables declaradas global o nonlocal*

Si queremos asignar valor a una variable en una subrutina, pero no queremos que Python la considere local, debemos declararla en la función como global o nonlocal, como muestran los ejemplos siguientes:

En el ejemplo siguiente la variable se declara como global, para que su valor sea el del programa principal:

```
def subrutina():
    global a
    print(a)
    a = 1
    return

a = 5
subrutina()
print(a)
```

Mostrando el resultado:

```
5
1
```

La última instrucción del programa escribe el valor de la variable global "a", ahora 1 y no 5, puesto que la función ha modificado el valor de la variable y ese valor se mantiene fuera de la función.

En este ejemplo la variable se declara **nonlocal**, para que su valor sea el de la función intermedia:

```
def subrutina():
    def sub_subrutina():
        nonlocal a
        print(a)
        a = 1
        return

    a = 3
    sub_subrutina()
    print(a)
    return

a = 4
subrutina()
print(a)
```

El resultado será:

```
3
1
4
```

Al declarar nonlocal la variable "a", Python busca en los niveles superiores, por orden, una variable que también se llame "a", que en este caso se encuentra en la función subrutina(). Python toma el valor de esa variable, es decir, 3.

La última instrucción del programa escribe el valor de la variable global "a", que sigue siendo 4, puesto que ninguna función la ha modificado.

> Si a una variable declarada global o nonlocal en una función no se le asigna valor en el nivel superior correspondiente, Python dará un error de sintaxis.

4. Funciones: Argumentos y devolución de valores

Las funciones admiten argumentos en su llamada y permiten devolver valores como respuesta. Esto permite crear funciones más útiles y fácilmente reutilizables.

Veamos cuatro ejemplos:

4.1 Función con argumentos

Siguiendo el tipo de sintaxis explicado en el punto anterior podemos escribir:

```
def escribe_media():
    media = (a + b) / 2
    print("La media de", a, "y", b, "es:", media)
    return

a = 3
b = 5
escribe_media()
print("Programa terminado")
```

Mostraría el resultado:

```
La media de 3 y 5 es: 4.0
Programa terminado
```

El problema de una función de este tipo es que es muy difícil de reutilizar en otros casos, ya que sólo es capaz de hacer la media de las variables "a" y "b". Si en el programa no se utilizan esos nombres de variables, la función no funcionaría.

Para evitar ese problema, las funciones utilizan **parámetros** y **argumentos**.

> **Argumento:** Valor que se envía junto con la llamada a la función.
>
> **Parámetro:** nombre asignado a la variable que recibirá el valor de un argumento.

Así, las funciones se pueden reutilizar más fácilmente, como muestra el ejemplo siguiente:

```
def escribe_media(x, y):
    media = (x + y) / 2
    print("La media de", x, "y", y, "es:", media)
    return

a = 3
b = 5
escribe_media(a, b)
print("Programa terminado")
```

Esta función utiliza dos argumentos (x e y). Al llamar a la función será necesario colocar entre paréntesis los valores que se envían como agrumentos a los parámetros.

En este caso estamos utilizando dos parámetros y dos argumentos. Por defecto los valores se van asignando a los parámetros en el mismo orden que se escriben los argumentos en la llamada.

La función podrá ser utilizada con más flexibilidad que la del ejemplo anterior, puesto que el programador puede elegir qué valores enviar como argumentos.

Existen otras formas de asignar los valores los argumentos a los parámetros de la función.

4.1.1 Incluir ambos en la llamada de la función

Asociamos directamente asocias el nombre y el valor dentro del argumento. Esta forma de trabajar te libera de la preocupación de ordenar correctamente los argumentos en la llamada a la función, pero es necesario que se conozca el nombre de los parámetros.

Por ejemplo:

```
def describe_pet(animal_type, pet_name):
    print(f"\nTengo un {animal_type}.")
    print(f"El nombre de mi {animal_type} es {pet_name.title()}.")

describe_pet(animal_type='hámster', pet_name='harry')
```

La llamada a la función indica explícitamente el valor que a asignar a cada parámetro.

4.1.2 Valores predeterminados

Al crear la función, podemos definir un valor predeterminado para cada parámetro. En ese caso si cuando se llame a la función se proporciona un argumento para un parámetro, Python usa el valor del argumento. Si no, usa el valor predeterminado del parámetro.

Por ejemplo:

```
def describe_pet(pet_name, animal_type='dog'):
    print(f"\nTengo un {animal_type}.")
    print(f"El nombre de mi {animal_type} es {pet_name.title()}.")

describe_pet(pet_name='Willie')
```

Ahora, cuando se llama a la función sin especificar animal_type, Python sabe que debe usar el valor 'dog' para este parámetro:

```
Tengo un perro.
El nombre de mi perro es Willie.
```

4.2 Utilizando return. Devolviendo valores

El tipo de función anterior tiene una limitación. Como las variables locales de una función son inaccesibles desde los niveles superiores, el programa principal no puede utilizar la variable "media" calculada por la función y tiene que ser la función la que escriba el valor. Esto complica la reutilización de la función, porque aunque es probable que en otro programa nos interese una función que calcule la media, tal vez no nos interese que escriba el valor nada más calcularlo.

Para evitar ese problema, las funciones pueden devolver valores, es decir, enviarlos al programa o función de nivel superior. Por ejemplo:

```
def calcula_media(x, y):
    resultado = (x + y) / 2
    return resultado

a = 3
b = 5
media = calcula_media(a, b)
print("La media de", a, "y", b, "es:", resultado)
print("Programa terminado")
```

La orden “return resultado” hace que al ser invocada la función devuelva “resultado” como respuesta al código superior.

Esta función puede ser utilizada con más flexibilidad que la del ejemplo anterior, puesto que el programador puede elegir qué hacer con el valor calculado por la función.

4.3 Número variable de argumentos

La función definida en el apartado anterior tiene un inconveniente, sólo calcula la media de dos valores. Sería más útil que fuera capaz de calcular la media de cualquier cantidad de valores.

Podemos definir funciones que admitan una cantidad indeterminada de valores. Veamos:

```
def calcula_media(*args):
    total = 0
    for i in args:
        total += i
    resultado = total / len(args)
    return resultado

a, b, c = 3, 5, 10
media = calcula_media(a, b, c)
print("La media de", str(a)+",", b, "y", c, "es:", media)
print("Programa terminado")
```

La expresión *args indica que la función calcula_media admitirá una serie indefinida de argumentos y que trabajará con ellos como si fuera una lista.

> Ojo: args es el nombre de la lista y como tal podemos elegir el nombre que queramos.

Así esta función puede ser utilizada con más flexibilidad que las anteriores. El programador puede elegir de cuántos valores hacer la media y qué hacer con el valor calculado por la función.

4.4 Funciones que devuelven más de un valor

Las funciones pueden devolver varios valores simultáneamente, como muestra el siguiente ejemplo:

```
def calcula_media_desviacion(*args):
    total = 0
    for i in args:
        total += i
        media = total / len(args)
    total = 0
    for i in args:
        total += (i - media) ** 2
        desviacion = (total / len(args)) ** 0.5
    return media, desviacion

a, b, c, d = 3, 5, 10, 12
media, desviacion_tipica = calcula_media_desviacion(a, b, c, d)
print("Datos:", a, b, c, d)
print("Media:", media)
print("Desviación típica:", desviacion_tipica)
print("Programa terminado")
```

Al invocar la función esta genera dos repuestas que serán almacenadas en dos variables distintas según la orden:

```
media, desviacion_tipica = calcula_media_desviacion(a, b, c, d)
```

En Python no se producen conflictos entre los nombres de los parámetros y los nombres de las variables globales. Es decir, el nombre de un parámetro puede coincidir o no con el de una variable global, pero Python no los confunde: en el ámbito de la función el parámetro hará referencia al dato recibido y no a la variable global.

5. Módulos, bibliotecas

Todo lo que veamos sobre módulos en este apartado está enfocado a funciones. Cuando veamos programación orientada a objetos veremos que es aplicable a clases.

Hasta ahora hemos definido módulo como, cualquier archivo de texto que contiene ordenes en Python. Vamos a profundizar un poco en este concepto. Ahora que ya comprendemos mucho más de Python podemos realizar esta definición:

Módulo: Archivo con extensión .py o .pyc (Python compilado) que posee un espacio de nombres y que puede contener variables, funciones, clases y submódulos.

Trabajar con módulos nos va a permitir **modularizar** (organizar) y **reutilizar** el código.

Modularizar supone dividir el código total en bloques independientes. La aplicación completa se convierte en una especie de puzle que necesita de todas las pequeñas piezas independientes (módulos) para trabajar.

En ocasiones puede suceder que queramos **reutilizar** una función desde distintos programas. Una solución sería copiar el código de la función en cada uno de nuestros programas, pero esto no es práctico por muchas razones. Por ejemplo, si modificáramos más adelante la función sería necesario realizar esa modificación en todas las ocasiones que hubiéramos utilizado la función.

Python permite almacenar definiciones de funciones en módulos externos. Cuando queramos utilizar estas funciones sólo será necesario llamarlas desde el módulo principal del programa.

En el siguiente ejemplo creamos un módulo que contendrá una función que permite calcular un número definido de términos de la serie de Fibonacci.

Serie de Fibonacci: Serie numérica en la que el siguiente término se obtiene sumando el valor de los dos anteriores. Los dos primeros términos de la serie son el 0 y el 1.

En primer lugar, creamos un módulo que guardaremos con el nombre fibo.py

```
# módulo de números Fibonacci

def fib(n):    # escribe la serie Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print (b,end='')
        a, b = b, a+b

def fib2(n): # devuelve la serie Fibonacci hasta n
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

El módulo consta de dos funciones: fib(n) que al ser invocada mostrará por pantalla los términos de la serie y fib2(n) que almacenará los elementos de la serie en una lista, sin mostrarlos.

A continuación, crearemos el programa principal. Para poder utilizar en él las funciones anteriores será necesario importar el modulo fibo.py. Para ello utilizaremos la orden:

```
import fibo
```

Para acceder a las funciones utilizaremos la forma ***nombre_del_modulo.fun(arg)***. Donde:

- **nombre_del_modulo** es el nombre del módulo que hemos importado.
- **fun** es el nombre de la función incluida en el módulo que vamos a utilizar.
- **arg** valores asignados a los argumentos de la función invocada.

En el caso anterior:

```
import fibo
fibo.fib(1000)
print()
print(fibo.fib2(1000))
```

Daría como resultado:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

Si vamos a usar la función frecuentemente, podemos asignarle un nombre local:

```
fib = fibo.fib
fib(500)
```

Daría como resultado:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

5.1 Importando directamente las funciones

Aunque no es recomendable, ya que hace que el código sea menos legible, es posible utilizar la declaración import para importar directamente los nombres de las funciones incluidas en un módulo al espacio de nombres del módulo que hace la importación. Por ejemplo:

```
from fibo import fib, fib2
fib(500)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto no introduce en el espacio de nombres local el nombre del módulo desde el cual se está importando (en el ejemplo, fibo no se define).

Hay incluso una variante para importar todos los nombres que un módulo define:

```
from fibo import *
fib(500)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Esto importa todos los nombres excepto aquellos que comienzan con un subrayado (_). El problema de trabajar así es que si estamos utilizando un módulo con muchas funciones y no las vamos a utilizar todas estaremos haciendo un uso ineficiente de la memoria del equipo (poco relevante en los equipos actuales y con los programas sencillos que vamos a crear).

5.2 Más sobre los módulos

Un módulo puede contener tanto definiciones de funciones como declaraciones ejecutables. Estas declaraciones están pensadas para inicializar el módulo. Se ejecutan solamente la primera vez que el módulo se importa en algún lado.

Cada módulo tiene su propio espacio de nombres. Por lo tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse acerca de conflictos con una variable global del usuario.

Es costumbre, pero no obligatorio el ubicar todas las declaraciones import al principio del programa que las va a utilizar.

Los nombres de los módulos importados se ubican en el espacio de nombres global del módulo que hace la importación.

Los módulos pueden incluir clases. Pero el concepto de clase se corresponde con lo que llamaremos programación orientada a objetos y que veremos más adelante.

5.3 El camino de búsqueda de los módulos - Organización

Cuando un modulo intenta importar el contenido de otro módulo, Python lo busca en primer lugar en el mismo directorio en el que se encuentra el módulo principal.

Si allí no lo encuentra, lo buscará en la lista de directorios dada por la variable sys.path. Esta lista contiene el listado con los directorios definidos en PYTHONPATH.

En ambos casos no habrá ningún problema. Sin embargo, en muchas ocasiones nos interesará que un módulo sea accesible a todos los programas de nuestros proyectos.

Para ello será necesario trabajar con “paquetes”.

5.3.1 Uso de paquetes

Paquete: Directorio donde se almacenarán módulos relacionados entre sí.

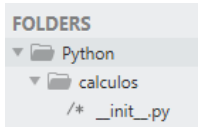
El uso de paquetes permite organizar y reutilizar los módulos de nuestra aplicación.

Para crear un paquete basta con crear una carpeta que contenga un archivo con el nombre __init__.py.

Vamos a desarrollar un ejemplo creando una aplicación que realice varios cálculos matemáticos. El módulo principal estará guardado dentro de la carpeta raíz Python y los módulos con las funciones en otra carpeta diferente.

Comenzaremos creando una carpeta que funcione a modo de paquete. La llamaremos cálculo y la guardaremos dentro del directorio raíz (Python).

Si queremos que la carpeta funcione como paquete debemos guardar en su interior un archivo (vacío) con el nombre __init__.py:



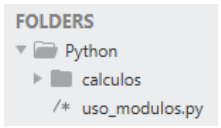
Crearemos ahora un módulo con el nombre calculos_generales.py que guardaremos dentro de la carpeta cálculos y con el código:

```
def sumar(op1, op2):
    print("El resultado de la suma es: ", op1 + op2)

def restar(op1, op2):
    print("El resultado de la resta es: ", op1 - op2)
```

```
def multiplicar(op1, op2):
   print("El resultado de la multiplicación es: ", op1 * op2)

def dividir(op1, op2):
   print("El resultado de la división es: ", op1 / op2)

def potencia(base, exponente):
   print("El resultado de la potencia es: ", base ** exponente)

def redondear(numero):
   print("El número redondeado es: ", round(numero))
```

Vamos a crear ahora un módulo dentro de la carpeta Python. Este módulo utilizará las funciones que están incluidas en el módulo cálculos_generales.py, que está guardado en la carpeta cálculos.

Para ello, creo este nuevo módulo al que llamaré uso_modulos.py:



Si queremos llamar a la función dividir del módulo cálculos_generales.py del paquete cálculos, tendremos que importarlo utilizando la sintaxis:

```
from calculos.calculos_generales import dividir
```

Ya podremos llamar a la función:

```
from calculos.calculos_generales import dividir
dividir(6, 3)
```

Que dará el resultado:

```
El resultado de la división es:  2.0
```

También habría sido posible realizar la importación del módulo completo:

```
import calculos.calculos_generales
```

Pero la llamada a la función de esta forma es muy farragosa:

```
calculos.calculos_generales.dividir(6, 3)
```

Si vamos al administrador de archivos de Windows:

Veremos que de forma automática se ha creado dentro del paquete una carpeta __pycache__ que contiene los elementos que Python necesita para utilizar estos módulos como paquetes.

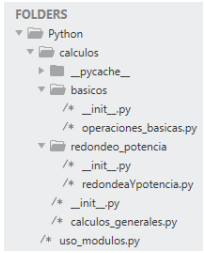
Subpaquetes

De cara a aumentar el nivel de organización, es posible crear subpaquetes dentro de un paquete.

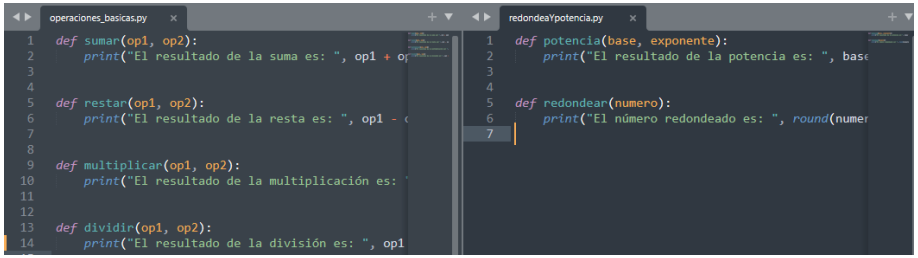
Por ejemplo dentro de nuestro paquete calculos podríamos crear varias carpetas agrupando el tipo de módulos en función de tipo cálculos que realicen sus funciones.

Crearemos una carpeta basicos y otra a la que llamaré redondeo_potencia. Cada una de ellas deberá llevar en su interior un archivo __init__.py.

Quedaría una estructura como:



Con los módulos:



Para poder llamar ahora a los módulos deberemos utilizar la sintaxis:

