

Bloque 1: Programación

Programando en Python



IES Fco. Grande Covián
TIC II - VERSION 6.0.0

1. Introducción

Un ordenador es, en esencia, un sistema capaz de realizar cálculos y tomar decisiones lógicas a una velocidad increíblemente alta.

Dentro de estos sistemas distinguimos una parte física, **hardware**, formada por todos los componentes que "se pueden tocar" y una parte lógica, **software**, compuesta por todos los datos que la máquina es capaz de **procesar** y **almacenar**.

El ordenador procesará (utilizará) estos datos según el control realizado por una secuencia de instrucciones predefinidas a las que llamaremos **programas**.

Programa informático: "texto" que incluye las instrucciones que un ordenador va a necesitar ejecutar para poder resolver un problema.

Al igual que podríamos utilizar distintos lenguajes (inglés, español...) para escribir un texto tradicional, existen muchos lenguajes de programación diferentes.

El objetivo de esta unidad es aprender a utilizar el lenguaje Python para crear programas que resuelvan situaciones concretas.

2. Algoritmos

Algoritmo: Serie de pasos a seguir para realizar una determinada tarea técnica.

Un algoritmo se parece a una receta. Una receta dice cómo preparar un plato siguiendo una serie de pasos. Por ejemplo, para preparar un pastel los pasos a seguir serían: Precalentar el horno, mezclar la harina con el azúcar y los huevos, poner la mezcla en un molde... y así sucesivamente.

Sin embargo, el término algoritmo es un término técnico más específico que una simple receta. Para que una “receta” pueda llegar a ser llamada algoritmo ha de cumplir las siguientes condiciones:

1. **Descripción no ambigua** que define que debe hacerse en cada paso. En una receta, un paso del tipo “cocinar hasta que esté hecho” es ambiguo, no deja claro que quiere decir “hecho”. Una descripción más objetiva como “cocinar hasta que el queso comience a burbujear” es más adecuada.
2. Un algoritmo ha de **definir el tipo de entradas**, tipo de datos que se van a suministrar al programa. Por ejemplo, se puede requerir dos números y que ambos sean mayores que cero. O puede requerir una palabra, o una lista de tres números.
3. Un algoritmo generará un conjunto de **salidas con características definidas**. Por ejemplo, puede generar el número que sea mayor de dos entradas, la versión en mayúsculas de una palabra, una lista ordenada de números...
4. El algoritmo siempre ha de **terminar** y producir un **resultado** en un tiempo finito.
5. Un algoritmo ha de ser capaz de generar el **resultado correcto en cualquier situación**. No sería útil tener un algoritmo que resolviera un problema en el 99% de casos posibles.
6. Si un algoritmo impone una condición a las entradas (precondición), esa condición se debe satisfacer. Por ejemplo, una precondición podría ser que el algoritmo solamente aceptara números positivos como entradas. Si las precondiciones no se cumplen sería aceptable que el algoritmo produjera una respuesta incorrecta o que nunca llegara a terminar.

Crear el algoritmo que resuelva el problema es un paso básico en el proceso de programación.

Ejemplo: Dada una lista de números positivos, devuelve el número mayor de la lista.

- **Entradas (Inputs):** Una lista L de números positivos, debe contener al menos un número.
- **Salidas (outputs):** Una “caja” que llamaremos max, almacenará el número mayor de la lista.
- **Algoritmo:**
 - o Asignar a max el valor 0.
 - o Para cada número x en la lista L, comparar su valor con max. Si x es mayor que max, asignar a max el valor x.
 - o Mostrar el valor almacenado en max que contiene el valor máximo en la lista.

¿Satisface esta “receta” los criterios para ser considerada un algoritmo?

1. ¿Es no ambigua? Sí. Cada paso es una operación sencilla fácilmente traducible a un código de programación (Python en nuestro caso).
2. ¿Se definen las entradas? Sí
3. ¿Se definen las salidas? Sí.
4. ¿Está garantizada su finalización? Sí. La lista L tiene un tamaño finito, por lo tanto, tras revisar cada elemento de la lista el algoritmo se detendrá.
5. ¿Produce el resultado correcto? Si. En un entorno más formal habría que probar su corrección de forma más rigurosa.

3. Diagramas de flujo

La representación de algoritmos a través de bloques de texto es tediosa y difícil de interpretar por una persona distinta de la que lo ha elaborado. Normalmente utilizaremos los llamados:

Diagrama de flujo: Representación gráfica mediante símbolos de un algoritmo.

Dentro de los símbolos se escribirán los pasos a seguir en el proceso.

Los diagramas de flujo, son herramientas importantes ya que permiten crear la estructura del programa sin necesidad de dominar un lenguaje de programación.

La creación del diagrama es la primera fase del desarrollo del programa; previa a la codificación.

Un diagrama de flujo permitirá que otros puedan comprender fácilmente nuestro algoritmo.

La única desventaja que supone utilizar diagramas de flujo es el espacio necesario para su desarrollo. El orden es sumamente importante, las hojas deben ser numeradas y señaladas.

Actualmente contamos con software diseñado para la elaboración de diagramas de flujo y herramientas *online* que permiten hacer el trabajo de forma más fácil y rápida.

Software instalado en los ordenadores de la sala de informática [yEd graph Editor](#).

3.1 Principales elementos en un Diagrama de Flujo simple

Terminal

Se usa para señalar el inicio y término del diagrama de flujo.



Inicio / Fin

Datos

Sirve para pedir que se ingrese un valor a la variable de **entrada** (input). También es usado para mostrar el resultado o **salida** (output). Para saber si se trata de una entrada o salida, comenzamos con la palabra 'leer' o 'escribir' respectivamente.



Entrada / Salida de Datos

Proceso

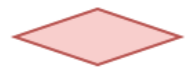
Es usado para ejecutar una orden. Por ejemplo, para hacer una asignación, calcular una operación, etc.



Proceso

Decisión

Es una condicional, sirve para tomar un camino según el resultado de una **expresión lógica**. Como resultado dividirá la secuencia en dos, una para el resultado positivo y otra para el negativo.



Condicional

Preparación

Proceso Predefinido

Llama a una sub-rutina para ejecutarse, de tal forma que ya no es necesario volver a crear su diagrama. Por ejemplo si hemos creado un diagrama para ordenar números podemos volver a llamarla, lo cual lo hace un elemento muy práctico.



Líneas de flujo

Las flechas indican la dirección y sentido que toma el flujo de ejecución



Flecha

Anotación

Nota que explica algún paso difícil de comprender en su funcionamiento o propósito. La figura es similar al signo de 'corchete' y el texto va en su interior.



Anotación

Entrada Manual

Se usa para solicitar la entrada de datos por teclado. Forma de trapecioide-rectángulo, similar a la forma que tiene un teclado:



Entrada de Datos Manual

Documentación

Indica la presencia de un documento a utilizar en el proceso, ya sea porque se genere, utilice o **salga del proceso**:



Documento

Base de Datos

Indica una lista de información con una estructura estándar que permite operaciones de búsqueda y ordenamiento.