```
from calculos.basicos.operaciones_basicas import dividir
```

Es decir, se sigue la sintaxis del punto desde el paquete principal al modulo que contiene las funciones, pasando por todos los subpaquetes intermedios.

5.3.2 Paquetes distribuibles

El objetivo es crear un paquete que podamos utilizar en cualquier otro equipo o proyecto.

Los paquetes que hemos creado en el apartado anterior tienen su carpeta raíz en el mismo directorio que el módulo que los va a utilizar, pero eso no tiene porque se ser siempre así. En ocasiones sería más útil que Python los pudiera utilizar en cualquier proyecto, aunque estuviera en otro directorio raíz o equipo.

A modo de ejemplo continuaremos con el ejemplo del punto anterior. Vamos a copiar el módulo uso_modulos.py, que hasta ahora funcionaba de forma correcta por compartir directorio raíz con el paquete y lo voy a pegar en un directorio exterior al raíz (por ejemplo el escritorio).

Renombraremos el modulo como uso_modulos_fuera.py, para no confundirlo con el anterior.

Si ejecutamos el archivo uso_modulos.py funcionará de forma correcta:

```
El resultado de la división es:  2.0
```

Sin embargo, el archivo uso_modulos_fuera.py genera un error:

```
Traceback (most recent call last):
```

```
  File "D:\Users\joses\Desktop\uso_modulos_fuera.py", line 1, in <module>
    from calculos.basicos.operaciones_basicas import dividir
ModuleNotFoundError: No module named 'calculos'
```

Vamos a resolver este problema. Nuestro nuevo módulo, y cualquier otro, ha de ser capaz de utilizar las funciones definidas en el paquete.

Para ello en primer lugar crearemos un paquete distribuible y a continuación lo instalaremos en nuestro Python.

En realidad, lo que estamos haciendo es instalar el paquete dentro del propio Python.

Creación del paquete distribuible

Vamos a crear un paquete distribuible que incluya la función dividir. Al hacerlo, dicha función se podrá utilizar desde cualquier proyecto, ya que estará instalada en el propio Python.

Creamos un archivo con el nombre setup.py en la raíz de la carpeta que contenga a nuestro módulo (en el ejemplo: python).

Este archivo contendrá una descripción de paquete que vamos a distribuir. Incluiremos cosas como: el nombre del paquete, versión, nombre del autor, descripción…:

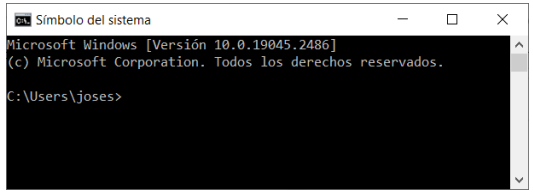
```
from setuptools import setup
setup(
   name="paquetecalculos",
   version="1.0",
   description="Paquete de división entre dos valores",
   author="IESFGC",
   author_email="josesallan@iesfgc.net",
   url="www.sallan.es",
   packages=["calculos", "calculos.basicos"]
)
```

Los campos autor_email y url son opcionales.

El campo packages es fundamental, indica cual es la dirección del paquete y módulo a empaquetar para distribuir).

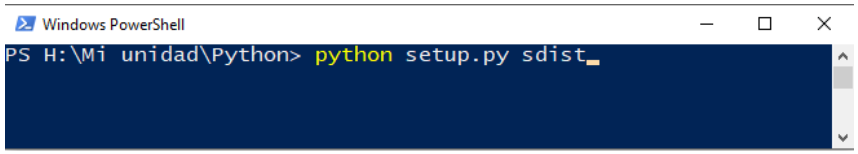
Guardamos el archivo setup.py.

Abrimos ahora una consola de Windows (botón Inicio, cmd) o el PoweShell que instala Windows (vamos con administrador de archivos al directorio que contiene el archivos setup.py y pulasmos shift+botón derecho del ratón)… esta última forma de trabajar ya nos abrirá la posición en la que está el archivo :

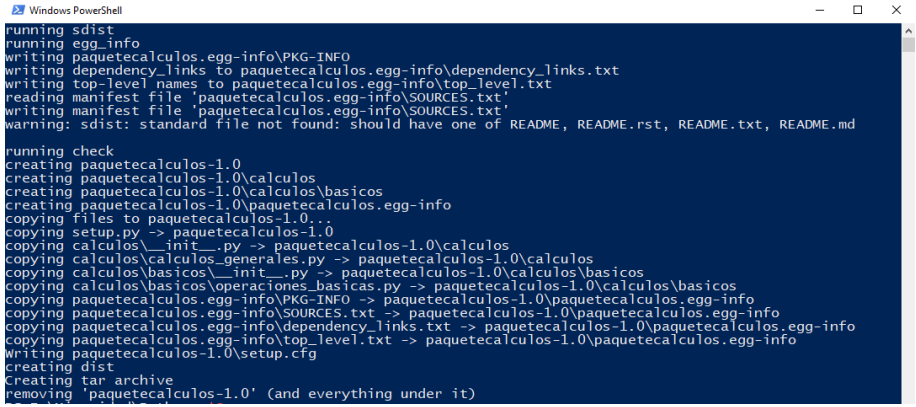


Nos movemos ahora hacia la carpeta que contiene el archivo setup.py

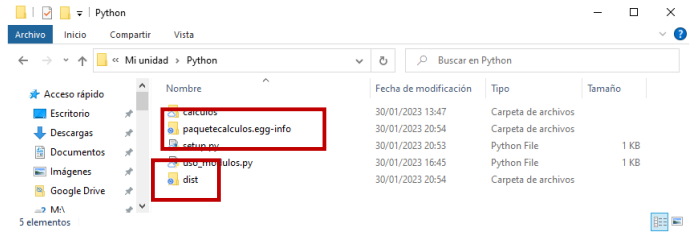
Escribimos Python setup.py sdist:



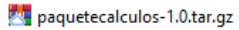
Se realiza la instalación:



En el directorio donde está el paquete que queremos distribuir se habrán creado dos carpetas:



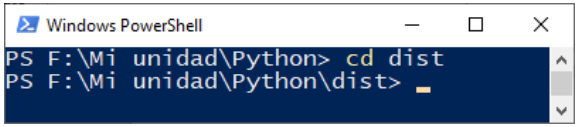
Nos interesa la carpeta dist (distribución), dentro tendremos un archivo comprimido con formato tar.gz.



Ese archivo es nuestro paquete distribuible, listo para instalar.

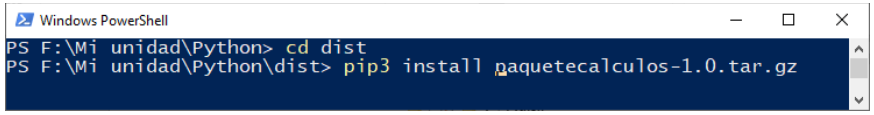
Instalación del paquete distribuible

Vamos de nuevo al powershell y entramos dentro de directorio dist:

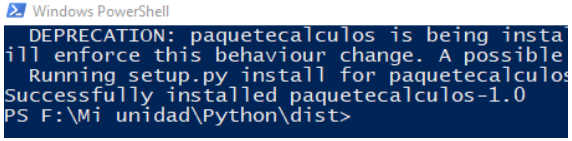


Introducimos la instrucción:

```
pip3 install nombreDelPaquete.tar.gz:
```



Enter y se inicia la instalación. Al terminar se mostrará un mensaje diciendo que la instalación ha sido exitosa:



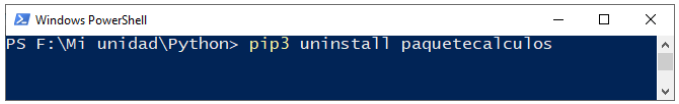
Si ejecutamos ahora el módulo uso_modulos_fuera.py, situado en el escritorio y veremos como el módulo funciona importando los paquetes necesarios:

Desinstalar un paquete instalado

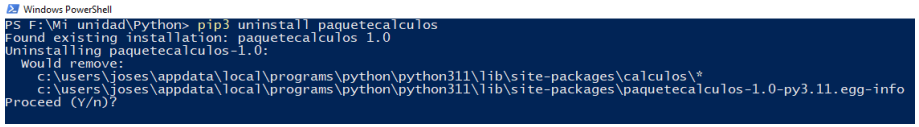
Abrimos una consola, igual da el directorio en que nos encontremos y tecleamos la instrucción:

```
pip3 uninstall nombreDelPaquete
```

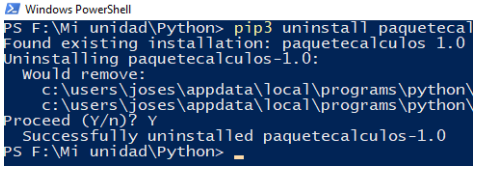
En nuestro caso:



El programa pide confirmación.



Tecleo Y, el programa informará que el paquete ha sido desinstalado:



6. Biblioteca estándar – Módulo random

Al realizar la instalación de Python se incluye la **llamada biblioteca estándar de Python**.

En esencia esta biblioteca es un enorme conjunto de módulos que podremos invocar en nuestros programas con la orden import de igual forma que añadíamos los módulos de nuestras bibliotecas personales en los puntos anteriores.

Al crear un modulo con Sublime Text también añadimos la referencia al directorio en que está guardados los módulos de la biblioteca estándar.

Puedes encontrar toda la información sobre la biblioteca estándar de python en el enlace:

https://docs.python.org/es/3/library/index.html

> A modo de ejemplo intenta buscar en módulo math.py la forma de utilizar el valor de π (es una constante en python) o como podríamos calcular el seno de un ángulo de 30º (ten en cuenta que la función seno en python utiliza radianes… tendrás que buscar una función que convierta un valor expresado en grados sexagesimales a radianes)

6.1 Módulo random.py

A modo de ejemplo vamos a trabajar con el módulo de la biblioteca estándar random.py

El módulo random permite generar valores aleatorios.

Existen una gran cantidad de métodos o funciones dentro de este módulo, destacamos:

Método	**Descripción**
random.randint(a, b)	Devuelve un número entero aleatorio entre a y b (ambos incluidos).-
random.choice(secuencia)	Devuelve de forma aleatoria un carácter de una variable o cadena de caracteres (secuencia).
random.shuffle(lista)	Reorganiza los elementos de una variable tipo lista. La lista mantiene su nombre de variable.
random.sample(secuencia, n)	Devuelve n elementos aleatorios de una variable o cadena de caracteres

Para poder utilizar los métodos incluidos en este módulo solamente es necesario utilizar la orden:

```
import random
```

A partir de ese instante si por ejemplo se quiere generar un número aleatorio entre 1 y 7 será suficiente con escribir:

```
random.randint(1,7)
```

Al igual que hacíamos con los módulos de nuestras bibliotecas personales podremos importar directamente las funciones incluidas en el módulo. El siguiente ejemplo importaría los procedimientos randint y simple del módulos random:

```
from random import randint, sample
```

A partir de ese momento para utilizar cualquiera de esos módulos es suficiente con nombrarlo directamente y no usar la sintaxis nombre_modulo.procedimiento. Según lo anterior:

```
randint(1,7)
```

6.2 Información sobre módulos de la biblioteca estándar

Puedes encontrar información sobre todos los componentes de esta biblioteca en:

https://docs.python.org/3/library/index.html

7. Excepciones y su gestión

Excepción: Error que ocurre durante la ejecución del programa. La sintaxis es correcta, pero durante la ejecución ocurre "algo inesperado".

Al llegar la excepción, Python genera un mensaje de error y la ejecución del programa se detiene.

Aprenderemos a gestionar las posibles excepciones. El programa será capaz de reconocer la excepción y generará una respuesta adecuada, permitiendo que el programa continúe.

7.1 Gestión del error división por cero

Veamos un ejemplo sencillo. Considera el siguiente código:

```
a=int(input("Introduce un número entero: "))
b=int(input("Introduce un número entero: "))
c=a/b
print(a,"/",b," = ", c)
print("Operación ejecutada")
```

El programa solicita dos números enteros. La tercera línea los divide y guarda el resultado en la variable, c. La cuarta línea muestra el resultado y la quinta un mensaje. Todo es correcto:

```
Introduce un número entero: 10
Introduce un número entero: 2
10 / 2  =  5.0
Operación ejecutada
```

Sin embargo, cuando asignamos el valor cero a la variable b (línea 2) obtenemos un error:

```
Introduce un número entero: 10
Introduce un número entero: 0
Traceback (most recent call last):
  File "..\src\print.py", line 3, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

La ejecución de programa termina al generarse el error (las órdenes print() no se ejecutan).

La solución es realizar una ***captura o control de excepción***. En esencia es la suma de dos acciones:

- Pedir al programa que intente ejecutar una instrucción o un bloque de instrucciones.
- En caso de que no pueda hacerse la acción dar una alternativa.

El procedimiento es el siguiente:

Observa la última parte de la pantalla del ejemplo anterior, recibe el nombre de **pila de llamadas**:

```
Traceback (most recent call last):
  File "..\src\print.py", line 3, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

Se muestra la información relativa a las últimas líneas de código que se han ejecutado hasta producirse el error. En este caso indica que la última línea que se ha intentado ejecutar es la

número 3 (c=a/b). También nos indica el tipo de error que se ha generado, en este caso ZeroDivisionError, siendo este un error de división por cero. Utilizaremos el nombre del tipo de error.

Para gestionar el posible error modificamos el código de la siguiente forma:

```
a=int(input("Introduce un número entero: "))
b=int(input("Introduce un número entero: "))
try:
   c=a/b
   print(a,"/",b," = ", c)
except ZeroDivisionError:
   print("No se puede dividir entre 0")
print("Operación ejecutada")
```

En negrita se indica como se gestiona el error. La estructura es parecida a un bloque if…else:

- El programa intenta realizar las ordenes incluidas dentro del bloque try.
- Cuando no se pueda ejecutar el bloque incluido en try y **solo si** el error producido es del tipo ZeroDivisionError se ejecutará el bloque de instrucciones incluidas en del apartado except.

Es decir:

- Si no hay error de división por cero:

```
Introduce un número entero: 8
Introduce un número entero: 2
8 / 2  =  4.0
Operación ejecutada
```

- Si hay error de división por cero:

```
Introduce un número entero: 8
Introduce un número entero: 0
No se puede dividir entre 0
Operación ejecutada
```

En ese segundo caso se ha conseguido dos cosas:

- Se ha gestionado el error producido al intentar dividir por cero.
- El programa continúa su ejecución tras la línea que habría generado el problema.

7.2 Gestión del error tipo de dato – While, break

Observa lo que ocurre al ejecutar el programa anterior cuando en lugar de un valor numérico introducimos en los input una cadena de texto:

```
Introduce un número entero: 8
Introduce un número entero: hola
Traceback (most recent call last):
  File "…\eclipse-workspace\Ensayos curos\src\print.py", line 2, in <module>
    b=int(input("Introduce un número entero: "))
ValueError: invalid literal for int() with base 10: 'hola'
```

El programa se interrumpe, la pila de llamadas informa de que el problema se ha producido en la segunda línea de código. En este caso el error es del tipo ValueError y se debe a que el programa no ha podido asignar un valor entero al dato que hemos introducido (hola) por ser este una cadena alfanumérica. La solución más sencilla sería, incluir las dos líneas input dentro de un bloque try:

```
try:
   a=int(input("Introduce un número entero: "))
   b=int(input("Introduce un número entero: "))
except ValueError:
   print("Los valores introducidos no son correctos"
```

El programa detectaría el primer problema, pero no va a funcionar de forma correcta:

```
Introduce un número entero: 7
Introduce un número entero: hola
Los valores introducidos no son correctos
Traceback (most recent call last):
  File "…\eclipse-workspace\Ensayos curos\src\print.py", line 7, in <module>
    c=a/b
NameError: name 'b' is not defined
```

En este caso evitamos el error del tipo de dato alfanumérico, pero se genera uno nuevo ya que no hemos asignado ningún valor a la variable b.

La solución será introducir las ordenes input en un bucle que se repita de forma infinita hasta que los datos hayan sido introducidos de forma correcta. Esto es:

```
while True:
   try:
      a=int(input("Introduce un número entero: "))
      b=int(input("Introduce un número entero: "))
      break
   except ValueError:
      print("Los valores introducidos no son correctos, inténtalo de nuevo:")
try:
   c=a/b
   print(a,"/",b," = ", c)
except ZeroDivisionError:
   print("No se puede dividir entre 0")
print("Operación ejecutada")
```

El resultado será:

```
Introduce un número entero: 5
Introduce un número entero: hola
Los valores introducidos no son correctos, inténtalo de nuevo:

Introduce un número entero: 5
Introduce un número entero: 2
5 / 2  =  2.5
Operación ejecutada
```

En el primer intento try detecta el error, se ejecuta la gestión de la excepción mostrando el texto “Los valores introducidos no son correctos”. Como estamos dentro del bucle while y es un bucle infinito (la concición es “True” y por lo tanto es siempre cierta) se vuelve a ejecutar su contenido.

En el segundo caso asignamos de forma correcta valor a las variables a y b, pasamos a la orden break y esta saca la ejecución del programa del interior del bucle continuando la ejecución en la siguiente orden try.

En un mismo bloque try: podemos incluir varias clausulas except consecutivas para capturar diferentes excepciones.

Otra forma de trabajar es colocar un único bloque except: sin indicar el tipo de error:

```
except:
    print(“menaje”)
```

Al producirse cualquier tipo de error (en el bloque try: asociado) se ejecutará la orden print(“mensaje”) y continuará la ejecución normal del programa.

- El lado positivo, es que sea cual sea el error la ejecución no se detendrá.
- El lado negativo, el programa no dará ningún mensaje específico asociado al tipo de error.

7.3 Clausula finally

Una sintaxis diferente a este bloque sería introducir la cláusula finally:

```
a=int(input("Introduce un número entero: "))
b=int(input("Introduce un número entero: "))
try:
    c=a/b
    print(a,"/",b," = ", c)
except ZeroDivisionError:
    print("No se puede dividir entre 0")
finally:
    print("Operación ejecutada")
```

El código incluido dentro de la sección finally, se ejecutará siempre una vez que se haya recorrido el conjunto try:-except:, independientemente de que se haya ejecutado el código try o el código de alguna de la excepciones.

Esta sintaxis parece equivalente a la anterior, más adelante veremos importantes diferencias.

7.4 Lanzar excepciones. Instrucción raise

Lanzar una excepción: El programador provoca una excepción cuando ocurre una circunstancia específica en el código.

La utilidad más importante de esta herramienta la veremos cuando estudiemos la unidad correspondiente a la programación orientada a objetos.

Lanzar una excepción

Veamos un ejemplo. Sea el siguiente código:

```
edad=int(input("Introduce tu edad: "))
if edad<20:
    print("Eres muy joven")
elif edad<40:
    print("Eres joven")
elif edad<65:
    print("Eres maduro")
elif edad<100:
    print("Cuídate...")
```

Si introducimos un valor negativo para la variable edad:

```
Introduce tu edad: -10
Eres muy joven
```

El programa no nos da un error, pero nos da una respuesta que no tiene sentido.

Podríamos resolver este problema de muchas formas, pero de cara a aprender cómo funciona esta herramienta, vamos a hacerlo lanzando una excepción diseñada por nosotros que se ejecutará cuando se cumpla una condición (en este caso edad<0).

La sintaxis sería:

```
edad=int(input("Introduce tu edad: "))
if edad<0:
    raise TypeError("No se permiten edades negativas")
if edad<20:
    print("Eres muy joven")
elif edad<40:
    print("Eres joven")
elif edad<65:
    print("Eres maduro")
elif edad<100:
    print("Cuídate...")
```

La parte que nos interesa está en negrita:

- Consideramos un if que detectará que el valor introducido no es correcto (edad<0).
- En ese caso se ejecuta la orden raise que lanzará una excepción. La excepción que invoquemos en realidad no es importante. Lo importante es que conseguimos parar el programa. En este caso uso TypeError…
- Entre paréntesis introducimos un mensaje personalizado que se ejecutará cuando lancemos la excepción y el **programa cae**.

Más adelante veremos cómo crear nuestras propias excepciones con un nombre específico.

Capturar una excepción

En el caso anterior el programa se detiene cuando generamos la excepción, veamos ahora como la podemos capturar y gestionar.

Veamos otro ejemplo. Vamos a crear un programa que calcule la raíz cuadrada de un número.

Como recordarás necesitaremos importar el módulo math y el método sqrt.

```
import math
def calculaRaiz(num):
    if num<0:
        raise ValueError("El número no puede ser negativo")
    else:
        return math.sqrt(num)
opt1=int(input("Introduce un número: "))
print(calculaRaiz(opt1))
print(“Programa finalizado”)
```

Si el número introducido es positivo, no habrá ningún problema:

```
Introduce un número: 256
16.0
```

```
Programa finalizado
```

Si el número es negativo:

```
Introduce un número: -256
Traceback (most recent call last):
  File "...\Ensayos curos\src\print.py", line 8, in <module>
    print(calculaRaiz(opt1))
  File "...\Ensayos curos\src\print.py", line 4, in calculaRaiz
    raise ValueError("El número no puede ser negativo")
ValueError: El número no puede ser negativo
```

Se lanza la excepción programada (no es posible calcular la raíz cuadrada de números negativos).

- El programa lanza el mensaje de error que hemos programado.
- Se detiene la ejecución y observa que la última línea de código ya no se ejecuta.

Para que el programa funcione de forma correcta debemos capturar la excepción y conseguir una respuesta adecuada:

La pila de llamadas nos informa (ya lo sabíamos) que el error se ha producido cuando la octava línea del código llama a la función, el problema está en la cuarta línea del código. La solución sería:

- Comprobar si se puede calcular la raíz.
- Si se puede mostrar el resultado.
- Si no se puede mostrar el mensaje de gestión del error y continuar con la ejecución.

Es decir:

```
import math
def calculaRaiz(num):
    if num<0:
        raise ValueError("El número no puede ser negativo")
    else:
        return math.sqrt(num)
opt1=int(input("Introduce un número: "))
try:
    print(calculaRaiz(opt1))
except ValueError:
    print("Error de Numero Negativo")
print("Programa finalizado")
```

Generará:

```
Introduce un número: -256
Error de Numero Negativo
Programa finalizado
```

8. Diccionarios

Diccionario: Lista no ordenada de términos. Cada uno de los términos queda definido por una **clave** única y almacena uno o varios **valores**.

Un ejemplo de diccionario podría ser un horario escolar. El diccionario tendría cinco términos. Cada uno de ellos quedaría identificado por una clave (el día de la semana) y cada uno de ellos almacenaría el nombre de las asignaturas de las que tengamos clase en ese día.

8.1 Crear un diccionario

Podemos crear un diccionario de dos formas:

Creamos el diccionario junto con los elementos que va a contener:

Se sigue la siguiente sintaxis

```
punto = {'x': 2, 'y':'texto', 'z': [4, 9]}
```

- Creamos un diccionario al que llamamos punto.
- Contiene tres términos cuyas claves son las cadenas de texto: 'x', 'y', 'z'.
- En este caso el valor guardado al termino cuya clave es 'x' es un número entero (2).
- El valor guardado en el término con clave 'y' es una cadena de texto 'texto'.
- El valor asociado al término con clave 'z' es una tupla [4,9] (lista no modificable).

Para referirnos a cada uno de los términos (por ejemplo, el que tiene por clave 'x') utilizaremos la forma punto['x'].

Así:

```
punto = {'x': 2, 'y': 1, 'z': 4}
print(punto)
print(punto['x'])
```

- El primer print mostrará el diccionario, incluyendo índices y valores.
- El segundo print muestra los valores asociados al índice x:

```
{'z': 4, 'y': 1, 'x': 2}
2
```

Definimos un diccionario vacío al que añadimos términos

Seguimos la sintaxis:

```
materias = {}
materias["martes"] = 6201
materias["miércoles"] = [6103, 7540]
materias["jueves"] = []
materias["viernes"] = 6201
print(materias)
print('Materias para el viernes', materias["viernes"])
```

Generando la respuesta:

```
{'martes': 6201, 'jueves': [], 'viernes': 6201, 'miércoles': [6103, 7540]}
Materias para el viernes 6201
```

Eliminar un elemento de un diccionario

Utilizaremos el método del:

```
>>> punto = {'x': 2, 'y': 1, 'z': 4}
>>> del punto['y']
>>> print(punto)
{'x': 2, 'z': 4}
```

Cada elemento se define con un par clave:valor, pudiendo ser la clave y el valor de cualquier tipo de dato ('int', 'float', 'chr', 'str', 'bool', 'object').

OJO: el diccionario no se muestra con el orden en que se han definido los términos.

8.2 Función get()

Observa el siguiente código:

```
1 materias = {}
2 materias["martes"] = 6201
3 materias["miércoles"] = [6103, 7540]
4 print(materias["viernes"])
```

La línea 4 llama aun termino cuya clave que no existe en el calendario, esto genera un error:

```
Traceback (most recent call last):
File "D:\Code\diccionarios\src\ensayo.py", line 4, in <module>
print(materias["viernes"])
KeyError: 'viernes'
```

Para evitar este error podemos utilizar la función **get**. Cuando utilizamos esta función con un diccionario:

- La función devuelve el valor asociado si el índice existe en el diccionario.
- La función devuelve el valor None cuando el índice no existe en el diccionario.

```
materias = {}
materias["martes"] = 6201
materias["miércoles"] = [6103, 7540]
print(materias.get("viernes"))
```

Genera:

```
None
```

En lugar de generar el valor None, get puede generar el valor que se desee incluyéndolo como segundo parámetro en la llamada de la función:

```
materias = {}
materias["martes"] = 6201
materias["miércoles"] = [6103, 7540]
print( materias.get("domingo", 'El índice referido no existe'))
```

Genera:

```
El índice referido no existe
```

Combinando get con un if podremos evitar invocar índices que no existan:

```
materias = {}
materias["martes"] = 6201
materias["miércoles"] = [6103, 7540]
if( materias.get("domingo")!=None):
    print(materias['domingo'])
```

El código anterior no mostrará nada por pantalla ya que al no existir el índice "domingo", la función get genera el valor None. La expresión lógica incluida en el if es falsa y por lo tanto la orden print no llega a invocarse.

Otra forma más elegante de conseguir no llamar índices no existentes es utilizar la palabra reservada in. Observa el siguiente ejemplo:

```
d = {'x': 12, 'y': 7}
if'y'in d:
    print(d['y'])  # Imprime 7
if'z'in d:
    print(d['z'])  # No se ejecuta
```

Genera la salida:

```
7
```

8.3 Acceder a todos los elementos de un diccionario

Hasta ahora para acceder a un término del diccionario es necesario conocer y usar su nombre.

Nos puede interesar mostrar de forma separada cada uno de los elementos sin tener que conocer los índices existentes.

Existen dos formas de conseguirlo:

Recorrer todas las claves y usar las claves para acceder a los valores:

En los ejemplos anteriores, añadiendo:

```
for dia in materias:
    print (dia, ":", materias[dia])
```

Produce:

```
martes : 6201
miércoles : [6103, 7540]
```

Obtener los valores como tuplas (listas no modificables):

Utilizamos el método .items() sobre el diccionario. Este método genera una lista en la que cada elemento es una tupla (lista inmutable) de dos elementos, el primero es el índice del término y el segundo su valor

En nuestro ejemplo genera:

```
[('martes', 6201), ('miércoles' ,[6103, 7540]), ('jueves',), ('viernes', 6201)]
```

Utilizamos un bucle for para leer los elementos de la lista de uno en uno. Utilizaremos dos variables de control para leer:

- La primera variable: El primer elemento de la tupla es el índice del término.
- La segunda variable: El segundo elemento de la tupla es valor asociado a ese índice.

```
for dia, codigos in materias.items():
```

```
    print (dia, ":", codigos)
```

Genera:

```
miércoles : [6103, 7540]
martes :6201
```

8.4 Funciones y métodos a utilizar con diccionarios

Existen una gran cantidad de funciones y métodos que podremos utilizar cuando trabajemos con diccionarios. Siendo name el nombre del diccionario, los más importantes son:

len(name)	Devuelve el número de elementos en el diccionario.
name.keys()	Devuelve una lista con los valores de las claves.
name.values()	Devuelve una lista con los valores del diccionario
name.setdefault(key,valor)	Si la clave key todavía no ha sido utilizada, añade un nuevo elemento al diccionario con esa clave y el valor 'valor'. Si ya existe no hace nada.
name.pop(key)	Elimina del diccionario el elemento con índice 'key'.
name.copy()	Crea una copia del diccionario name.
name.clear()	Elimina todos los elementos del diccionario.
name.update(Name2)	Añade los elementos del diccionario Name2 al Name1

8.5 Usos de diccionarios

Los diccionarios son herramientas muy útiles para crear bases de datos temporales en las que la clave es el identificador del elemento y el valor los datos asociados a dicho elemento. Ejemplos de sus aplicaciones serían:

- Contar cuantas veces aparece cada palabra en un texto.
- Contar cuantas veces aparece cada letra en un texto.
- Crear una agenda donde la clave sea el nombre de la persona y los valores los datos asociados a la misma.
- Mantener un listado de personas inscritas en una actividad siendo la clave su NIF y los valores los datos asociados a esa persona.
- Realizar traducciones. La clave sería la palabra en el idioma original y el valor la palabra en el idioma al que se quiere traducir (sistema muy malo).
- Crear un sistema de codificación de mensajes.

9. Módulo os.system() – Borrar pantalla de consola

La librería estándar de Python posee diversos módulos que nos facilitan muchas tareas. Por ejemplo, el módulo "os" hace posible el manejo de funciones dependientes del Sistema Operativo.

En primer lugar, hay que importar el módulo:

```
import os
```

Veamos un par de aplicaciones muy útiles.

Averiguar el tipo de sistema operativo utilizado

El método os.name nos indica el sistema operativo utilizado por el equipo. Así para conocer el sistema operativo en el que se está trabajando bastaría con:

```
import os
var = os.name
print ("El sistema operativo de mi ordenador es: "+var)
```

Los nombres registrados en este módulo son:

- 'posix' – UNIX
- 'mac' – Apple
- 'java' – Máquina virtual de Java
- 'nt', 'dos', 'ce' – Sistemas operativos desarrollados por Microsoft

- ***Borrado de la pantalla de consola***

Esta aplicación no realizará ninguna acción en la ventana de Eclipse, es solo válida para el trabajo con consola.

Una forma sencilla de abrir la consola es teclear cmd en el menú de inicio de Windows.

La línea de código que utilizaremos es:

- Para Unix/Linux/MacOS/BSD

```
os.system("clear")
```

- Para DOS/Windows

```
os.system("cls")
```

Si estamos creando un programa que deseamos que funcione en cualquier ordenador, el código debería primero identificar en que sistema operativo está corriendo. Para ello, crearemos la siguiente función:

```
def borrarPantalla():
    if os.name == "posix":
        os.system ("clear")
    elif os.name == "ce" or os.name == "nt" or os.name == "dos":
        os.system ("cls")
```

Posteriormente, para ejecutar la función solo debemos invocarla.

```
borrarPantalla()
```

Aplicación sencilla: Generar una pequeña animación

Vamos a mostrar un mensaje dinámico o simular movimiento.

Para entender a que nos referimos, veamos los siguientes ejemplos:

```
# Mensaje dinámico
import os
import time
if os.name == "posix":
  var = "clear"
elif os.name == "ce" or os.name == "nt" or os.name == "dos":
  var = "cls"
time.sleep(2)
os.system(var)
print("****************************************")
print("*                                      *")
print("*       ¡Bienvenidos a AprenderPython.net!      *")
print("*                                      *")
print("****************************************")
time.sleep(2)
os.system(var)
print("****************************************")
print("*                                      *")
print("*    Aquí aprenderas a programar con Python    *")
print("*                                      *")
print("****************************************")
time.sleep(2)
os.system(var)
print("****************************************")
print("*                                      *")
print("*                Es totalmente gratis                *")
print("*                                      *")
print("****************************************")
time.sleep(2)
```

10. Crear archivos ejecutables

Archivo ejecutable: Archivo que contiene todo lo necesario para poder ejecutarse de forma independiente al programa en el que ha sido diseñado.

La extensión característica de un archivo ejecutable dependerá de cual sea el sistema operativo para el cual se haya construido.

- En Windows la extensión será: .exe
- En Linux la extensión será: .tar .gz
- En Mac .dmg

Los pasos para crear un ejecutable a partir de un módulo .py son los siguientes:

Instalación de pyinstaller

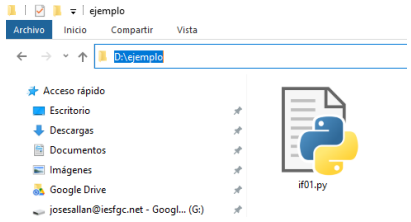
Vamos a utilizar la aplicación pyinstaller. En primer lugar, tenemos que instalarla en nuestro ordenador. Para ello abrimos una consola (Windows), tecleando cmd desde el botón de inicio de Windows y ejecutando la orden:

```
pip install pyinstaller
```

Ejecutamos pyinstaller

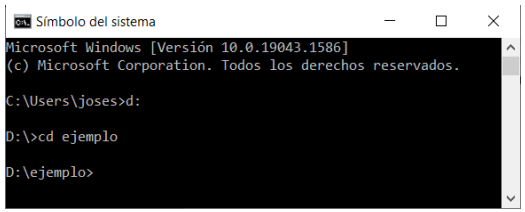
Debemos navegar en la consola hasta situarnos dentro del directorio en que está el módulo del cual queremos crear el ejecutable.

A modo de ejemplo vamos a copiar un archivo .py en una carpeta de nombre ejemplo en el disco duro d:

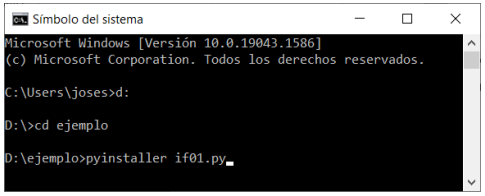


En este caso la dirección del directorio que contiene el módulo es: D:\ejemplo

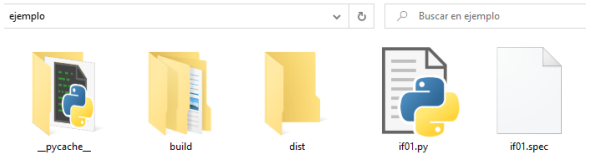
Vamos a la consola y nos movemos a esa dirección:



Ejecutamos ahora la orden pyinstaller seguida de un espacio y del nombre del módulo:



Se iniciará la ejecución de pyinstaller y en el directorio donde está el módulo aparecerán varios archivos y carpetas:



Dentro de la carpeta dist se habrá creado una carpeta con el nombre del módulo (if01 en el ejemplo). Dentro de ella encontraremos muchos archivos. Todos ellos son necesarios para hacer funcionar el ejecutable.

El archivo ejecutable es el que tiene extensión .exe (if01.exe) en el ejemplo. Si hacemos doble clic sobre su icono el programa arrancará.

Empaquetar todos los archivos en un único .exe

Para compilar todos los archivos anteriores en un único archivo utilizamos un modificador en la instrucción anterior:

```
pyinstaller --onefile nombre.py
```

Se creará un único archivo con el siguiente icono:

Al hacer doble clic se ejecutará la aplicación (tardará un poco más que antes ya que ahora es necesario leer todos los archivos que están contenidos dentro de nuestro .exe).

Sustituir el icono por uno personalizado

Antes de crear el ejecutable hay que guardar en el mismo directorio que hayamos ubicado el módulo .py de la imagen con el logo en formato .ico.

Añadiremos un nuevo modificador en el momento de crear el ejecutable:

```
pyinstaller --onefile --icon=./nombre.ico nombre.py
```

Aplicaciones gráficas sin consola

Si el módulo .py con el que trabajamos funciona con una interfaz gráfica (veremos algo de ello al final del curso), cuando lo ejecutemos, tras la interfaz gráfica se verá la consola de Windows, lo cual queda antiestético. Para que no se muestre la consola añade a la orden pyinstaller un modificador:

```
pyinstaller --windowed nombre.py
```

11. Manejo de archivos con Python

Archivo: Información almacenada en un dispositivo que puede ser utilizada por un programa informático.

Un archivo queda identificado por su:

- **Nombre:** Cadena de texto que utilizamos para referirnos a él.
- **Extensión:** Cadena de caracteres (habitualmente 3) colocada tras el nombre y separada de él por medio de un punto. Informa del tipo de información que contiene el archivo (por ejemplo, la extensión .jpg indica que la información debe ser interpretada como una fotografía comprimida en formato jpeg).
- **Path (Ruta):** Dirección que indica el "espacio lógico" en el que está guardado. Normalmente un dispositivo físico de almacenamiento como por ejemplo un disco duro.

No pueden existir en un mismo dispositivo dos archivos con igual nombre, extensión y path.

Al hablar de manejo de archivos nos referimos cualquier operación que permita crear, modificar o eliminar un archivo o la información almacenada en él.

Las operaciones básicas a realizar con archivos son cuatro:

Estas operaciones permiten conseguir algo que hasta hora no podíamos hacer, conseguir la persistencia de los datos. Leer datos que utilicen nuestros módulos y guardar datos que más adelante podremos recuperar.

11.1 Abrir en modo lectura y cerrar archivos

Para leer información de un archivo, Python utiliza la función open. Su sintaxis básica es:

```
fichero = open("file.txt", "r")
```

La función utiliza dos parámetros

- **file.txt:** Nombre y extensión del archivo que vamos a leer.
- **"r":** El valor "r" indica que queremos leer la información contenida en el archivo file.txt. Este parámetro es opcional, si no se escribe, por defecto open realiza la operación de lectura.

La función devuelve un "objeto archivo", un elemento que tendrá unos "atributos" característicos y al que le podremos aplicar unos "métodos" que veremos más adelante.

En el ejemplo *fichero* es el nombre con el que nos referiremos al objeto archivo en el código.

Tras terminar de trabajar con el archivo lo cerraremos con el método close() del objeto file.

```
fichero.close()
```

11.1.1 Sublime y posición relativa de los ficheros

Cuando utilizamos en Sublime Text una orden del tipo:

```
fichero = open("file.txt", "r")
```

Sublime buscará en el archivo file.txt en el "directorio de trabajo". En muchos programas (Eclipse) el directorio de trabajo será el mismo directorio en el que se ubica el modulo .py que estamos ejecutando. Por lo tanto en ellos si el archivo a abrir y el archivo que lo abre están en el mismo directorio, la orden anterior funcionará sin problemas.

Sin embargo en Sublime Text y otros IDEs el directorio de trabajo es la carpeta raíz de trabajo. Por ejemplo en la siguiente estructura:

Si estamos ejecutando el archivo 2.py, el directorio de trabajo, no será la carpeta "15. Ficheros", sino que será la carpeta "Informática II".

Puedes comprobarlo utilizando el siguiente código:

```
import os
# get current working directory
cwd = os.getcwd()
print("Directorio:", cwd)
```

El resultado será:

```
Directorio: /home/joses/Dropbox/Python/ejercicios python/Informática II
```

Esto supone que al ejecutar la orden open, el archivo 2.py no encontrará al archivo 1.txt, generando el error:

```
FileNotFoundError: [Errno 2] No such file or directory: '1.txt'
```

Este problema se puede resolver de tres formas:

1. Trabajar en Sublime desde la carpeta que contenga los archivos y utilizar la orden sencilla que hemos visto en el punto anterior.
2. Definir la dirección relativa completa del archivo (1.txt) respecto al directorio de trabajo. Si estudias el ejemplo lo entenderás:

   ```
   origen = open(r'15. Ficheros\1.txt')
   ```

   Desde el directorio de trabajo (Informática II) debemos entrar al directorio "15. Ficheros" y dentro de él (\) buscamos el fichero a abrir.

   El carácter r antes del la cadena de texto con la dirección del fichero transforma dicha cadena en una cadena cruda de texto y con ello evita que el carácter \ actúe como modificador del texto (visto cuando estudiamos las opciones de la función print).

3. **Obtener la dirección absoluta completa del módulo en el que se está trabajando**:

   ```
   import os
   ruta_modulo = os.path.abspath(__file__)
   ```

   A continuación obtenemos el nombre del directorio que contiene el archivo activo:

```
directorio_modulo = os.path.dirname(ruta_modulo)
```

En el caso del ejemplo:

```
import os
ruta_modulo = os.path.abspath(__file__)
print(ruta_modulo)
directorio_modulo = os.path.dirname(ruta_modulo)
print(directorio_modulo)
```

Generaría como respuesta:

```
/home/joses/Escritorio/Google Drive/Código/Python/Informática II/15. Ficheros/ficheros15.py
/home/joses/Escritorio/Google Drive/Código/Python/Informática II/15. Ficheros
```

11.2 Leer un archivo

Una vez que hemos abierto un archivo tenemos **dos formas de leer su información**:

A partir de ahora utilizaremos como fichero de prueba un fichero file.txt, que guardaremos en el mismo directorio nuestro programa principal. El contenido de file.txt debe ser:

```
All You Need Is Love - John Lennon/Paul McCartney

There's nothing you can do that can't be done.
Nothing you can sing that can't be sung.
Nothing you can say but you can learn how to play the game.
It's easy.

Nothing you can make that can't be made.
No one you can save that can't be saved.
Nothing you can do but you can learn how to be you in time.
It's easy.
```

11.2.1 Método read(). Leer un archivo como cadena de caracteres

Al aplicar el método read() sobre el objeto tipo archivo, el archivo se lee como una cadena de texto, que como ya sabes es a su vez una lista. En nuestro caso:

```
song = open("file.txt")
songLista=song.read()
print(songLista)
```

Al ejecutar el programa se mostrará el contenido completo del fichero.

En cambio, ya que el fichero lo estamos tratando como una cadena de texto:

```
song= open("file.txt")
songLista=song.read()
print(songLista[28:40])
```

Escribirá por pantalla los caracteres contenidos entre la posición 28 y 40 en el archivo file.txt.

La cadena leída contiene toda la información guardada en el archivo, incluyendo los retornos de carro y caracteres de formato.

.read(numero) leería "numero" caracteres dentro de la cadena desde la posición actual (relacionado con el punto siguiente seek y tell), lo veremos más adelante.

11.2.2 Método readlines(). Leer un fichero como lista

El método readlines() aplicado sobre un objeto archivo lee su contenido generando como resultado una lista en la cual cada línea del archivo se ha convertido en un elemento de la lista:

```
song= open("file.txt")
songLista=song.readlines()
print(songLista)
```

Genera:

```
['All You Need Is Love - John Lennon/Paul McCartney\n', '\n', "There's nothing you can do that can't be done.\n", "Nothing you can sing that can't be sung.\n", 'Nothing you can say but you can learn how to play the game.\n', "It's easy.\n", '\n', "Nothing you can make that can't be made.\n", "No one you can save that can't be saved.\n", 'Nothing you can do but you can learn how to be you in time.\n', "It's easy.\n"]
```

Podremos manejar la información exactamente igual que cualquier otra lista. Así si queremos imprimir la primera línea de la letra de la canción:

```
print(songLista[2])
```

Mostrará:

```
There's nothing you can do that can't be done.
```

Si queremos leer todas las líneas del archivo con su formato original, podemos utilizar un for:

```
fichero = open("file.txt", "r")
contenido = fichero.readlines()
for i in contenido:
    print(i)
fichero.close()
```

La primera línea abre el archivo file.txt y lo asocia a un objeto archivo que llamaremos fichero.

La segunda línea crea una lista con el nombre contenido. Cada elemento de esta lista será una de las líneas de fichero.

La variable i del bucle toma en cada iteración como valor el contenido de una nueva línea.

La última línea cierra el archivo.txt.

Si ejecutamos el programa obtendremos:

```
All You Need Is Love - John Lennon/Paul McCartney

There's nothing you can do that can't be done.
Nothing you can sing that can't be sung.
Nothing you can say but you can learn how to play the game.
It's easy.

Nothing you can make that can't be made.
No one you can save that can't be saved.
Nothing you can do but you can learn how to be you in time.
It's easy.
```

Eliminando retornos de carro y espacios en blanco: rstrip()

Observa que al terminar cada línea se añade el retorno de carro propio de un print() pero también otro más que se corresponde con el final de línea en el archivo original.

Podríamos evitar este problema utilizando el terminador de cadena end="" que hemos utilizado con en otras ocasiones con la función print. Sin embargo, vamos a aprender a hacerlo también a través de un método aplicado sobre el "objeto" "linea".

Método rstrip()

Aplicandos el **método rstrip()** sobre la línea que hemos leído (veremos más adelante que es un **objeto de la clase String**) eliminamos los espacios en blanco y retornos de carro que tengamos **al final de cada línea**:

```
fichero= open("file.txt")
for linea in fichero:
    print(linea.rstrip())
fichero.close()
```

El resultado ahora será:

```
All You Need Is Love - John Lennon/Paul McCartney

There's nothing you can do that can't be done.
Nothing you can sing that can't be sung.
Nothing you can say but you can learn how to play the game.
It's easy.

Nothing you can make that can't be made.
No one you can save that can't be saved.
Nothing you can do but you can learn how to be you in time.
It's easy.
```

El método .stript() funciona de forma similar, pero elimina todos los espacios y en blanco (y el retorno de carro) que tenga la cadena de texto **al comienzo y al final**.

11.3 Abrir en modo escritura y escribir en un archivo

Antes de escribir en un archivo es necesario abrirlo en modo escritura. Para ello utilizaremos la misma función open, pero como segundo parámetro utilizaremos la opción "w".

Una vez abierto utilizaremos el método **write()** incluido en el objeto archivo.

Ojo: Abriendo el archivo en modo w, creamos un archivo nuevo vacío si este no existía ya o en caso de existir perdemos toda la información que previamente estuviera guardada en él.

Veamos un caso sencillo:

```
fh= open("file.txt", "w")
fh.write("To write or not to write\nthat is the question!\n")
fh.close()
```

- En este caso abrimos el archive file.txt en modo escritura y lo asignamos a un objeto archivo al que llamamos fh. Al abrir el archivo en modo escritura se perderá la información que tuviéramos contenida en él.
- El método fh.write() escribe en el archivo de texto la cadena indicada como parámetro.
- Por último cerramos el archivo utilizando el método fh.close()

Leyendo de un archivo y escribiendo en otro

Un ejemplo un poco más complicado: Abriremos el archivo que contenía la letra de los Beatles, añadir al comienzo de cada línea una cifra entera que indique el número de línea dentro del mismo y guardarlo con un nuevo nombre en otro archivo diferente.

```
fichero_in = open("file.txt")
contenido = fichero_in.readlines()
fichero_out = open("file2.txt", "w")
i = 1
for line in contenido:
   print(line.rstrip())
   fichero_out.write(str(i) + ": " + line)
   i = i + 1
fichero_in.close()
fichero_out.close()
```

El resultado sería:

```
1: All You Need Is Love - John Lennon/Paul McCartney
2:
3: There's nothing you can do that can't be done.
4: Nothing you can sing that can't be sung.
5: Nothing you can say but you can learn how to play the game.
6: It's easy.
7:
8: Nothing you can make that can't be made.
9: No one you can save that can't be saved.
10: Nothing you can do but you can learn how to be you in time.
11: It's easy.
```

La clave está en la línea:

```
fichero_out.write(str(i) + ": " + line)
```

Esta línea añade en cada pasada una nueva línea al fichero file2.txt. El contenido de la línea es una cadena de texto formada por el número de línea y el contenido almacenado en la variable line (línea leída del fichero file.txt).

11.4 Añadir información a un archivo ya existente

Recuerda que al abrir un archivo con el método open("file_name","w") automáticamente borraremos su contenido. Si queremos abrir el archivo para añadir información sin borrarlo utilizaremos como segundo parámetro el valor "a" (append).

Ojo: Este modo de trabajo añade la nueva información al final del texto anterior (modo append), no es posible insertar texto en una posición intermedia.

11.5 seek y tell. Posición del puntero en el archivo

En ocasiones puede interesar acceder directamente a un punto definido del archivo en lugar de tener que leer toda la información del mismo. Para poder hacerlo definiremos un nuevo concepto:

Offset: Posición que ocupa el puntero dentro del fichero.

Entiende el puntero como un elemento que apunta a un carácter dentro del archivo utilizado.

Inicialmente al abrir el archivo offset es 0, es decir el puntero dirige al primer carácter del archivo.

Llevar al puntero a una posición en concreto del fichero

Para realizar esta operación utilizaremos el método ***seek***. Este método utiliza como parámetro un número entero que determina la posición que queremos adoptar como offset.

Para comprobar la posición que está ocupando el puntero tenemos, el método .***tell().***

Veamos un ejemplo. Utilizaremos de nuevo el archivo de prueba file.txt.

```
fh = open("file.txt")
print('Posición inicial',fh.tell())
print(fh.read(7))
fh.seek(10)
print('Posición actual',fh.tell())
print(fh.read(5))
print('Posición final',fh.tell())
print(fh.read(7))
```

El resultado es:

```
Posición inicial 0
All You
Posición actual 10
ed Is
Posición final 15
Love -
```

- El primer ft.tell() da como resultado un 0 por ser la posición inicial del puntero.
- **La orden fh.read(7) lee siete caracteres desde la posición actual del puntero (All You).**
- A continuación, llevamos el puntero a la posición 10 y mostramos el puntero de nuevo.
- La siguiente orden lee 5 caracteres, desde la posición actual (10): (ed Is).
- La posición final del puntero es 15.
- Si leemos 7 caracteres más, se leen desde la posición actual del puntero (15 – "Love – ".

También es posible modificar la posición del puntero de forma relativa a la posición actual. Así:

```
fh = open("file.txt")
print(fh.read(15))
fh.seek(fh.tell() -7)  # fijamos la posición actual siete caracteres a la izquierda:
print(fh.read(7))
fh.seek(fh.tell() + 25) # Avanzamos 25 caracteres a la derecha
print(fh.read(9))
```

Produce como resultado:

```
All You Need Is
Need Is
McCartney
```

11.6 Leer y escribir en el mismo archivo

En muchos casos es necesario abrir un archivo y tener que leer y escribir información al mismo tiempo. En este caso podemos utilizar dos modos de apertura diferentes:

Modo w+: Borra el contenido anterior, una vez abierto es posible leer y escribir en él.

Modo r+: No borra el contenido anterior, una vez abierto es posible leer y escribir en él. Si el archivo no existe se genera un error.

Veamos un ejemplo.

Si intentamos abrir un archivo y este no existe, la función open creará el archivo.