Base de datos

3.2 Reglas para la elaboración de un Diagrama de Flujo

1. Todo Diagrama de Flujo debe tener principio y fin.
2. Todos los símbolos han de estar conectados
3. A un símbolo de proceso pueden llegarle varias líneas
4. A un símbolo de decisión pueden llegarle varias líneas, pero sólo saldrán dos (Si o No, Verdadero o Falso).

5. A un símbolo de inicio nunca le llegan líneas.
6. De un símbolo de fin no parte ninguna línea.
7. Las líneas de conexión son rectas (Verticales u Horizontales), nunca oblicuas o cruzadas.
8. Se deben dibujar todos los símbolos, siguiendo un proceso de arriba hasta abajo. Y de izquierda a derecha.
9. Evitar términos que se asemejen a algún lenguaje de programación.
10. Utilizar comentarios al margen, para que sean entendibles por cualquiera que los lea.
11. A cada bloque se accede por arriba y/o izquierda y se sale por abajo y/o derecha.
12. Si el diagrama abarca más de una hoja es conveniente enumerarlo e identificar de donde viene y a donde se dirige.

Ventajas

- Favorece la comprensión del algoritmo, nuestro cerebro reconoce de manera fácil los dibujos.
- Permite identificar problemas, redundancias, conflictos, cuellos de botella, entre otros.
- Útil herramienta de aprendizaje para quienes se inician en el lenguaje de programación.

Desventajas

- A medida que crece la complejidad de las proposiciones, crece el detalle de la diagramación, haciendo que sea más difícil de comprender y de seguir.

3.3 Como hacer un diagrama de flujo

Normalmente para realizar un diagrama de flujo primero crearemos su algoritmo.

Ejemplo (simplificado) de algoritmo y diagrama de flujo para cocinar un huevo para otra persona sería:

- Pregunto si quiere el huevo frito.
- Si me dice que sí, lo frito, si me dice que no, lo hago hervido.
- Una vez cocinado le pregunto si quiere sal en el huevo.
- Si me dice que no lo sirvo en el plato. Si me dice que si le hecho sal y después lo sirvo en el plato.

Una vez que ya tenemos un algoritmo con todos los pasos, haremos un esquema con los pasos a seguir. Este esquema será el Diagrama de Flujo.

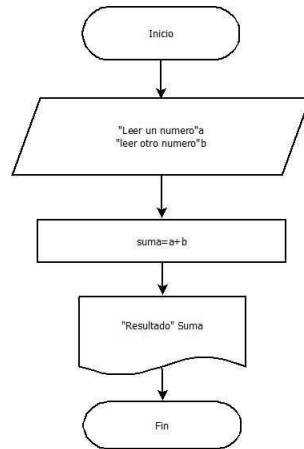


3.4 Ejemplos de Diagramas de Flujo

Programa que suma dos números y da el resultado por pantalla.

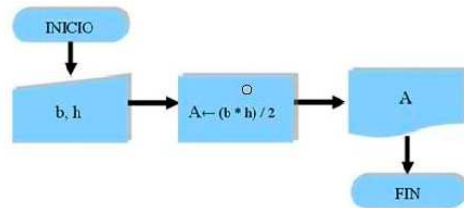
El símbolo de resultado es un símbolo usado en los diagramas para soluciones con el ordenador. Es el símbolo de salida del resultado por la pantalla del ordenador.

En el ejercicio tenemos el inicio y el fin, una entrada de datos, para introducir los 2 números, una operación a realizar, la suma, y un resultado a mostrar. Cada uno de esos pasos con su símbolo correspondiente en el diagrama.



Mostrar el resultado del área de un triángulo por pantalla.

En este caso ni siquiera hemos puesto las operaciones dentro de los símbolos de entrada y de salida, ya que con la forma del símbolo ya se entiende.



Determinar si un número es para o impar

En este caso tenemos dos posibles salidas, utilizaremos el símbolo de tomar una decisión y la operación mod, que devuelve el resto de una división:

¿Al dividirlo entre 2 el resto es 0? Hay 2 posibilidades. Si lo es se ve en pantalla "Si es par", si no lo es se ve en pantalla "No es par". Eso es la toma de decisiones.

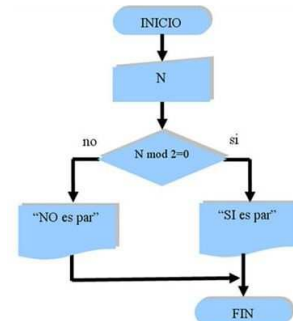
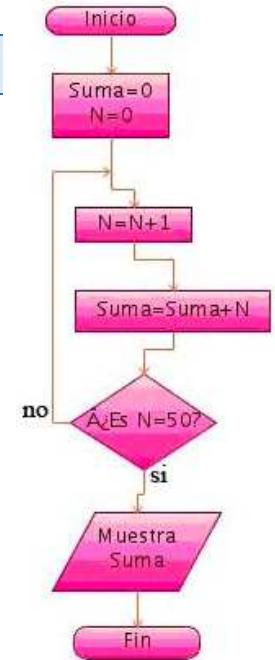


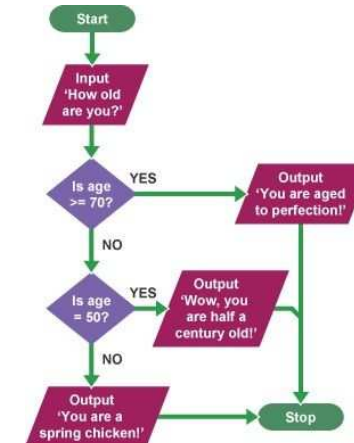
Diagrama de flujo para mostrar la suma de los 50 primeros números.

Lo primero es poner a cero la suma y dar el primer número a sumar que será el 0.

Fíjate que el diagrama acaba cuando N, que es el número que sumamos en cada momento, es 50. Mientras no sea 50 el programa vuelve a la tercera secuencia que será sumarle un número al anterior $N = N + 1$. Intenta comprenderlo y ver que hace. Puedes realizar mentalmente el diagrama para el número 0 y verás cómo lo acabas entendiendo.



Interpreta el siguiente diagrama de flujo.



4. Tipos de lenguajes de programación

Los programadores pueden escribir instrucciones utilizando varios tipos de lenguajes de programación, algunos son directamente entendibles por los ordenadores y otros requieren un proceso intermedio de traducción al lenguaje que pueden entender las máquinas. En la actualidad existen cientos de lenguajes diferentes. Todos ellos se pueden clasificar en tres categorías:

1. Código máquina.
2. Lenguaje ensamblador
3. Lenguajes de alto nivel.

Código máquina

Cualquier ordenador puede entender solamente su propio lenguaje al que llamaremos código máquina. Este código habrá sido definido para su exclusivo diseño de hardware.