```
fh = open('colores.txt', 'w+')
fh.write('Color marrón')

# Vamos al sexto byte del fichero
fh.seek(6)
print(fh.read(6))
print(fh.tell())
fh.seek(6)
fh.write('azul')
fh.seek(0)
content = fh.read()
print(content)
```

Obtendremos el siguiente resultado:

```
marrón
12
Color azulón
```

Observa que una vez que definimos el punto en el que se quiere escribir, el nuevo texto sustituye al anterior (no se produce inserción de texto sino sustitución).

El ejemplo anterior se podría realizar a partir de un archivo inicial con la información "Color marrón", sería necesario realizar la apertura en modo r+ para evitar eliminar la información inicial.

11.7 Módulo pickle

En este punto aprenderemos a guardar y recuperar datos complejos en un fichero de una forma sencilla. Python ofrece para ello el módulo pickle (conservar en vinagre).

> El módulo pickle está incluido en la biblioteca estándar, por ello debe ser importado utilizando la orden **import pickle.**

Este módulo permite:

- **Pickle:** Transformar la información contenida en un objeto complejo (por ejemplo; una lista) en una cadena de bytes.
- **Unpickle:** Leer la cadena de caracteres anterior y recuperar la información original.

Pickle tiene dos métodos principales: dump y load:

11.7.1 Método dump

Permite escribir la cadena de bytes generada por pickle en un archivo.

> Antes de utilizar dump debemos haber abierto el archivo en modo escritura binaria, utilizando como segundo parámetro “wb” o “bw”.

El formato de este método es:

```
pickle.dump(objeto_a_exportar, nombre_archivo)
```

Veamos un ejemplo sencillo:

```
1 import pickle
2 ciudades = ["París", "Madrid", "Lisboa", "Munich"]
3 file = open("data.pkl", "bw")
4 pickle.dump(ciudades, file)
5 file.close()
```

Analicemos cada línea de código.

1. Importamos el módulo pickle de la biblioteca estándar de Python.
2. Creamos una lista con el nombre ciudades.
3. Abrimos (y en este caso creamos) un archivo con el nombre data.pkl y en modo de escritura de bytes. Asociamos el archivo data.pkl al objeto archivo file.
4. Utilizamos el método dump para escribir la lista ciudades en el archivo data.pkl.
5. Cerramos el archivo data.pkl.

El archivo data.pkl se habrá guardado en la misma carpeta donde esté guardado el programa principal, en nuestro caso D:\Code. Si abrimos el archivo con el bloc de notas obtendremos un documento aparentemente vacío. Esto se debe a que lo que hemos guardado es una cadena de bytes plana que el bloc de notas no puede interpretar.

> Normalmente utilizaremos la extensión .pkl para un archivo generado con pickle. Sin embargo, esto no es necesario. Podemos utilizar cualquier otra o incluso ninguna extensión. Utilizar .pkl informa al usuario de la función del archivo.

11.7.2 Método load

Permite leer una cadena de bytes generada previamente por pickle y recuperar el objeto a partir del cual se ha generado.

> Antes de utilizar load debemos abrir el archivo en modo lectura de bytes para lo cual es necesario que el segundo parámetro de open sea “br” o “rb”.

A modo de ejemplo vamos a crear el código necesario para leer el archivo data.pkl que creamos en el apartado anterior:

```
1 import pickle
2 file = open("data.pkl", "br")
3 villas=pickle.load(file)
4 print(villas)
5 file.close()
```

Analicemos cada línea de código.

1. Importamos el módulo pickle
2. Abrimos el archivo data.pkl en modo lectura de bytes y lo asignamos al objeto archivo con el nombre file.
3. Leemos el archivo y guardamos la información en una variable de nombre villas. Como el contenido almacenado en file es una lista, villas tomará el formato de variable lista.
4. Escribimos el contenido de la lista villas.
5. Cerramos el archivo.

> El objeto empaquetado podría ser todo lo complejo que yo quisiera. Por ejemplo, una lista de objetos, cada uno de los cuales fuera a su vez una lista (nombre jugador, número camiseta, posición en el campo, titular/suplente…)

11.8 Módulo shelve

El módulo pickle solamente es capaz de empaquetar un objeto en cada operación y debe desempaquetar toda la información cada vez.

Imagina que queremos utilizar pickle para crear una agenda telefónica. Cada vez que queramos obtener los datos de una persona habrá que leer toda la información. Ocurriría lo mismo si deseamos añadir un nuevo dato, tendríamos que escribir de nuevo toda la agenda.

Para casos de este tipo Python ofrece el módulo shelve. Este módulo crea objetos que funcionan como diccionarios persistentes (archivos que quedan guardados en nuestro equipo).

Los elementos de estos objetos estarán formados por una clave y un valor asociado como en un diccionario normal pero ***la clave ha de ser una cadena de texto***.

Veamos cómo funcionan:

- En primer lugar, debemos importar el módulo de la biblioteca estándar de Python.
- Crearemos un objeto shelve utilizando el método open. Este método abrirá un **archivo** tipo shelve (estantería) en modo lectura y escritura. Asignaremos a este objeto un nombre:

```
import shelve
s = shelve.open("MiEstanteria")
```

Si el archivo "MiEstanteria" ya existe (y es un archivo tipo shelve) el método lo abrirá, si no existe creará uno nuevo (en realidad crea tres ficheros de extensiones .bak, .dat y .dir).

A partir de ese momento podemos utilizar el archivo como si fuera un diccionario.

Por ejemplo, si utilizamos índices tipo cadena de carácter:

```
s["street"] = "Fleet Str"
s["city"] = "London"
for key in s:
    print(key)
```

Mostraría:

```
city
street
```

Como cualquier otro archivo, tras utilizarlo debemos cerrarlo:

```
s.close()
```

Recuerda que en un objeto shelve las claves de los valores han de ser cadenas de texto.

11.9 Resumen modo de apertura de archivos

open("file_name","r")	Abre un archivo en modo sólo lectura.
open("file_name","w")	Abre un archivo en modo sólo escritura borrando su contenido.
open("file_name","a")	Abre un archivo en modo sólo escritura y mantiene su contenido. Esta sólo se podrá añadir a continuación de los datos ya existentes.
open("file_name","w+")	Abre un archivo en modo lectura y escritura. Se borra la información anterior.
open("file_name","r+")	Abre un archivo, modo lectura y escritura. Mantiene la información. Si el archivo no existe, genera un error.
open("file_name","rb")	Modo lectura de bytes.
open("file_name","wb")	Modo escritura de bytes.

Nota: Intentar abrir un archivo que no existe, en modo lectura (r) genera un error; en cambio si lo abrimos para escritura se crea un archivo (w, a, w+ o con a+).

Si no se especifica el modo, los archivos son abiertos en modo sólo lectura (`r`).

12. Programación orientada a objetos

Iniciamos una nueva sección en el curso. La programación orientada a objetos (POO)

Python es un lenguaje de programación orientado a objetos. ¿Qué quiere decir eso?

Existen dos formas de entender la programación (paradigmas):

- Programación orientada a procedimientos.
- Programación orientada a objetos.

Programación orientada a procedimientos

Fueron los primeros lenguajes de programación de alto nivel. Surgieron en los años 60 y 70s. Hay muchos de ellos Fortran, Cobol, Basic.

En esencia consisten en una lista órdenes que se van ejecutando de forma secuencial de arriba abajo. La forma en la que hemos utilizado Python hasta ahora sería casi como si se tratara de un programa orientado a procedimientos.

Este tipo de lenguajes tienen muchas limitaciones:

- Generan muchísimas líneas de código. Esto los hace difíciles de entender, interpretar.
- Son poco reutilizables. Si queremos modificar una aplicación previa será complicado.
- Si hay un fallo en el código es muy fácil que caiga todo el programa.
- Habitualmente tienen órdenes tipo go to o go sub que hace que el flujo de ejecución de un programa vaya saltando adelante o atrás, dificultando su interpretación y modificación.

Programación Orientada a Objetos

Simplificaremos el proceso de programación creando elementos a los que llamaremos **objetos**. Un objeto en POO tiene la misma esencia que un "objeto" en el mundo real.

Un objeto de la vida real, por ejemplo, un coche, queda definido por tres tipos de elementos:

- **Propiedades** del coche: ¿Qué atributos tiene? Color, peso, tamaño…
- **Comportamientos** del coche: ¿Qué puede hacer? Puede frenar, arrancar, acelerar, girar…
- **Estados** del coche: ¿Está parado?,¿en movimiento?,¿aparcado?

Los objetos en POO tendrán a su vez propiedades y comportamientos y estados.

Estos lenguajes son más recientes, entre ellos tenemos Java, C++, Python.

Entre sus ventajas tenemos:

- Los programas se pueden dividir en "trozos", **modularización**.
- El código es muy **reutilizable**.
- Ante un error el resto del programa **continúa** con su funcionamiento.
- Permite algo que llamaremos "**Encapsulamiento**" (lo veremos más adelante).

La desventaja de estos lenguajes es que habrá que familiarizarse con un vocabulario (Clase, objeto, ejemplar de clase, modularización…) que inicialmente puede resultar complejo.

12.1 Conceptos previos

1. *Clase:*

Una clase es un **modelo** donde se definen **las características comunes** de un **grupo de objetos** del mismo tipo.

En el ejemplo del coche, una clase coche podría quedar definida por tener un mismo tipo de chasis y cuatro ruedas.

Una aplicación Python que trabajara con objetos coches tiene que partir de una clase que defina las características comunes de todos los coches que vayamos a construir.

2. *Ejemplar de clase = instancia de clase = objeto perteneciente a una clase:*

Es un objeto o ejemplar perteneciente a nuestra clase.

En nuestro ejemplo un objeto sería un coche en concreto. Comparten una serie de características comunes dadas por la clase a la que pertenecen, tienen un chasis y cuatro ruedas, pero cada uno de ellos tendrá sus propias características (ej: color, matrícula, peso…).

3. *Modularización*

Modularizar es dividir el programa completo en partes (clases) independientes.

Normalmente, una aplicación compleja estará formada por varias clases. Al igual que un antiguo equipo HIFI estaba formado por varios módulos (radio, cd, amplificador)

Cada uno de los módulos funciona de forma independiente. Esto supone varias ventajas: Permite reutilizar componentes, trozos de código y aunque un módulo falle el resto sigue funcionando.

4. *Encapsulación*

El funcionamiento de cada uno de los módulos ha de ser independiente del resto del programa.

Todas las clases del programa están conectadas entre sí para que puedan funcionar como equipo, pero el funcionamiento interno de cada una de las clases será independiente del resto.

Las diferentes **clases quedan conectadas** por los **métodos de acceso**.

5. *Nomenclatura del punto*

Nomenclatura que se utiliza para poder acceder a las propiedades y comportamientos de un objeto. Común a la mayoría de lenguajes de POO.

Consiste en:

- **Cada objeto** queda definido por **un nombre** (miChoche)
- **Para acceder a sus propiedades**, escribimos el nombre del objeto seguido de un punto y del nombre de la propiedad:

  miCoche.color="rojo" o miCoche.peso=1500
- **Para acceder a los comportamientos** del objeto, escribimos el nombre del objeto seguido de un punto y el nombre del comportamiento seguido de unos paréntesis.

  miCoche.arranca()

13. Pasando a código lo visto hasta ahora

13.1 Crear una clase

Sintaxis:

- Palabra reservada class.
- Nombre de la clase, primera letra mayúscula. Seguido de dos paréntesis y dos puntos.
- En el interior (sangría) código de la clase.

A modo de ejemplo vamos a definir la clase Coche. De momento tendríamos:

```
class Coche():
```

Por convenio el nombre de una clase comienza siempre por mayúscula.

Recuerda que un objeto quedará definido por sus propiedades, comportamientos y estados. Vamos a ir añadiendo los mismos:

13.1.1Añadir propiedades y estados:

Solo es necesario definir el nombre de la propiedad y su valor, por ejemplo:

```
class Coche():
    largoChasis = 250
    anchoChasis = 120
    ruedas = 4
    enMarcha = False
```

En realidad, **enMarcha sería un estado y no una propiedad**, por ser un valor que podremos cambiar. Es decir enMarcha (estado) podrá ser True en algún momento, sin embargo el número de ruedas (propiedad) no cambiará.

Todos los objetos que, más adelante, creemos pertenecientes a la clase Coche tendrán en común un chasis que medirá 250 cm de largo, 120 cm de ancho, cuatro ruedas e inicialmente no estarán en marcha.

13.1.2Añadir comportamientos, procedimientos o métodos:

Un **método** es un bloque de código, una "función", perteneciente a una clase, que permite hacer una o varias acciones.

Sintaxis

- Palabra reservada def (igual que haríamos con una función).
- Nombre del método, paréntesis y en su interior escribimos como parámetro la palabra self. Terminamos la línea con dos puntos.
- Bloque de código que define que va a hacer el método (indentado).

self indica como parámetro que el método se refiere al propio objeto que creemos en un futuro (es decir a sí mismo).

En nuestro ejemplo vamos a crear un método que permita arrancar a los coches que creemos más adelante. Este método permitirá cambiar el valor del estado enMarcha en los objetos que creemos en un futuro de False a True:

```
class Coche():
    largoChasis = 250
    anchoChasis = 120
    ruedas = 4
    enMarcha = False
    def arrancar(self):
        self.enMarcha = True
```

13.2 Creando objetos

Vamos a crear dos objetos pertenecientes a la clase Coche.

En primer lugar, salimos de la definición de la clase Coche eliminando la indentación. Para crear nuestro primer objeto:

Sintaxis

- Asignamos un nombre, siguiendo las normas habituales para nombres de variables.
- Símbolo igual.
- Nombre de la clase a la que pertenece el objeto.

```
miCoche = Coche()
```

La línea anterior crearía un objeto llamado miCoche perteneciente a la clase Coche()

Para ver las propiedades del objeto o cambiar su comportamiento utilizaremos la nomenclatura del punto que hemos visto en el apartado anterior.

Por ejemplo, ***para ver la propiedad*** longitud de su chasis:

```
print(miCoche.largoChasis)
```

O si queremos una presentación más rica:

```
print("La longitud de mi coche es:", miCoche.largoChasis)
```

Para cambiar la propiedad enMarcha de False a True debemos ***ejecutar el comportamiento*** arrancar:

```
miCoche.arrancar()
```

Al leer esta orden, Python llama al método arrancar. El método recibe como parámeto (self) es decir el nombre del propio objeto que ha llamado al método en este caso miCoche. El método asigna a la propiedad enMarcha el valor True.

A modo de práctica intenta añadir un procedimiento que nos diga si nuestro coche está en marcha o está parado:

```
class Coche():
    largoChasis = 250
```

```
    anchoChasis = 120
    ruedas = 4
    enMarcha = False
    def arrancar(self):
        self.enMarcha = True
    def estado(self):
        if (self.enMarcha == True):
            return "El coche está en marcha"
        else:
            return "El coche está parado"

miCoche = Coche()
print("La longitud de mi coche es:", miCoche.largoChasis)
# Llamo al método que arranca mi coche
miCoche.arrancar()
# Muestro por pantalla el estado del coche a través de un método
print(miCoche.estado())
```

13.2.1 Creando un segundo objeto de la misma clase

Creo un nuevo objeto asignándole un nombre distinto al anterior.