El código máquina consiste normalmente de una cadena de números (formado por 1s y 0s) que le dice al ordenador como realizar las operaciones más básicas en forma sucesiva. Este tipo de código varía en función de la máquina (para cada tipo de computador se desarrolla un código específico).

Este tipo de lenguaje es ininteligible para los seres humanos y utilizarlo como lenguaje de programación resulta sumamente tedioso.

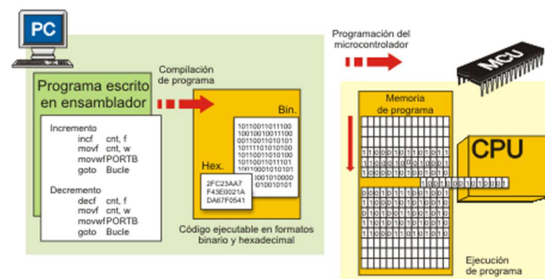
Lenguaje ensamblador

Los primeros computadores se programaban utilizando código máquina, este proceso era demasiado lento y tedioso para la mayoría de los programadores.

En lugar de utilizar una cadena de números directamente entendible por el ordenador, los programadores comenzaron a utilizar abreviaturas y palabras en inglés asociadas a operaciones básicas.

Estas abreviaturas formaron la base de lo que se conoce como lenguajes ensamblador. Se obtiene así un código mucho más entendible para los humanos, pero que sin embargo es incomprensible para los ordenadores hasta que es traducido a código máquina.

Utilizando programas traductores intermedios llamados ensambladores era posible convertir este código en código máquina aprovechando la gran velocidad que para ello ofrece el ordenador y facilitando con ello la tarea del programador.



Lenguajes de alto nivel y compiladores

Con la aparición de los lenguajes ensamblador, el uso de los ordenadores aumentó rápidamente, pero los programadores todavía tenían que utilizar una gran cantidad de instrucciones para realizar incluso las tareas más sencillas. Para acelerar el proceso de programación se desarrollaron los lenguajes de alto nivel.

Estos lenguajes utilizan sentencias individuales sencillas que permiten realizar al ordenador tareas complejas. Cada sentencia en lenguaje de alto nivel es la suma de muchas órdenes en lenguaje ensamblador.

Los lenguajes de alto nivel permiten escribir órdenes muy parecidas a una frase escrita en inglés corriente, utilizando además una notación matemática ordinaria.

Lenguaje de Alto Nivel C	Lenguaje Ensamblador
<pre> SWITCH TYPE) { case 'a': type=type+10; break; case 'b': type= type+20; break; default: break;} </pre>	<pre> MOV1 R1 = Type LD4 R2 = [R1] ;; cmp.eq P1, P2 = 'a', R2 cmp.eq P3, P4 = 'b', R2 ;; (P1) Add R2 = 10, R2 (P3) Add R2 = 20, R2 ;; st4 (R1) = R2 default:: </pre>

Existen dos tipos de programas traductores, los llamados **intérpretes** y los **compiladores** que transforman el código escrito utilizando un lenguaje de alto nivel en código máquina.

La diferencia básica entre un lenguaje que utiliza intérprete y otro que utiliza compilador es:

- El **intérprete** traduce a código máquina cada orden que se va a ejecutar cada vez que se ejecuta.
- Un **compilador** transforma el código inicial en un archivo nuevo escrito en código máquina que será entendible por el ordenador.

Un lenguaje interpretado será pues más lento que uno compilado, pues el proceso de traducción se tiene que ejecutar continuamente. Sin embargo, el lenguaje compilado tendrá que ser traducido completamente a código máquina cada vez que hagamos una modificación, en el caso de los programas complejos esto puede llevar un tiempo considerable.

Existen infinidad de lenguajes de alto nivel. Este curso vamos a centrarnos en lenguaje Python. La pregunta sería ¿Por qué Python?

4.1 ¿Por qué Python?

Existen muchos lenguajes de programación. Hemos elegido Python por varias razones:

Calidad del software: El código Python es fácil de entender, incluso si la persona que lo interpreta no es la que lo ha escrito. Esto hace que los bloques de código sean fáciles de reutilizar y mantener.

Portabilidad: La mayoría de los programas escritos en Python funcionan sin cambios en cualquiera de los principales tipos de ordenadores.

Librerías de ayuda: Python viene con una gran colección de funcionalidades prediseñadas y portables, conocidas como librería estándar. Esta librería consta de una colección acciones preprogramadas capaces de realizar múltiples tareas. Esta biblioteca puede extenderse utilizando librerías personales o librerías disponibles de forma libre en la red.

Integración de componentes: Los bloques de código Python pueden comunicarse fácilmente con otros bloques de otras aplicaciones y lenguajes.

5. Instalación Python

Para poder ejecutar en un ordenador un programa escrito utilizando el lenguaje Python es necesario instalar en dicha máquina un software al que llamaremos **intérprete Python**.

Interprete Python: Paquete de software que ejecuta el código incluido en el archivo Python.

Es una aplicación que lee el código y ejecuta las órdenes contenidas en el archivo de forma secuencial, comenzando con la primera orden y bajando progresivamente hasta llegar a la última.

Es una capa de software intermedia entre el código original y el hardware sobre el que va actuar.

La instalación de Python supone la instalación una serie de elementos, entre ellos y por lo menos tendremos el intérprete de Python y una librería de soporte.

La forma de instalar Python depende del sistema operativo empleado. Las diferentes versiones se pueden descargar desde: <https://www.python.org/>

5.1 Instalación en Windows 7-10

Es posible que ya tengamos instalada una versión de Python. Para comprobarlo es suficiente con abrir el panel de control y seleccionar la opción “Desinstalar o cambiar un programa” y comprobar si existe una versión instalada (repasando la lista de programas instalados).

Actualmente existen dos tipos de versiones de Python, las 2.x y las 3.x. Se recomienda la instalación de las versiones 2.x cuando estemos utilizando código antiguo que pudiera presentar algún problema de compatibilidad con la versión 3.x. En entornos como el nuestro en que vamos a programar código desde cero se recomienda instalar la versión más reciente de la familia 3.x.

En este caso se va a instalar la versión más reciente para Windows de 64 bits. Desde el sitio web python.org buscamos y descargamos la última versión.

Una vez descargado hacemos doble clic en el icono del archivo de instalación. Se mostrará:



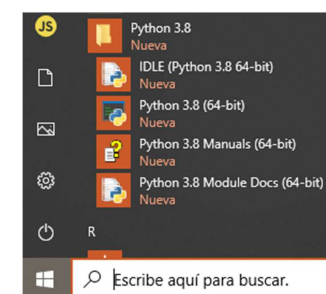
Activamos la opción **AddPython 3.8 to PATH**. Esto permitirá al sistema operativo encontrar Python desde la consola. A continuación, hacemos clic en el botón Install Now, iniciando el proceso de instalación.