```
miCoche2 = Coche()
```

Este nuevo objeto tendrá las mismas propiedades y comportamientos de clase que el objeto anterior. Modificando las propiedades o ejecutando los procedimientos puedo conseguir que los diferentes objetos tengan diferentes estados:

```
class Coche():
    largoChasis = 250
    anchoChasis = 120
    ruedas = 4
    enMarcha = False
    def arrancar(self):
        self.enMarcha = True
    def estado(self):
        if (self.enMarcha == True):
            return "El coche está en marcha"
        else:
            return "El coche está parado"

miCoche = Coche()
print("La longitud de mi coche es:", miCoche.largoChasis)
# Llamo al método que arranca mi coche
miCoche.arrancar()
# Muestro por pantalla el estado del coche a través de un método
print(miCoche.estado())

print("----Creamos un nuevo objeto: miCoche2----")
miCoche2 = Coche()
print("La longitud de mi coche2 es:", miCoche2.largoChasis)
print(miCoche2.estado())
```

En este caso:

```
La longitud de mi coche es: 250
El coche está en marcha
----Creamos un nuevo objeto: miCoche2----
La longitud de mi coche2 es: 250
El coche está parado
```

Estos dos objetos de la clase Coche, tienen unas características comunes definidas por su clase, pero el estado es diferente ya que en el primer caso se ha ejecutado el método arrancar y en el segundo no.

Añade al código un procedimiento que informe sobre las propiedades de los objetos de la clase Coche().

13.3 Métodos que reciben parámetros

Al igual que ocurría con las funciones, los métodos pueden recibir parámetros.

Supón un objeto (coche1) en el cual queremos ejecutar un método (arrancar) que funcione con un parámetro (por ejemplo "arrancamos").

La forma de invocar el método sería:

```
coche1.arrancar(arrancamos)
```

Este método habrá tenido que ser definido antes. La cláusula que define el nombre del método tendrá que contener el nombre del parámetro y el nombre del parámetro por defecto self, que hace referencia al propio objeto que ejecuta el método.

```
def arrancar(self, arrancamos)
```

A modo de ejemplo vamos a modificar el código anterior de tal forma que el método arrancar no se encargue solamente de arrancar el coche, si no que informe también del estado. Por otro lado, cambiaremos la función del método estado para que nos informe del número de ruedas, ancho y largo de cada uno de los dos objetos del ejemplo anterior.

La idea es llamar al método arrancar, pero pasándole un parámetro que indique si estamos arrancando o no.

```
class Coche():
    largoChasis=250
    anchoChasis=120
    ruedas=4
    enMarcha=False
    def arrancar(self,arrancamos):
        self.enMarcha=arrancamos
        if (self.enMarcha==True):
            return "El coche está en marcha"
        else:
            return "El coche está parado"
    def estado(self):
        print("El coche tiene ", self.ruedas, "ruedas.  Un ancho de ", self.anchoChasis,
" y un largo de ", self.largoChasis)

miCoche=Coche()
print(miCoche.arrancar(True))
miCoche.estado()

print("----Creamos un nuevo objeto: miCoche2---")
miCoche2=Coche()
print(miCoche2.arrancar(False))
miCoche2.estado()
```

14. Constructor, definir el estado inicial

Todos los objetos que vayamos creando van a tener unas propiedades iniciales que quedan definidas en la definición de clase. En nuestro ejemplo estas propiedades son:

```
largoChasis=250
anchoChasis=120
ruedas=4
enMarcha=False
```

En POO es habitual definir ese estado inicial mediante un método especial al que llamaremos **constructor.**

Constructor: Método especial que define el estado inicial de los objetos pertenecientes a una clase.

Es una forma de especificar claramente el estado inicial en el código. Puede parecer superfluo, pero ten en cuenta que en un programa real en número de propiedades puede ser muy grande, incrementando la complejidad del código.

La **sintaxis** en Python está formada por:

- Palabra reservada def
- Espacio en blanco
- Dos guiones bajos
- Palabra reservada init
- Dos guines bajos
- Parámetro self entre paréntesis
- Dos puntos

```
def __init__(self):
```

Para hacer ahora la definición de las propiedades utilizaremos la nomenclatura del punto referida al parámetro self. Es decir, en nuestro ejemplo:

```
class Coche():
    def __init__(self):
        self.largoChasis=250
        self.anchoChasis=120
        self.ruedas=4
        self.enMarcha=False
```

14.1 Crear objetos con un constructor con parámetros

Si estudias el código del ejemplo anterior, observarás una limitación. Todos los objetos van a tener las mismas propiedades iniciales. Imagina que quieres crear objetos de la clase Coche, pero quieres tener la posibilidad de crear objetos coche de diferentes colores.

La solución sería pasar color al constructor como un parámetro cuando creamos el objeto.

La orden de creación del objeto tomaría la forma:

```
miChoche=Coche(rojo)
```

Para que esta orden tenga sentido, será necesario haber modificado previamente la definición de la clase y su constructor. Quedaría:

```
class Coche():
```

```
  def __init__(self, parametro):
    self.largoChasis=250
    self.anchoChasis=120
    self.ruedas=4
    self.enMarcha=False
    self.color=parametro
```

Observa la última línea. Cuando se cree el objeto le asignará un valor a la propiedad color. Este valor no será el mismo para todos los coches. Será definido por el valor que enviamos como parámetro al realizar la creación del objeto.

14.2 Asignar un valor por defecto a los parámetros

Observa que ocurre en el programa anterior si al crear un objeto de la clase Coche no asignamos un valor al parámetro:

```
class Coche():
    def __init__(self, parametro):
            self.largoChasis = 250
            self.anchoChasis = 120
            self.ruedas = 4
            self.enMarcha = False
            self.color = parametro

miCoche = Coche()
```

Se genera el error:

```
miCoche = Coche()
TypeError: Coche.__init__() missing 1 required positional argument: 'parametro'
process is terminated with return code 1.
```

La razón es que el método constructor está esperando un valor para el parámetro “parametro” y no está recibiendo ninguno.

A veces puede ser muy útil asignar un valor por defecto al parámetro en la orden __init__ del constructor. En caso de que no enviemos ningún valor en la orden de ejecución, el parámetro tomará dicho valor.

La sintaxis es muy sencilla, es suficiente con escribir el signo = seguido del valor por defecto tras el nombre del parámetro. Observa el ejemplo:

```
class Coche():
    def __init__(self, parametro="Azul"):
            self.largoChasis = 250
            self.anchoChasis = 120
            self.ruedas = 4
            self.enMarcha = False
            self.color = parametro

miCoche = Coche()
print(miCoche.color)
```

La llamada Coche() crea un objeto de la clase Coche, la propiedad color toma el valor por defecto (Azul):

```
Azul
```

15. Encapsulación de variables

Encapsular: Proteger una o varias propiedades de una clase para que **no se pueda modificar** (**ni siquiera acceder** a ella) *desde fuera de la clase en que ha sido definida*.

En nuestro ejemplo de los coches esta línea de código:

```
miCoche.ruedas=2
```

Haría que nuestro coche tuviera dos ruedas, lo cual no tiene sentido. Encapsular esta propiedad (ruedas) impedirá que se pueda cambiar el valor de su propiedad a lo largo del código.

Para encapsular una propiedad cuando la definimos hay que preceder su nombre de dos guiones bajos:

```
class Coche():
  def __init__(self):
    self.largoChasis=250
    self.anchoChasis=120
    self.__ruedas=4
    self.enMarcha=False
```

Cuando nos refiramos a esta propiedad en el código habrá que utilizar su nombre completo sin olvidar los dos guiones bajos.

Encapsulando una variable evitamos que se pueda modificar su valor desde fuera del código de la clase, pero **OJO Si que se puede modificar desde su interior**.

En nuestro ejemplo la forma más correcta sería:

```
  def __init__(self):
    self.__largoChasis=250
    self.__anchoChasis=120
    self.__ruedas=4
    self.__enMarcha=False
```

Los tres primeros casos son evidentes. Todos los coches tendrán esas propiedades inalterables.

Modificando el valor de una variable encapsulada desde dentro de su clase

El cuarto caso del ejemplo anterior, no es tan evidente ya que "enMarcha" puede tomar dos valores (True/False). La razón para encapsular su valor es que el programador quiere que esta propiedad solo pueda cambiar su valor desde el procedimiento que hemos definido para ello y este procedimiento está dentro de la clase, por lo que se podrá cambiar su valor. Lo que no podrá hacerse es cambiar su valor desde el resto del código.

Nuestro ejemplo quedaría:

```
class Coche():
  def __init__(self):
    self.__largoChasis=250
    self.__anchoChasis=120
    self.__ruedas=4
    self.__enMarcha=False
  def arrancar(self,arrancamos):
    self.__enMarcha=arrancamos
    if (self.__enMarcha==True):
      return "El coche está en marcha"
    else:
      return "El coche está parado"
  def estado(self):
    print("El coche tiene ", self.__ruedas, "ruedas.  Un ancho de ",
self.__anchoChasis, " y un largo de ", self.__largoChasis)

miCoche=Coche()
print(miCoche.arrancar(True))
miCoche.estado()

print("----Creamos un nuevo objeto: miCoche2---")
miCoche2=Coche()
print(miCoche2.arrancar(False))
miCoche2.estado()
```

16. Encapsulación de métodos

La idea es equivalente a la encapsulación de variables vista en el punto anterior:

> **Encapsular un método:** Hacer que un método sólo sea accesible desde la clase en la que se ha definido.

La sintaxis sigue la misma forma que con las variables encapsuladas. Para encapsular un método es necesario que el nombre del método comience con dos barras bajas. Por ejemplo:

```
def __ejemplo(self):
```

Para entender la utilidad del uso de la encapsulación de métodos volvamos al ejemplo del coche.

Querremos que al llamar al método arrancar, y antes de establecer el valor de la propiedad __enMarcha como True, el coche, de forma autónoma, llame y ejecute otro método que realice un chequeo previo para comprobar que el coche puede arrancar.

En principio crearíamos ese nuevo método de la forma habitual:

```
def chequeo_interno(self):
    print("Realizando chequeo interno")
    self.gasolina="ok"
    self.aceite="ok"
    if(self.gasolina=="ok" and self.aceite=="ok"):
      return True
    else:
      return False
```

Analizando el código:

- El método tiene un único parámetro (self).
- Presuponemos que los coches tienen gasolina y aceite (en realidad habría que comprobarlo)
- Un if determina la respuesta del método. En este caso la respuesta siempre será True ya que hemos establecido que el coche tiene gasolina y aceite.

El objeto coche ha de realizar la comprobación antes de arrancar. Así, el método chequeo_interno ha de ser llamado antes de realizar la acción. Debemos modificar el método arrancar.

Hasta ahora el método tenía la forma:

```
def arrancar(self,arrancamos):
    self.__enMarcha=arrancamos
    if (self.__enMarcha==True):
      return "El coche está en marcha"
    else:
      return "El coche está parado"
```

En resumen:

- La llamada al método envía como parámetro un valor (True y False).
- Guardamos ese parámetro dentro de una variable encapsulada __enMarcha.
- En función de que el valor enviado sea True o False, arrancamos el coche o no.

Modifiquemos este código teniendo en cuenta que debemos realizar la llamada al método de chequeo. En primer lugar, hay que tener en cuenta que el chequeo solamente se realizará cuando el método arrancar esté enviando como parámetro el valor True (es decir cuando estamos intentando arrancar). Una posible opción sería:

```
def arrancar(self,arrancamos):
    if  arrancamos== True:
      chequeo=self.chequeo_interno()
      if (chequo==True):
        self.__enMarcha= True
        return "El coche está en marcha"
      else:
        return " Algo ha ido mal en el chequeo. No es posible arrancar "
    else:
        return " El coche está parado "
```

En primer lugar, comprobamos que valor toma el parámetro del método arrancamos. Si es True (queremos arrancar) comprobamos si el coche puede arrancar a través del método chequeo_interno. Si este método genera True, arrancamos, si no lo hace, informamos de la imposibilidad de realizar el arranque. En caso de que hayamos enviado el valor False como parámetro en el método arrancar, el coche estará parado.

Juntándolo todo, la clase coche quedaría:

```
class Coche():
  def __init__(self):
    self.__largoChasis=250
    self.__anchoChasis=120
    self.__ruedas=4
    self.__enMarcha=False
  def arrancar(self,arrancamos):
    if  arrancamos== True:
      chequeo=self.chequeo_interno()
      if (chequo==True):
        self.__enMarcha= True
        return "El coche está en marcha"
      else:
        return " Algo ha ido mal en el chequeo. No es posible arrancar "
    else:
        return " El coche está parado "
  def estado(self):
    print("El coche tiene ", self.__ruedas, "ruedas.  Un ancho de ",
self.__anchoChasis, " y un largo de ", self.__largoChasis)
  def chequeo_interno(self):
    print("Realizando chequeo interno")
    self.gasolina="ok"
    self.aceite="ok"
    if(self.gasolina=="ok" and self.aceite=="ok"):
      return True
    else:
      return False
```

Si ejecutamos el programa todo parece funcionar de forma correcta:

```
miCoche=Coche()
print(miCoche.arrancar(True))
```

```
miCoche.estado()

print("----Creamos un nuevo objeto: miCoche2---")
miCoche2=Coche()
print(miCoche2.arrancar(False))
miCoche2.estado()
```

Se ejecuta el chequeo para el objeto miCoche y como todo es correcto, miCoche arranca. En el caso de miCoche2, como no lo queremos arrancar no se ejecuta el chequeo.

```
Realizando chequeo interno
El coche está en marcha
El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250
----Creamos un nuevo objeto: miCoche2---
El coche está parado
El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250
```

Hagamos ahora que la variable aceite no esté ok, le asignamos un valor "mal". En ese caso el programa responde también de forma correcta:

```
Realizando chequeo interno
Algo ha ido mal en el chequeo. No es posible arrancar
El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250
----Creamos un nuevo objeto: miCoche2---
El coche está parado
El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250
```

Todo parece funcionar bien, sin embargo, el método chequeo_interno no está encapsulado, por lo tanto, es accesible desde cualquiera de los objetos. Eso no tiene ningún sentido, observa que ocurre si modificamos el código del primer objeto en la forma:

```
miCoche=Coche()
print(miCoche.arrancar(True))
miCoche.estado()
print(miCoche.chequeo_interno())
```

Generaremos la respuesta:

```
Realizando chequeo interno
El coche está en marcha
El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250
Realizando chequeo interno
True
```

Una vez que el coche ya ha arrancado, vuelve a realizar el chequeo porque lo llamamos desde el código, generando un valor True.

Todavía tiene menos sentido hacer el chequeo desde el segundo objeto que está parado. Sin embargo, se podría hacer:

```
miCoche2=Coche()
print(miCoche2.arrancar(False))
miCoche2.estado()
print(miCoche2.chequeo_interno())
```

Generaría:

```
----Creamos un nuevo objeto: miCoche2---
El coche está parado
El coche tiene 4 ruedas. Un ancho de 120 y un largo de 250
Realizando chequeo interno
True
```

Para evitar estos problemas encapsularemos el método. Su uso quedará restringido al interior de la clase.

La sintaxis es sencilla:

- Comenzar con dos guiones bajos su nombre en la definición.

En nuestro ejemplo:

```
class Coche():
    def __init__(self):
        self.__largoChasis=250
        self.__anchoChasis=120
        self.__ruedas=4
        self.__enMarcha=False
    def arrancar(self,arrancamos):
        if  arrancamos== True:
            chequeo=self.__chequeo_interno()
            if (chequo==True):
                self.__enMarcha= True
                return "El coche está en marcha"
            else:
                return " Algo ha ido mal en el chequeo. No es posible arrancar "
        else:
            return " El coche está parado "
    def estado(self):
        print("El coche tiene ", self.__ruedas, "ruedas.  Un ancho de ",
self.__anchoChasis, " y un largo de ", self.__largoChasis)
    def __chequeo_interno(self):
        print("Realizando chequeo interno")
        self.gasolina="ok"
        self.aceite="ok"
        if(self.gasolina=="ok" and self.aceite=="ok"):
            return True
        else:
            return False
```

Si intentamos ahora ejecutar el método desde los objetos:

```
miCoche=Coche()
print(miCoche.arrancar(True))
miCoche.estado()
print(miCoche.chequeo_interno())
```

Obtendremos:

```
Realizando chequeo interno
El coche está en marcha
El coche tiene 4 ruedas.  Un ancho de 120 y un largo de 250
Traceback (most recent call last):
  File "...Ensayos curos\src\encapsularmetodos.py", line 31, in <module>
    print(miCoche.__chequeo_interno())
AttributeError: 'Coche' object has no attribute '__chequeo_interno'
```

El programa detecta que estamos intentando ejecutar el método encapsulado desde fuera de la definición de clase y por lo tanto da un error.

Si retiramos las llamadas al método chequeo_interno desde los objetos todo funcionará de forma correcta.

17. Método __str__

El método __str__ se ejecutará cuando llamemos desde el código el nombre del objeto. **Se utiliza para devolver a través de la orden return una cadena String al código principal**.

Veamos un ejemplo. Sea la clase Animal

```
class Animal:
  def __init__(self, nombre):
    self.nombre = nombre
```

De la cual hemos creado un objeto:

```
perro = Animal('bobby')
```

Si llamamos la propiedad nombre del objeto:

```
print(perro.nombre)
```

Obtendríamos como resultado el valor de la propiedad.

```
bobby
```

Sin embargo si ejecutamos la función print sobre el objeto:

```
print(perro)
```

Obtenemos como resultado:

```
<__main__.Animal object at 0x1053fab38>
```

El método __str__ nos permitirá definir un procedimiento que devuelva una cadena de texto, en este caso el nombre del objeto.

```
class Animal:
  def __init__(self, nombre):
    self.nombre = nombre
  def __str__(self):
    return self.nombre
```

Si añadimos

```
perro = Animal('bobby')
print(perro)
```

Se mostrará el resultado:

```
bobby
```

18. Herencia

Herencia: Procedimiento mediante el cual una clase adquiere características y comportamientos de otra clase de un nivel jerárquico superior.

Sea el ejemplo una serie de cinco clases relacionadas según la siguiente jerarquía de herencia.

La clase 1, ocupa el primer nivel en la escala jerárquica, recibe el nombre **clase padre** o **superclase**.

La clase 2 heredará propiedades de la clase 1 por lo que es **subclase** de esta.

Por otro lado, las clases 3, 4 y 5 heredarán propiedades de la clase 2. Por la tanto la clase 2 es **superclase** de ellas y 3, 4 y 5 son subclases de 2.

18.1 Finalidad de la herencia en programación

Trabajar con herencia nos permite **reutilizar código**.

Volviendo al ejemplo del coche. Supongamos queremos crear ahora los siguientes objetos:

En primer lugar, nos planearemos que características tienen en común todos los objetos que vamos a crear.

- Todos estos objetos tendrán en común dos propiedades (por ejemplo). Todos tendrán una marca y todos serán un modelo.
- Tendrán tres comportamientos en común: Arrancan, aceleran y frenan.
- Si creáramos los objetos a partir de clases independientes, para todos ellos habría que programar su código desde cero.
- La solución es crear una superclase que contenga todas las características y procedimientos comunes a los distintos tipos de objetos.
- Posteriormente construiremos subclases que hereden las características de esa clase padre y le añadiremos a cada una las peculiaridades de ese tipo de vehículo (por ejemplo, número de ruedas).

18.2 Sintaxis

Guarda todo el código que veamos en este punto en un módulo con nombre herencia.py, lo reutilizaremos más adelante.

Superclase

En primer lugar, crearemos la superclase, la llamamos Vehículos, seguimos las normas habituales:

```
class Vehiculos():
    def __init__(self, marca, modelo):
        self.marca=marca
        self.modelo=modelo
        self.enMarcha=False
        self.acelera=False
        self.frena=False
    def arrancar(self):
        self.enMarcha=True
    def acelerar(self):
        self.acelera=True
    def frenar(self):
        self.frena=True
    def estado(self):
        print("Marca:", self.marca,"\nModelo:", self.modelo,"\nEn Marcha:",
self.enMarcha,"\nAcelerando:", self.acelera,"\nFrenando:", self.frena)
```

La superclase contiene:

- Un constructor que ha de recibir por parámetros (además de self) marca y modelo.
- Dentro del constructor se definirá la propiedad marca y modelo según los valores que introduzcamos al llamar al constructor y además agregamos tres propiedades (enMarcha, acelera y frena) cuyos valores por defecto son False (al construir el objeto está quieto).
- Tres métodos que permiten arrancar, acelerar y frenar.
- Un método que permite mostrar el estado del vehículo.

Clase moto que hereda propiedades y métodos de la clase Vehiculo

Definimos la clase igual que hasta ahora pero dentro de los paréntesis incluimos el nombre de la superclase de la cual la nueva clase tiene que heredar. En nuestro ejemplo:

```
class Moto(Vehiculos):
    pass
```

(pass se correspondería con el código particular de la clase Moto, de momento no añado nada).

Si ahora creamos un objeto de la clase Moto (ten en cuenta que hay que pasarle los parámetros marca y modelo) y a continuación ejecutamos el método estado:

```
miMoto=Moto("Honda", "Special")
miMoto.estado()
```

El resultado será:

```
Marca: Honda
Modelo: Special
En Marcha: False
Acelerando: False
```

```
Frenando: False
```

18.3 Añadir propiedades y métodos a una subclase

En el ejemplo anterior hemos creado un objeto de la clase Moto, hija de la clase Vehiculos. Sin embargo, la clase Moto carecía de código propio.

Lo más normal es que una vez definida la clase Moto, está tenga un código propio que la diferencie de otras clases (Camión, Autobus…).

Vamos a crear un comportamiento propio característico de una moto. A modo de ejemplo vamos añadir un comportamiento que permita hacer el caballito (es algo que un camión o autobús no podrá hacer).

```
class Moto(Vehiculos):
    hcaballito=""
    def caballito(self):
        self.hcaballito="Voy haciendo el caballito"
```

Asigno a la clase moto la propiedad hcaballito, a la que asigno el valor cadena de texto vacía y un método que consiste en hacer el caballito.

Con ello un objeto moto dispondrá ahora de seis métodos (los cinco heredados y el propio).

18.4 Sobrescribir un método heredado

Para ejecutar el caballito en principio ejecutaríamos:

```
miMoto.caballito()
```

El programa funciona, no nos da error, pero la información es incompleta:

```
Marca: Honda
Modelo: Special
En Marcha: False
Acelerando: False
Frenando: False
```

Como el método estado, ha sido definido en la clase Vehiculos, no puede informarnos de si la moto está haciendo el caballito o no. Una forma de solucionar esto, es sobrescribir el método heredado.

Sobrescribir el método heredado consiste en crear un método propio para la clase hijo que sustituirá al de la clase padre.

Para ello basta crear dentro de la clase hijo un nuevo método con el mismo nombre que el que queremos sustituir y con los mismos parámetros.

```
class Moto(Vehiculos):
    hcaballito=""
    def caballito(self):
        self.hcaballito="Voy haciendo el caballito"
    def estado(self):
        print("Marca:", self.marca,"\nModelo:", self.modelo,"\nEn Marcha:",
self.enMarcha,"\nAcelerando:", self.acelera,"\nFrenando:", self.frena,"\n",
self.hcaballito)
```

Nos dará como resultado:

```
Marca: Honda
Modelo: Special
En Marcha: False
Acelerando: False
Frenando: False
 Voy haciendo el caballito
```

18.5 Subclases con parámetros propios - Función super()

Vamos a partir de un ejemplo nuevo. En el definiremos la superclase Persona:

```
class Persona():
  def __init__(self, nombre, edad, lugar_residencia):
    self.nombre = nombre
    self.edad = edad
    self.lugar_residencia = lugar_residencia
  def descripcion(self):
    print("Nombre:", self.nombre, "Edad:", self.edad, "Resiencia:",
self.lugar_residencia)
```

Vamos a crear una subclase de la clase Persona que defina un tipo especial de persona. Esta subclase se llamará Empleado.

Queremos que los objetos de la clase Empleado tengan las mismas propiedades que los objetos de su superclase (nombre, edad y lugar de residencia) pero además tendrán dos propiedades específicas: salario y antigüedad.

Por lo tanto cuando vayamos a crear un objeto de la clase Empleado habrá que enviar cinco parámetros. Tres de ellos se definen a través del constructor de la superclase.

La solución será definir la subclase en la forma:

```
class Empleado(Persona):
  def __init__(self, nombre, edad, lugar_residencia, salario, antiguedad):
    self.nombre = nombre
    self.edad = edad
    self.lugar_residencia = lugar_residencia
    self.salario = salario
    self.antiuedad = antiguedad
```

Trabajar de esta forma es posible, pero no tiene mucho sentido. Estamos perdiendo las posibilidades que nos ofrece que Empleado sea una subclase de la superclase Persona.

Las tres líneas de código en rojo, están incluidas en el constructor de la superclase. Es mucho más sencillo llamar al constructor de la superclase desde la subclase:

```
class Empleado(Persona):
  def __init__(self, nombre, edad, lugar_residencia, salario, antiguedad):
    Persona.__init__(self, nombre, edad, lugar_residencia)
    self.salario = salario
    self.antiuedad = antiguedad
```

La línea en rojo, llama al método constructor de la clase Persona y con ello conseguimos crear el objeto de la subclase aprovechando la herencia.

Esta sintaxis puede resultar un poco confusa ya que en la misma línea necesitamos referirnos dos veces a la superclase (una vez con su nombre y otra con el parámetro self). Esto se puede evitar utilizando la función super:

Función super(): Función que permite llamar a un método de la clase padre de una clase hijo.

En este caso queremos llamar al método constructor de la clase padre Persona. La sintaxis sería:

```
class Empleado(Persona):
  def __init__(self, nombre, edad, lugar_residencia, salario, antiguedad):
    super().__init__(Persona, nombre, edad, lugar_residencia)
    self.salario = salario
    self.antiuedad = antiguedad
```

Mostrar el estado completo utilizando la función super():

Sí aplicamos el método descripcion que creamos para la clase Persona a un objeto de la sublase Empelado, no mostrará sus cinco propiedades, si no que solo mostrará las tres primeras. Tal y como vimos en el punto anterior, es necesario sobreescribir el método descripcion heredado por uno nuevo que podría tener la forma:

```
def descripcion(self):
    print("Nombre:", self.nombre, "Edad:", self.edad, "Resiencia:",
self.lugar_residencia)
    print("Salario:", self.salario,"Antigüedad:",self.antiguedad)
```

Sin embargo, si observas la línea en rojo, esa información ya es mostrada por el método descripcion de la superclase, por lo tanto podemos utilizar la función super():

```
def descripcion(self):
    super().descripcion()
    print("Salario:", self.salario,"Antigüedad:",self.antiguedad)
```

Simplificando el código y aprovechando lo que ya hemos creado en un bloque anterior.

En conjunto:

```
class Persona():
  def __init__(self,nombre,edad, lugar_residencia):
    self.nombre=nombre
    self.edad=edad
    self.lugar_residencia=lugar_residencia
  def descripcion(self):
    print("Nombre:",self.nombre, "Edad:", self.edad,
"Residencia:",self.lugar_residencia)
class Empleado(Persona):
  def __init__(self,salario,antiguedad, nombre_empleado, edad_empleado,
residencia_empleado):
    super().__init__(nombre_empleado, edad_empleado, residencia_empleado)
    self.salario=salario
    self.antiguedad=antiguedad
  def descripcion(self):
    super().descripcion()
    print("Salario:", self.salario,"Antigüedad:",self.antiguedad)
Antonio=Empleado(1500,15,"Antonio",17,"Zaragoza")
Antonio.descripcion()
```

Dando como resultado:

```
Nombre: Antonio Edad: 17 Residencia: Zaragoza
Salario: 1500 Antigüedad: 15
```

19. Identificar la clase a la que pertenece un objeto

En Python, puedes determinar a qué clase pertenece un objeto utilizando la función type() o el método isinstance().

19.1 Función type():

Función type(): Devuelve el tipo de un objeto.

La sintaxis de la función es:

```
type(nombre_del_objeto)
```

La función devuelve un objeto del tipo 'type'

Para el siguiente código:

```
python
class MiClase:
   pass

objeto = MiClase()
print(type(objeto))
```

Obtenemos la respuesta:

```
<class '__main__.MiClase'>
```

19.2 Función isinstance()

Función isinstance(): Informa de si un objeto es instancia de una clase.

La sintaxis de la función es:

```
Isinstance(nombre_del_objeto, nombre_de_la_clase)
```

La función devuelve el valor True si el objeto de nombre nombre_de_objeto pertenece a la clase de nombre nombre_de_la_clase y False si no lo hace.

En el caso anterior:

```
class MiClase:
   pass
objeto = MiClase()
print(isinstance(objeto, MiClase))
```

Generará por pantalla:

```
True
```

Es importante tener en cuenta que isinstance() también considera la herencia, es decir, si el objeto pertenece a cualquiera de las clases de su "árbol de herencia".

20. Herencia múltiple

Es algo complejo y no lo utilizaremos habitualmente, sin embargo, vale la pena comprenderlo.

Herencia múltiple: Proceso en el que una nueva clase hereda características de varias clases padre.

Sigamos con el ejemplo anterior.

Creemos ahora una nueva clase padre que defina las características que van a tener todos los vehículos eléctricos. Podría ser:

```
class VElectricos():
   def __init__(self):
      self.autonomia=100
   def cargarEnergía(self):
      self.cargando=True
```

Es una clase independiente que no hereda de nadie y que tiene un constructor que define la autonomía de este tipo de vehículos y un método que permite cargar energía.

Si ahora queremos crear un nuevo tipo de vehículo como por ejemplo una bicicleta eléctrica, nos interesaría que heredara las características de la clase Vehiculo pero también las de la clase VElectrico.

20.1 Sintaxis

Basta con definir la nueva clase por su nombre y entre paréntesis llamar a todas aquellas clases de las cuales vaya a heredar. En este caso:

```
class BicicletaElectrica(Vehiculos,VElectricos):
   pass
```

Puedo ahora crear objetos de la clase BicicletaElectrica con una llamada del tipo:

```
miBici=BicicletaElectrica("Yamaha",HC098")
```

El orden en el que pongamos el nombre de las clases padre a la hora definir la clase hijo es muy importante ya que se da prioridad a la clase que se indica en primer lugar. Así:

- Si ambas clases tienen un método con el mismo nombre, se heredará el método de la clase padre que se escriba en primer lugar en la definición de la clase hijo.

- El número de parámetros que tengo que definir para crear el objeto hijo será igual al número de parámetros que tenga la primera de las funciones padre (en nuestro ejemplo he llamado primero a Vehiculos, por lo tanto hay que definir sus dos parámetro, si hubiera llamado primero a VElectricos no habría que poner ninguno.

21. Polimorfismo

Polimorfismo: Un objeto puede cambiar de forma (propiedades y comportamientos) en función de cuál sea el contexto en el que se utilice.

Siguiendo con el ejemplo de los vehículos, sería hacer que objeto coche pase a ser un camión…

Veamos un nuevo ejemplo:

```
class Coche():
   def desplazamiento(self):
      print("Me desplazo utilizando cuatro ruedas")
class Moto():
   def desplazamiento(self):
      print("Me desplazo utilizando dos ruedas")
class Camion():
   def desplazamiento(self):
      print("Me desplazo utilizando seis ruedas")
```

Hemos creado tres clases. Cada una de ellas tiene un método al que hemos llamado desplazamiento, pero el texto generado por cada uno de ellos es diferente.

Crearemos ahora un objeto moto y ejecutemos el método desplazamiento:

```
miVehiculo=Moto()
miVehiculo.desplazamiento()
```

El resultado por pantalla será:

```
Me desplazo utilizando dos ruedas
```

Si ahora queremos que nuestro objeto pase a ser de la clase Coche habrá que hacer lo siguiente:

Creamos una función que va a recibir por parámetro el nombre del tipo de clase correspondiente al tipo de vehículo que queremos crear:

```
def desplazamientoVehiculo(vehiculo):
   vehiculo.desplazamiento()
```

Esta función llama al método desplazamiento de la clase asociada a la clase definida por el parámetro vehiculo. Así para llamar a la función, en el caso de que el objeto sea una moto:

```
miVehiculo=Moto()
desplazamientoVehiculo(miVehiculo)
```

La respuesta será:

```
Me desplazo utilizando dos ruedas
```

Si queremos que este objeto pase a comportarse como un coche bastaría con volver a definir su clase:

```
miVehiculo=Moto()
desplazamientoVehiculo(miVehiculo)
miVehiculo=Camion()
desplazamientoVehiculo(miVehiculo)
```

Daría por resultado:

```
Me desplazo utilizando dos ruedas
Me desplazo utilizando seis ruedas
```

Inicalmente el objeto pertenece a la clase Moto y a continuación a la clase Camion.

22. POO. Métodos de cadenas – Objetos clase String

Python es un lenguaje POO. Todos los elementos que utilizamos en nuestros programas son objetos. Como tales tendrán propiedades, estados y métodos.

En este punto vamos a ver a estudiar las cadenas de texto.

En Python una cadena de texto (una expresión entre comillas) es un objeto de la clase **String**.

Por ser un objeto tendrá asociados una serie de métodos, asociados a su clase, que nos van a facilitar mucho tareas que de otra forma serían más complejas.

La método más habituales de un objeto de la case String son:

Método	**Descripción**
upper()	Convierte en mayúsculas todos los caracteres de la cadena.
lower()	Convierte en minúsculas todos los caracteres de la cadena.
title()	La primera letra de cada palabra pasa a ser mayúscula.
capitalize()	La primera letra de la cadena pasa a ser mayúscula, el resto serán minúsculas.
count()	Cuenta cuantas veces aparece un carácter (o caracteres) en la cadena.
rstrip())	Elimna los espacios en blanco al final de la cadena.
find()	Da la posición de un carácter o grupo de caracteres en la cadena.
isdigit()	Devuelve True/False en función de que la cadena sea un número o no.
isalum()	Devuelve True/False en función de que la cadena ase alfanumérica o no.
isalpha()	Devuelve True/False en función de que la cadena contiene solamente letras.
split()	Separa por palabras utilizando espacios como referencia.
strip()	Elimina espacios en blanco al comienzo y al final.
replace()	Cambia un carácter o grupo por otro/s.
rfind()	Equivalente a find() pero da índices comenzando desde el final de la cadena.
removeprefix('xxx')	Elimina los caracteres 'xxx' al comienzo de la cadena de texto.
removesuffix('xxx')	Elimina los caracteres 'xxx' al final de la cadena de texto.

Existen muchos métodos más para la clase String. Explicarlos todos aquí, carece de sentido. De forma similar alguno de los métodos que aparecen en la lista anterior tienen alguna peculiaridad sintáctica que no es necesario saber de memoria. Es mucho mejor saber dónde buscar.

Si buscamos en Google "Documentación de Python" el primer enlace debería de ser el sitio http://pyspanishdoc.sourceforge.net/. Haciendo clic en el vínculos "Referencia de bibliotecas", llegaremos a una tabla de contenidos desde la que podemos obtener información sobre todos los elementos incluidos en la biblioteca de Python.

En este caso la información sobre los métodos de cadenas está incluida en punto 2.1.5 Tipos secuenciales. Bajando al final de la página, tenemos el punto 2.1.5.1 que nos habla sobre los métodos de las cadenas.

Este punto nos dará una descripción de cada uno de los métodos y de los parámetros que son necesario pasar al método para que funcionen de forma correcta.

Así por ejemplo queremos utilizar el método count, obtenemos la siguiente información:

> **count** (sub[, start[, end]])
>
> Devuelve cuántas veces aparece sub en la cadena S[start:end]. Los argumentos opcionales start y end se interpretan según la notación de corte.

Nos informa de la utilidad de este método, pero también de los tres parámetros que tenemos que introducir:

- Sub es el carácter o cadena a buscar.
- Start y end índices de comienzo y final del segmento de búsqueda.

Estos métodos se utilizan igual que cualquiera de los métodos que hemos visto en los puntos anteriores:

Veamos un ejemplo sencillo, observa el siguiente código:

```
nombreUsuario=input("Introduce tu nombre de Usuario")
print("El nombre es:", nombreUsuario )
```

Si queremos forzar que el print nos muestre el nombre del usuario en mayúsculas, sería suficiente con:

```
nombreUsuario=input("Introduce tu nombre de Usuario")
print("El nombre es:", nombreUsuario.upper() )
```

El funcionamiento del programa sería:

```
Introduce tu nombre de Usuario lucas
El nombre es:  LUCAS
```

23. Dando formato con format

Vamos a ver como utilizar la expresión format para dar formato a un número como para crear cadenas de texto complejas.

Aunque la expresión sea la misma. Cuando la utilizamos para formatear números, se utilizará como una función y cuando la utilicemos para crear cadenas String complejas, será un método.

23.1 Formatea la presentación de números con la función format

Podemos utilizar la **función format** para definir varios aspectos de la presentación de un valor numérico por pantalla. Aquí tenéis varios ejemplos:

```
numero = 3141.59265

# Dos decimales de precisión:
print(format(numero, '0.2f'))

# Jusitificación a la derecha con diez caracteres, un decimal de precisión:
print(format(numero, '>10.1f'))

# Jusitificación a la izquierda con diez caracteres, un decimal de precisión:
print(format(numero, '<10.1f'), "hola")

# Separador de miles:
print(format(numero, ','))
print(format(numero, '0,.1f'))

# Notación científica:
print(format(numero, 'e'))
print(format(numero, '0.2E'))
```

23.2 Método format para crear cadenas String complejas

Vamos a ver en detalle un método de los objetos String que puede ser muy útil. Es el método format que permitirá crear cadenas de texto que incluyan, por ejemplo, valores de variables.

Considera un programa que te pregunte tu nombre y tu edad. A continuación el programa debe generar un saludo que incluya tu nombre y edad.

Tradicionalmente utilizaríamos un código similar a:

```
nombre = input("Introduce tu nombre: ")
edad = int(input("Introduce tu edad: "))
print("Hola", nombre, ", tienes", edad, "años")
```

Generando un resultado:

```
Introduce tu nombre: Andrés
Introduce tu edad: 17
Hola Andrés , tienes 17 años.
```

Si te fijas, tras el nombre del usuario y antes de la coma, el programa ha dejado un espacio en blanco que formalmente no debería existir. Ese espacio se debe a que cuando utilizamos la función print con varios parámetros separados por comas, print añade un espacio en blanco.

Hasta ahora resolvíamos este problema utilizando el concatenador "+" en lugar de la coma. Esta forma de trabajar puede crear otros problemas. Por ejemplo en este caso, si sustituimos las comas por el concatenador, el código sería:

```
print("Hola " + nombre + " tienes " + edad + " años.")
```

Esta línea va a generar un error. ¿Puedes identificarlo?

La variable edad es un número entero. Por lo tanto el operador + no sabe si tiene que actuar como un concatenador o como símbolo suma:

```
Introduce tu nombre: Andrés
Introduce tu edad: 17
    Traceback (most recent call last):
    print("Hola " + nombre + ", tienes" + edad + " años.")
    TypeError: can only concatenate str (not "int") to str
```

Para evitar este error necesitaríamos transformar el objeto edad que es un entero en un objeto de tipo string, utilizando la función str:

```
print("Hola " + nombre + ", tienes " + str(edad) + " años.")
```

Además, hay que considerar los espacios en blanco a añadir dentro de los segmentos de texto.

Poco a poco vamos generando una expresión complicada y difícil de entender.

En lugar de hacer esto vamos a utilizar el método format.

El método format() en Python permite crear cadenas compuestas por partes estáticas y partes variables, donde los valores de estas variables se insertan en la cadena en posiciones específicas definidas por marcadores de posición.

Sintaxis básica:

```
cadena_formateada = cadena_original.format(valor1, valor2, ...)
```

Donde cadena_original es una cadena de texto que contiene ***marcadores de posición***, y valor1, valor2, etc., son los valores que se insertarán en lugar de los marcadores de posición.

Los marcadores de posición se indican mediante llaves `{}`.

Lo entenderás mejor estudiando el siguiente código:

```
nombre = input("Introduce tu nombre: ")
edad = int(input("Introduce tu edad: "))
cadena_original = "Hola {}, tienes {} años."
cadena_formateada = cadena_original.format(nombre, edad)
print(cadena_formateada)
```

La tercera línea crea el objeto String "Hola {}, tienes {} años.".

En la cuarta línea ejecutamos el método format sobre el objeto tipo String y utilizamos como parámetros los valores almacenados en las variables nombre y edad.

La función format genera una nueva cadena de texto en la que los marcadores de posición {} son sustituidos por cada uno de los parámetros.

El resultado será:

```
Introduce tu nombre: Andrés
Introduce tu edad: 17
Hola Andrés, tienes 17 años.
```

Puede parecer un poco confuso, pero eso es así porque estamos utilizando, para que se entienda mejor, dos variables que en realidad no necesitamos. Observa el código ahora:

```
nombre = input("Introduce tu nombre: ")
```

```
edad = int(input("Introduce tu edad: "))
print("Hola {}, tienes {} años.".format(nombre, edad))
```

Estamos aplicando el método sobre la cadena de texto (sin tener que asignarle ningún nombre) y enviando los dos parámetros que en este caso necesitamos.

Por último si queremos aumentar la claridad del código podríamos asignar un nombre al valor de los parámetros en la forma:

```
print("Hola {age}, tienes {name} años.".format(age=edad, name=nombre))
```

Observa que el texto a escribir queda claramente definido y que incluso hemos cambiar el orden en el que asignamos los parámetros a los marcadores de posición.

24. Módulos que incluyen clases

En el punto 5 de estos apuntes tratamos el tema de los módulos. Ahora que ya hemos visto la parte correspondiente a POO, veamos como estos módulos pueden incluir definiciones de clases.

Veamos un ejemplo. Vamos a crear un módulo nuevo que va a contener una de las clases contenidas en uno de los módulos que habíamos creado en puntos anteriores.

Creamos un nuevo módulo al que llamaremos modulo_vehiculos.py. Vamos a copiar y pegar en este módulo ***las clases*** incluidas en el archivo el archivo herencia.py que creamos en un punto anterior de estos apuntes.

```
class Vehiculos():
    def __init__(self, marca, modelo):
        self.marca=marca
        self.modelo=modelo
        self.enMarcha=False
        self.acelera=False
        self.frena=False
    def arrancar(self):
        self.enMarcha=True
    def acelerar(self):
        self.acelera=True
    def frenar(self):
        self.frena=True
    def estado(self):
        print("Marca:", self.marca,"\nModelo:", self.modelo,"\nEn Marcha:",
self.enMarcha,"\nAcelerando:", self.acelera,"\nFrenando:", self.frena)

class VElectricos():
    def __init__(self):
        self.autonomia=100
    def cargarEnergía(self):
        self.cargando=True

class Moto(Vehiculos):
    hcaballito=""
    def caballito(self):
        self.hcaballito="Voy haciendo el caballito"
    def estado(self):
        print("Marca:", self.marca,"\nModelo:", self.modelo,"\nEn Marcha:",
self.enMarcha,"\nAcelerando:", self.acelera,"\nFrenando:", self.frena,"\n",
self.hcaballito)


class BicicletaElectrica(Vehiculos,VElectricos):
    pass
```

Cerramos el módulo herencia.py y continuamos programando con el modulo modulo_vehiculos.py.

Vamos a utilizar ahora este código en un archivo nuevo, uso_vehiculos.py.

Para utilizar las clases del módulo anterior utilizaremos la directiva:

```
from modulo_vehiculos import *
```

A partir de ese momento podremos utilizar las clases contenidas en el modulo modulo_vehiculos.py dentro del modulo uso_vehiculos.py.

```
from modulo_vehiculos import *
miCoche=Vehiculos("Mazda", "XMS")
miCoche.estado()
```

Mostrará por pantalla:

```
Marca: Mazda
Modelo: XMS
En Marcha: False
Acelerando: False
Frenando: False
```

25. Libro de estilo

Libro de estilo de un lenguaje de programación: Conjunto de recomendaciones sobre la forma de dar formato a los programas.

Utilizar un estilo específico incrementa la legibilidad del código, facilita su reutilización y la detección de errores.

El estilo utilizado en estos apuntes y que se recomienda sigan los alumnos se basa en la guía de estilo oficial de Python.

La guía de estilo oficial de Python se encuentra en https://www.python.org/dev/peps/pep-0008/. Recogemos aquí algunas de las recomendaciones principales de esta guía:

Sangrado

Sangrar con espacios en vez de tabuladores (Eclipse escribe espacios cuando se pulsa un tabulador).

Usar 4 espacios en cada nivel de sangrado.

Longitud de línea

Las líneas no deben contener más de 80 caracteres. Si una línea tiene más de 80 caracteres, se debe dividir en varias líneas (este valor se toma de la longitud de las antiguas tarjetas perforadas.

Es posible hacer visible una línea que muestre este tope de 80 caracteres (o cualquier otro que nos pudiera interesar). La ruta para ello es: Preferences ->General ->Editors -> Text Editors -> Show Print Margin

Se aconseja sangrar la línea partida para distinguirla del resto del programa.

Hay varias formas de partir las líneas:

Si una expresión larga está entre paréntesis (o corchetes o llaves), la línea se puede partir en cualquier lugar, aunque se recomienda hacerlo después de comas o de un operador:

En el caso de argumentos de funciones, se recomienda hacerlo después de una coma:

```
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
var_three, var_four)
```

En el caso de expresiones lógicas, pueden añadir paréntesis para poder partirlas después de un operador:

```
if (condicion1 and condicion2 and condicion3 and
condicion4):
```

Si no hay paréntesis (ni corchetes ni llaves), la línea se puede partir utilizando la contrabarra (\).

En el caso de expresiones lógicas, se aconseja hacerlo después de un operador:

```
if condicion1 and condicion2 and condicion3 and \
condicion4:
```

En el caso de cadenas, se aconseja partir la cadena en varias cadenas en vez de usar la contrabarra.

```
print("Esta línea está cortada en dos líneas de menos de 79 caracteres \
usando una contrabarra")
```

```
print("Esta línea está cortada en dos líneas de menos de 79 caracteres",
   "partiendo la cadena en dos")
```

Instrucciones por línea

Aunque Python permite escribir varias instrucciones por línea separándolas por puntos y comas (;), se recomienda escribir una única instrucción por línea. Así en lugar de:

```
print("Hola"); print("Adiós")
```

Se recomienda:

```
print("Hola")
print("Adiós")
```

Espacios en blanco

Evitar espacios en las siguientes situaciones:

Justo en el interior de paréntesis, corchetes o llaves:	**Sí:** spam(ham[1], {eggs: 2}) No: spam(ham[1], { eggs: 2 })
Justo antes de una coma, punto y coma o dos puntos:	**Sí:** if x == 4: print x, y; x, y = y, x No: if x == 4 : print x , y ; x , y = y , x
Justo antes de abrir el paréntesis de una función:	**Sí:** spam(1) No: spam (1)
Justo antes de los corchetes que indican un índice:	**Sí:** dict['key'] = list[index] No: dict ['key'] = list [index]
No se debe alinear operadores de distintas líneas usando espacios:	**Sí:** x = 1 y = 2 long_variable = 3 No: x = 1 y = 2 long_variable = 3

Otras recomendaciones

Escribir un espacio antes y después de los operadores: asignación (=), asignación aumentada (+=, -= etc.), comparaciones (==, <, >, !=, <>, <=, >=, in, not in, is, isnot), booleanos (and, or, not).

Si se utilizan operadores con distintas prioridades, se aconseja utilizar espacios en los operadores de menor prioridad. Nunca escriba más de un espacio y escriba los mismos espacios antes y después de cada operador.

```
Sí:
    i = i + 1
    submitted += 1
    x = x*2 - 1
    hypot2 = x*x + y*y
     c = (a+b) * (a-b)
No:
    i=i+1
    submitted +=1
     x = x * 2 - 1
    hypot2 = x * x + y * y
     c = (a + b) * (a - b)
```

Se recomienda que el nombre del programa no contenga espacios sino guiones o subrayados. Por ejemplo, aritmeticas_1.py.

Comentarios en los programas

Es importante añadir comentarios a los programas para que sean más comprensibles para los humanos (el ordenador ignora completamente el contenido de los comentarios). En Python un comentario empieza por el carácter almohadilla (#) y acaba al final de la línea. No es necesario comentar todas las líneas, sino sólo aquellas que lo necesiten. Por ejemplo:

```
ep = 166.386 # Factor de conversión de euros a pesetas
```

Utilizando PyDev es posible comentar un bloque de código completo seleccionándolo y pulsando la combinación de teclas Ctrl+4 (Ctrl+5 para descomentar).

Cabecera de los programas

Para poder identificar qué hace cada programa, quién lo ha escrito y cuándo, se aconseja que los programas incluyan una cabecera similar a esta, hecha a base de comentarios:

```
# Archivo: XXXXXXX_XX.py
# Autor: XXXXX XXXXXXX (nombre y apellidos)
# Fecha: XX de XXXXXX de XXXX
# Descripción: Ejercicio XXXXXXX. Este programa blablabla
# Si la descripción es larga, escriba varias líneas
```

26. Funciones

Listado de las funciones más habituales.

Función	Descripción
str()	Transforma un número en cadena de caracteres.
int()	Transforma una cadena de caracteres en un número entero.
float()	Transforma una cadena de caracteres en un número decimal.
round(a,b)	Redondea el número a con b decimales.
chr(num)	Devuelve el carácter correspondiente al valor “num” en código ASCII.
ord(‘x’)	Devuelve el código ASCII del carácter ‘x’.

27. Métodos

Listado de métodos de interés.

Función	Descripción
cadena.lower()	Devuelve la cadena de caracteres o variable en letras minúsculas.
cadena.upper()	Devuelve la cadena de caracteres o variable en letras mayúsculas.

Tabla de contenido

28. Materiales referencia

Para la elaboración de estos apuntes se han tomado como referencia estos estupendos materiales:

- Introducción a la programación con Python por Bartolomé Sintes Marco
- https://librosweb.es/libro/algoritmos_python/

Diccionarios:

- http://librosweb.es/libro/algoritmos_python/capitulo_9.html

Gestión de archivos:

- http://www.python-course.eu/python3_file_management.php

Excepciones y programación orientada a objetos:

- Curso Python desde cero. Creado por Juan Díaz, del sitio web pildorasinformaticas:
- https://www.youtube.com/watch?v=G2FCfQj-9ig&list=PLU8oAlHdN5BlvPxziopYZRd55pdqFwkeS

Borrar la pantalla en Python:

- https://unipython.com/como-borrar-pantalla-en-python/