Al terminar la instalación se mostrará la ventana:

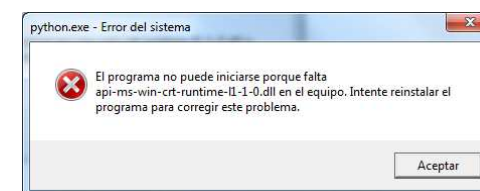


Hacemos click en el botón “Disable path length limit”.

Podremos comprobar cómo se ha creado una carpeta en el menú de inicio y desde allí se puede acceder al programa intérprete (en este caso desde el elemento Python 3.8 (64-bit)).



En algunos equipos, al intentar ejecutar Python se generará el error:



Este archivo se instala con las actualizaciones de Windows. También se puede obtener a través del archivo que instala Visual C++ Redistributable de Microsoft (<https://www.microsoft.com/es-ES/download/details.aspx?id=48145>).

6. Ejecutar un programa Python

¿Qué ocurre cuando el intérprete Python ejecuta un archivo de código?

Desde el punto de vista del programador.

La forma más sencilla de programa Python consiste en crear un archivo de texto con órdenes Python, lo que llamaremos código fuente.

Código fuente: Conjunto de instrucciones que le dicen al ordenador lo que tiene que hacer.

Normalmente el archivo con el código fuente tendrá extensión .py

Ej.:

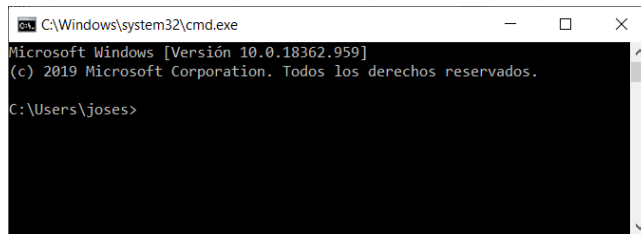
```
print("hola")
print(2**100)
```

Para escribir el código fuente se puede utilizar cualquier editor de texto.

Tras escribir el código lo guardaremos en un archivo.py. Deberás recordar la ubicación donde has guardado el fichero, por ejemplo, la raíz del disco duro D:.

Ahora habrá que decirle al intérprete que lo ejecute (que corra cada una de las órdenes del archivo de arriba hacia abajo una tras otra).

Para ejecutar el archivo, abriremos una ventana de intérprete de comandos (CMD). Para ello es suficiente con teclear la combinación de teclas Windows+R y teclear CMD en la ventana emergente. Se mostrará la ventana del intérprete de comandos (en lugar de joses aparecerá el nombre del usuario activo).



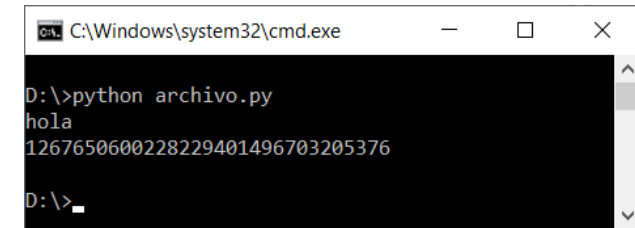
Siguiendo los pasos que se ven en la siguiente imagen nos dirigimos al lugar donde hayamos guardado nuestro archivo.py:



Para ejecutar el archivo escribimos en el terminal:

```
python nombreArchivo.py
```

Mostrará:



También es posible ejecutar el programa haciendo doble clic sobre su icono (parecerá que no, ya que se cierra al terminar la ejecución) o a través del IDE que estemos utilizando (esta opción la veremos más adelante).

Desde el punto de vista del intérprete de Python

¿Qué es lo que está ocurriendo a nivel interno? Al ejecutar la aplicación suceden varias cosas que el programador no observa:

1. Compila el código fuente en algo llamado "bytecode".

Compilar: Proceso de traducción mediante el cual transformamos la información contenida en el archivo .py, dividiéndola en procesos más sencillos y traduciéndola a un idioma de más bajo nivel (más cercano a lo que el ordenador puede entender directamente).

Estos bytecodes son multiplataforma y por lo tanto portables y capaces de funcionar en cualquier ordenador que tenga instalada la versión de Python en la que el bytecode ha sido compilado.

Si el proceso Python tiene acceso al proceso de escritura en el ordenador guardará el archivo con una extensión .pyc (py compilado). Estos archivos se guardarán en un subdirectorio llamado _pycache_ situado en el directorio donde reside el .py y con un nombre que también indicará la versión con la que fue creado (ej. nombreArchivo.cpython-36.pyc).

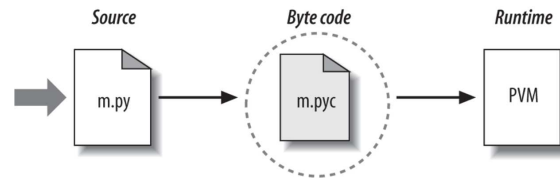
Una vez creado el pyc, cuando volvamos a ejecutar el programa, Python saltará el proceso de compilación y por lo tanto la ejecución será mucho más rápida.

En caso de que el intérprete detecte que ha cambiado algo en código fuente o que la versión de Python que intenta ejecutar la aplicación sea diferente se volvería a ejecutar la compilación.

Si Python no tiene acceso al proceso de escritura en el ordenador, el proceso de compilación se tendrá que ejecutar cada vez que ejecutemos el código fuente, ralentizando el proceso. Siempre que sea posible es aconsejable trabajar de la primera forma.

Si trabajamos desde el terminal, nunca podremos generar el archivo .pyc.

2. Enviamos el bytecode a una "máquina virtual". Una vez compilado y cargado en memoria el bytecode es enviado para su ejecución a una aplicación llamada Máquina virtual de Python (PVM).



Máquina virtual de Python: Parte del intérprete de Python capaz de ejecutar estos bytecodes. Software que simula el funcionamiento de un ordenador prescindiendo de las características específicas de cada sistema operativo y configuración de hardware.

Estamos hablando pues de un proceso intermedio entre un traductor (lenguajes como basic) que tendría que compilar completamente cada línea de código cada vez que se ejecuta en el programa y un lenguaje compilado puro (C++) en el cual el proceso de compilación genera un archivo en un código máquina que entienda directamente **MI** ordenador. La velocidad es intermedia, pero la ventaja es que la PVM es capaz de tratar el bytecode y transformarlo en algo que **CUALQUIER** ordenador entenderá, adaptándolo a su propia arquitectura.

Tipos de Python

Existen múltiples distribuciones diferentes del lenguaje Python. Las tres más importantes son:

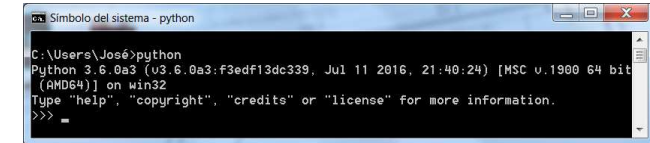
- CPython: Es el estándar, la llamamos CPython para contrastarla del resto, sino nos referiremos a ella como Python simplemente.
- Jython, para Java: Compila el código fuente en un bytecode propio del lenguaje Java y el resultado se ejecuta en una máquina virtual de Java (JVM). La integración es total, un programa en Jython funcionará aparentemente como una aplicación Java, el código podrá importar otras clases (programas) escritos en Java...
- IronPython, para .NET: Similar al anterior pero diseñado para trabajar con las aplicaciones .NET de Microsoft (y el equivalente Mono de Linux).

A lo largo de esta unidad trabajaremos con CPython.

7. Utilizar Python directamente desde la línea de comandos

Vamos a comenzar a trabajar con Python y lo haremos por la forma más simple, a través de la línea de comandos, de forma muy similar a como lo haríamos con una calculadora.

En Windows utilizaremos la consola y en Linux una ventana de Terminal. Basta con abrir la consola y teclear la orden Python. Se mostrará la ventana (Windows)



También podemos seleccionar la opción Python contenida en el menú del mismo nombre contenida en el menú de inicio.

En que veamos los iconos >>> quiere decir que estamos en una sesión interactiva del intérprete de Python y podemos ejecutar ordenes en Python.

Atajo: La forma más rápida de acceder al intérprete de Python es abrir el cuadro de diálogo ejecutar (tecleando TeclaWindows+R) y escribir en él la palabra python.

El hecho de que al teclear Python sin más el ordenador sea capaz de ejecutarlo se debe a que el sistema operativo es capaz de encontrar donde está instalado Python por haberlo incluido esta ruta en su path de búsqueda de programas cuando hemos realizado la instalación.

Si el ordenador no fuera capaz de encontrarlo (o hubiera instalada más de una versión y quisiéramos ejecutarla en lugar de la versión por defecto) habría que añadirlo en la llamada (por ejemplo c:\python27\python).

A esta forma de trabajar, escribiendo algo de código en la ventana y ejecutarlo directamente se le llama **crear código interactivamente**. Cada vez que se pulsa enter se muestra el resultado de la orden ejecutada (ni siquiera haría falta poner print)

```
>>>pera = 'okay'
>>>pera
'okay'
>>>2 ** 8
256
```

Esta forma de trabajo no permite guardar código, normalmente no la utilizaremos, pero permite: Experimentar con el lenguaje sin riesgo a estropear cosas que funcionan en el programa real.

Probar archivos ya programados al vuelo.

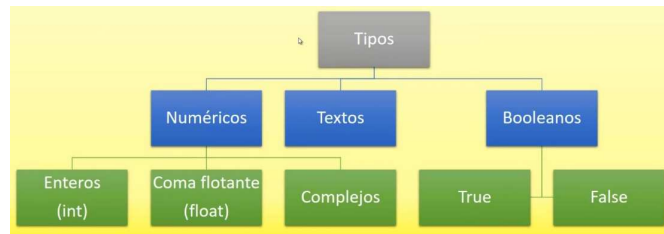
Cuando utilizamos una orden que requiere más de una línea de código (ordenes compuestas) es necesario hacer dos veces clic en Enter para ejecutar el programa.

Vamos a comenzar a utilizar algunos sencillos comandos de Python, para comprender su funcionamiento utilizaremos una sesión interactiva a través de la consola.

Salir de la sesión interactiva: Ejecuta la orden exit()

8. Tipos de datos

Python puede manejar los siguientes tipos de datos:



Numéricos: Pueden ser

- Enteros: tipo **int**
- Coma flotante: tipo **float**, números decimales
- Complejos no los veremos

Textos: Sera dato tipo texto cualquier información encerrada entre comillas (simples, dobles o triples (permiten utilizar saltos de línea dentro del dato tipo texto)).

Booleanos: Solo puede tomar dos valores True y False.

9. Operadores. Trabajando con números

Podemos utilizar la consola para realizar cálculos. Los operadores matemáticos en Python son:

Operador	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
División entera	//
Resto de la división	%
Potencia	**
Paréntesis	()
Redondear *	round(a,b)

*round(a,b) redondearía el número a con b decimales (ej. round(3.14,1) generaría 3.1).

Ojo: El punto decimal se indica por medio del símbolo punto (.) y no con la coma (,).

Una vez que pulsemos Enter, tras introducir la expresión matemática, Python mostrará dos nuevas líneas, la primera (sin el prompt>>>) contendrá el resultado y lasiguiente(que comenzará con el prompt>>>) quedará a la espera de que introduzcamos una nueva orden.

```

# Abre una sesión interactiva y realiza los siguientes cálculos:
>>> 2 + 2
>>> 50 - 5*6
>>> (50 - 5*6) / 4
>>> 8 / 5 # la división siempre retorna un número de punto flotante
>>> 17 // 3 # la división entera descarta la parte fraccional
>>> 17 % 3 # el operador % retorna el resto de la división
>>> 2 ** 7 # 2 a la potencia de 7
  
```

10. Comenzando a utilizar variables

Variable: Porción de la memoria del ordenador donde se puede almacenar un valor que más adelante podrá ser utilizado/modificado en el programa.

Cada variable queda referenciada a través de un **nombre de variable**. Al utilizar el nombre de variable dentro del código accederemos al valor que la variable tiene almacenado (los nombres no pueden comenzar por números ni tener espacios en blanco).

En otros lenguajes es común que al comienzo del programa haya que definir el nombre y el tipo de datos que va almacenar la variable. Esto no es así en Python, podremos crear una variable en cualquier punto del código.

Normas para dar nombres a las variables

- El nombre debe empezar por una letra o por un guion bajo (_).
- El nombre no puede tener espacios en blanco.
- Es recomendable evitar utilizar caracteres no ingleses (ñ, tildes...)
- Python distingue entre mayúsculas y minúsculas (casesensitive).

Interesa utilizar nombres de variables que tengan relación con el dato almacenado.

Por convenio la primera letra se escribe en minúsculas y la primera letra de cada palabra que forme parte del nombre tras ella con mayúscula (Ej.: miNombre)

El signo igual (=) se utiliza dentro del código para asignar un valor a una variable. Así la expresión `n = 5` haría que se creara una variable (por ser la primera vez que se utilizaría en el código) con el nombre `n` y almacenaría en ella el valor 5.

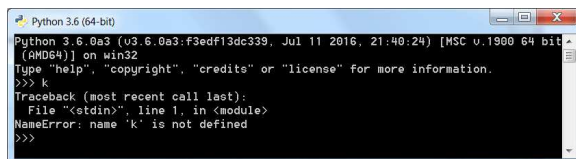
```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

En el código anterior la primera orden crea una variable a la que le asignaremos el nombre `ancho` y almacenará el valor 20. Observa que esta línea no genera un resultado por pantalla.

La siguiente línea de código crea la variable de nombre `largo` y se almacena en ella el resultado de multiplicar cinco por nueve.

La tercera línea de código realiza una operación (multiplicación) como en los ejemplos anteriores, pero observa que no se utilizan valores numéricos sino los nombres de las variables definidos anteriormente. La orden se leería "multiplica el valor almacenado en la variable `ancho` por el valor almacenado en la variable `largo` y muestra el resultado en una nueva línea".

En caso de que queramos mostrar el valor contenido dentro de una variable que todavía no ha sido definida obtendríamos un mensaje de error (la siguiente pantalla muestra el resultado de hacer una llamada a la variable `k` que todavía no ha sido definida).



El último valor generado en este modo interactivo se almacena en una variable de nombre `_` (barra baja).

Es una variable de solo lectura.

Esto es muy útil en ejemplos como el siguiente:

Calcula el precio final de un objeto si su precio antes de impuestos es 100.50€ y la tasa impositiva es el 12.5%

```
>>> impuesto = 12.5 / 100
>>> precio = 100.50
>>> precio * impuesto
12.5625
>>> precio + _
113.0625
>>> round(_, 2)
113.06
```

Más adelante hablaremos de los distintos tipos de valores que se pueden almacenar en una variable numérica.

11. Cadenas de caracteres – función print()

Cadena de caracteres: Cualquier información que Python interprete como un texto.

Python interpretará como cadena de caracteres cualquier elemento que esté contenido entre comillas simples ('...') o comillas dobles ("...").

Se utilizan dos tipos distintos de comillas para poder incluir dentro de nuestra cadena de caracteres cualquier de los dos iconos. Por ejemplo, si quisiera mostrar la palabra don't, al tener una comilla sencilla en su interior no podría crear una cadena de caracteres en la forma 'don't', en su lugar tendría que utilizar la forma "don't".

Existe otra forma de evitar el problema anterior, anteponer a la comilla o comilla doble interior a la cadena el icono \. Así podría indicar 'don\t' o "\hola\".

Si utilizamos el intérprete interactivo para definir una cadena de caracteres obtendremos:

```
>>> 'hola que tal'
'hola que tal'
```

Es decir, se mostrará la cadena junto con los iconos que la limitan. Normalmente nos interesará mostrar por pantalla solamente el contenido de la cadena. Para ello utilizaremos la función **print()**

print() Función que muestra por pantalla la información contenida entre los paréntesis, argumentos.

- La orden añade al final un salto de línea.
- Los argumentos podrán ser cadenas de caracteres y/o variables.

Volviendo al ejemplo anterior, el resultado sería:

```
>>>print('hola que tal')
hola que tal
```

En el caso de trabajar con variables:

```
>>>c='hola que tal'
>>>print('c')
c
>>>print(c)
hola que tal
```

11.1 print() con más de un argumento

La función print admite varios argumentos seguidos.

- Si los argumentos están separados por comas, print() los separa con un espacio en blanco:

```
>>>print('Hola ', 'adiós')
Hola adiós
```

- Si los argumentos se escriben uno a continuación de otro sin más, print() no añade un espacio en blanco entre ellos:

```
>>>print('Hola "adiós")
Holaadiós
```

Si alguno de los argumentos a mostrar es una variable, es necesario utilizar la coma o el signo + (concatenación) tal y como veremos al estudiar la función input().

11.2 Opciones sencillas cadenas de caracteres y print()

Esta serie de opciones permiten dar un cierto formato a la información mostrada por pantalla:

\n – Retorno de carro

El modificador \ seguido de la letra n, dentro de una cadena de caracteres se interpreta como un retorno de carro.

```
>>>print('C:\algún\nombre') # aquí \n significa nueva línea!  
C:\algún  
ombre
```

r - Crear cadenas crudas

Si añadimos una r antes de la primera comilla del print(), los caracteres \ dejan de ser especiales y se muestran como un símbolo más:

```
>>>print(r'C:\algún\nombre') # nota la r antes de la comilla  
C:\algún\nombre
```

\t - Tabulador

El modificador \t, dentro de una cadena de caracteres se interpreta como un tabulador.

```
print("1\t2\t3")  
1      2      3
```

end="" - Evitar el salto de línea al final de la función print()

Al final añadimos el argumento end="".

```
print('Hola ', end='')  
print('adiós')  
Holaadiós
```

Concatenar cadenas de caracteres

El símbolo + permite unir cadenas de caracteres. Especialmente útil cuando se quieren separar cadenas largas en partes.

No se pueden concatenar cadenas y números con el operador suma. En ese caso será necesario convertir los números a cadenas con la función str().

Repetir cadenas de caracteres

El símbolo * permite repetir cadenas de caracteres:

```
>>>print(2 * 'ho' + 'la')  
hohola
```

Ojo: Dos cadenas de caracteres yuxtapuestas son automáticamente consideradas como una sola cadena:

```
>>>print(2 * 'ho"la')  
holahola
```

len() – Longitud cadena:

Función que devuelve la longitud de una cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'  
>>>len(s)  
33
```

Líneas múltiples

Si dentro del print() definimos la cadena por medio de tres pares de comillas, los finales de línea incluidos en su interior son tenidos en cuenta, generando cadenas de más de una línea:

```
print("""  
Uso: algo [OPTIONS]  
-h Muestra el mensaje de uso  
-H nombrehost Nombre del host al cual conectarse  
""")
```

El código anterior genera la salida:

```
Uso: algo [OPTIONS]  
-h Muestra el mensaje de uso  
-H nombrehost Nombre del host al cual conectarse
```

11.3 Índices en las cadenas de texto

Cada carácter de una cadena de texto queda referenciado por medio de un índice asociado a su posición en la cadena. Al primer carácter se le asigna el índice 0, al segundo el 1, al tercero el 2....

Ojo: Esta opción solamente es válida para cadenas de texto. Si trabajamos con números es necesario transformarlos previamente en una cadena por medio de la función `str()`.

Esta característica es muy útil cuando la cadena de caracteres está guardada en una variable. Al añadir el índice entre corchetes al nombre de la variable, se mostrará el carácter asociado al índice:

```
>>>variable = 'melocotón'
>>>print(variable[0]) # caracter en la posición 0
m
>>>print(variable[5]) # caracter en la posición 5
c
```

También es posible empezar a contar los caracteres desde la derecha. Para ello el carácter final queda referido con el índice -1, el anterior con el -2 y así sucesivamente.

```
>>>variable = 'melocotón'
>>>print(variable[-1]) # caracter en la posición -1
n
>>>print(variable[-9]) # caracter en la posición -9
m
```

“rebanadas de texto”

Una rebanada de texto es una porción de una cadena de caracteres con más de un carácter.

Para definir una rebanada de texto utilizaremos la siguiente notación:

```
variable[a:b]
```

Donde `variable` es el nombre de la variable que contiene la cadena de caracteres, `a` es el índice del primer carácter incluido en la rebanada y `b` el índice el primer carácter excluido de la rebanada.

Sería algo parecido a un intervalo $[a,b)$ en matemáticas.

```
>>>variable = 'melocotón'
>>>print(variable[0:3]) #desde la posición 0 (incluida) hasta la 3 (excluida)
mel
```

Esta forma de definir los intervalos permite dividir cadenas en dos partes de manera sencilla:

```
>>>variable = 'melocotón'
>>>print(variable[:3])
mel
>>>print(variable[3:])
ocotón
```

Observa que si no definimos el índice inicial de la rebanada se toma por defecto el valor cero y si no definimos el índice final se toma la longitud total de la cadena.

12. Listas

Lista: Tipo de variable que se capaz de almacenar más de un valor. Cada uno de los valores queda definido por un subíndice.

La forma más sencilla de definir una lista es a través de un bloque de elementos agrupados entre corchetes y separados entre sí por medio de comas:

```
>>>cuadrados = [1, 4, 9, 16, 'palabra']
>>>cuadrados
[1, 4, 9, 16, 'palabra']
```

Cada elemento de la lista pasa a estar indexado de forma análoga a la vista en el punto anterior:

```
>>>cuadrados[0] # índices retornan un ítem
1
>>>cuadrados[-1]
'palabra'
>>>print(cuadrados[-1])
palabra
>>>cuadrados[-3:] # rebanadas retornan una nueva lista
[9, 16, 'palabra']
```

Modificar el valor almacenado en un elemento de la lista:

```
>>>cubos = [1, 8, 27, 65, 125] # hay algo mal aquí
>>>4 ** 3 # el cubo de 4 es 64, no 65!
64
>>>cubos[3] = 64 # reemplazar el valor incorrecto
>>>cubos
[1, 8, 27, 64, 125]
```

En el ejemplo anterior modificamos el valor almacenado en el cuarto elemento de la lista.

Modificar los valores almacenados en una rebanada

Podemos modificar, borrar parcialmente, borrar completamente...

Ojo: Recuerda que el intervalo está cerrado por la izquierda y abierto por la derecha.

```
>>>letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>>letras
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>># reemplazar algunos valores
>>>letras[2:5] = ['C', 'D', 'E']
>>>letras
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>># ahora borrarlas
>>>letras[2:5] = []
>>>letras
['a', 'b', 'f', 'g']
>>># borrar la lista reemplazando todos los elementos por una lista vacía
>>>letras[:] = []
>>>letras
[]
```

12.1 Opciones muy interesantes trabajando con listas

Añadir un nuevo elemento a la lista

El método `append()` permite añadir un nuevo elemento a la lista:

```
>>>cubos.append(216) # agregar el cubo de 6
>>>cubos.append(7 ** 3) # y el cubo de 7
13
>>>cubos
[1, 8, 27, 64, 125, 216, 343]
```

len():

En este caso la función `len` nos devuelve un valor indicando el número de elementos en la lista:

```
>>>letras = ['a', 'b', 'c', 'd']
>>>len(letras)
4
```

Crear listas a partir de otras listas

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Comprobar si un valor está en una lista

La operación `in` permite comprobar si un valor está incluido en una lista:

- Si el elemento está incluido en la lista `in` devuelve el valor lógico `True`.
- Si el elemento no está incluido en la lista `in` devuelve el valor lógico `False`.

La sintaxis de la operación es:

```
elemento in lista
```

Donde `elemento` es el elemento que queremos comprobar y `lista` el nombre de la lista en la que lo queremos buscar. Así por ejemplo:

```
>>> xs=[78455, 79211, 54988, 66540, 47890]
>>> 78 in xs
False
>>> 66540 in xs
True
```

13. Ejecutar un archivo .py

Existen varias formas diferentes de ejecutar un archivo `.py`. Veamos las más habituales:

13.1 Ejecutar el archivo desde la línea de comandos

Normalmente no ejecutaremos el código Python desde la línea de comandos. Es mucho más habitual escribir el código en archivos de texto `(.py)` que luego ejecutaremos. Utilizaremos la palabra `módulo` para referirnos a estos archivos.

Módulo: Cualquier archivo de texto que contiene ordenes en Python.

Una vez guardado el módulo podemos hacer que el intérprete de Python ejecute las órdenes contenidas en él, de arriba hacia abajo y de una en una, de muchas formas distintas (línea de comandos, doble clic en su icono, IDE...)

Muchas veces nos referiremos a los módulos como programas o scripts.

Para cerrar la sesión interactiva del interprete podemos utilizar la orden `exit()`, `Ctrl+Z` (Windows), `Ctrl+D` (Linux) o cerrar la ventana de comandos.

Veamos un primer ejemplo:

13.1.1 Nuestro primer programa

Abrimos el bloc de notas (valdría cualquier editor de texto) y escribimos el siguiente código:

```
# Un primer script en Python
import sys          # Carga un módulo de librería
print(sys.platform)
print(2 ** 100)      # Elevamos 2 a la potencia 100
x = 'Spam!'
print(x * 8)         # Repetimos una cadena de texto ocho veces
```

Guardaremos el archivo como `script1.py` y lo guardamos en nuestro directorio de trabajo (`D:\Code`).

En resumen, este código:

- Importa un módulo (otro programa incluido en las librerías de herramientas de Python) que muestra el nombre del tipo de plataforma que estamos utilizando.
- Ejecuta la función `print` para mostrar el resultado del módulo anterior.
- Muestra el resultado de elevar 2 a la 100
- Almacena la palabra `Spam!` en una variable `x` a la que llamaremos `x`
- Utiliza la función `print` para escribir ocho veces el valor de la variable `x`.
- Se han añadido comentarios. Todo aquello precedido por `#`.

Comentario: Línea/s de código que el compilador va a ignorar. Su única función es mejorar la legibilidad del programa.

El archivo podría llamarse simplemente script1 sin añadir la extensión .py. El compilador lo entendería igual, pero es muy aconsejable añadir la extensión .py por dos razones:

- 1: Si queremos importar este módulo desde otro, sólo será posible si está guardado con esa extensión.
- 2: El editor reconocerá que es un archivo Python y podrá utilizar propiedades como el coloreado sintáctico o la indentación automática.

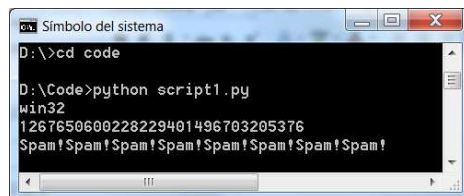
13.1.2 Ejecutar desde la línea de comandos

Abrimos la ventana de comandos.

Utilizamos las ordenes cd.. y dir hasta situarnos dentro del directorio que contiene el módulo (d:\Code)

Tecleamos la expresión python script1.py

Se obtendrá el resultado:



Una variación. Podemos utilizar las propiedades de la ventana de comandos, así si ejecutamos el programa según:

```
python script1.py > saveit.txt
```

El resultado se guardará en un archivo de nombre saveit.txt en el directorio de trabajo.

En Windows 7 podemos escribir directamente el nombre modulo (sin anteponer la palabra Python) ya que el SO reconoce la extensión .py

13.2 Ejecutar haciendo clic en el icono del módulo

Al hacer doble clic sobre el icono de un archivo, el sistema operativo lee la extensión del y lo abre con el programa apropiado para ese tipo de extensiones según se indique en el archivo de asociación de nombres de archivo.

Los archivos con extensión .py se ejecutarán por la aplicación py.exe a través de una consola.

Si ejecutamos el código anterior haciendo doble clic en Windows, veremos que el programa se ejecuta, pero inmediatamente se cierra la pantalla sin dejar a la vista el resultado.

Un truco sería llamar desde el código la función predefinida input(), la veremos más adelante:

```
# Un primer script en Python
import sys      # Carga un módulo de librería
print(sys.platform)
print(2 ** 100) # Elevamos 2 a la potencia 100
```

```
x = 'Spam!'
print(x * 8)      # Repetimos una cadena de texto ocho veces
input()           # <== ADDED
```

Input queda esperando la siguiente información que se introduzca por teclado. En este contexto lo utilizaremos como una pausa. Al pulsar la tecla Enter el programa terminará.

Ejecutar los programas por clic tiene muchas más limitaciones (por ejemplo detección de errores).

13.3 Interfaz de usuario IDLE

IDLE es una interfaz gráfica incluida dentro del sistema Python, que permite desarrollar código Python de un forma más “agradable”.

Para iniciar IDLE vamos al menú de inicio y dentro de la carpeta de Python 3.6 seleccionamos el elemento IDLE (64-bits). Se mostrará una ventana con fondo blanco que ejecuta una sesión interactiva de Python (>>>) similar a la vista en puntos anteriores.

En resumen IDLE permite:

Gestionar módulos: A través de los menús habituales permite abrir, crear, guardar, guardar como... módulos.

Editar texto utilizando “syntax-directed colorization”: Se utiliza distintos colores para las palabras clave, cadenas de textos... Esto ayuda a interpretar el código cuando el programa se va haciendo más grande.

Ejecutar el código: Utilizando la opción run, run module se ejecuta el código mostrando el resultado del programa.

Autoindentado: Mover el texto a la derecha (tab) o izquierda (del) para crear bloques de código más legibles.

Autocompletado de palabras: Si mientras estamos escribiendo una palabra pulsamos el tabulador se mostrará una lista con todas las opciones posibles para completar la expresión.

Globos de ayuda: Al escribir el (de una llamada de función se mostrará un globo con información sobre las opciones de dicha función).

Pop-up lista de selección de los atributos de un objeto cuando tecleamos “.” Después del nombre del objeto.

Depurar el código del programa: Primero seleccionamos Debug, Debugger en la ventana principal y luego ejecutamos el programa con Run, Run Module. Veremos las opciones de depurado más adelante.

Al hacer clic con el botón derecho sobre un mensaje de error nos mostrará información sobre el mismo.

Al utilizar IDLE es obligatorio añadir la extensión .py al módulo

13.4 Eclipse y PyDev

Aplicación similar a la anterior pero mucho más avanzada. Originalmente creada como herramienta de desarrollo para Java. Al instalar el plug-in PyDev permite trabajar con Python obteniendo una herramienta muy superior a IDLE y por ello será el IDE que utilizemos a lo largo del curso.

Al final de los apuntes se incluye un anexo en el que se explica cómo instalar Eclipse en Windows.

13.5 Sublime text

Otro editor de texto preparado para programar en muchos lenguajes, entre ellos Python. Su gran ventaja es su poco peso que lo hace ideal para ser utilizado en equipos no muy potentes. Su versión sin licencia puede ser utilizada de forma indefinida. Puedes descargar el programa desde el enlace:

<https://www.sublimetext.com/3>

13.6 PyCharm

PyCharm es un entorno de desarrollo integrado (IDE) utilizado en programación y específicamente para el lenguaje Python. Está desarrollado por la empresa checa JetBrains.

PyCharm es multiplataforma, con versiones de Windows, macOS y Linux. Existe una edición “Community Edition” gratuita y también hay una edición profesional con características adicionales, lanzada bajo una licencia propietaria.

14. Programar con Eclipse. Primer programa

En este apartado vamos a crear nuestro primer proyecto. Es algo sencillo, crear un módulo que muestre un mensaje por pantalla. Es algo que ya sabemos hacer utilizando una sesión interactiva. En este punto vamos a aprender a crear lo que llamaremos un proyecto utilizando Eclipse.

Vamos por partes:

14.1 Pasos previos 1: Crear un proyecto

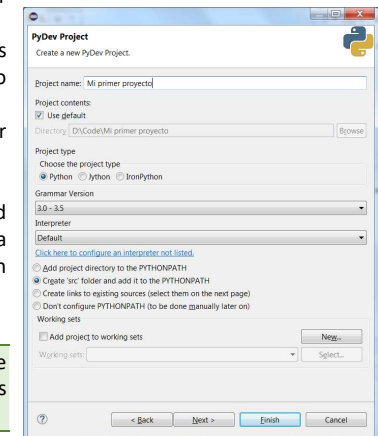
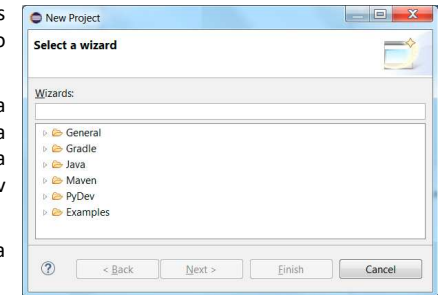
Cuando trabajamos con un IDE tipo Eclipse los programas con el código fuente se guardarán dentro de lo que llamaremos “proyecto”.

Así el primer paso va a ser crear ese proyecto. Para ello vamos a **File>New>New Project...** Se muestra la ventana New Project en la cual seleccionaremos la carpeta PyDev y dentro de ella el elemento pyDev Project. A continuación, clic en el botón Next.

Se mostrará la ventana PyDev Project. En ella definiremos las características del proyecto a crear:

- Nombre del proyecto (por ejemplo: Mi primer proyecto).
- Ubicación donde se va a guardar el código fuente. Es recomendable crear una carpeta dentro del directorio D:\Code con el nombre del proyecto.
- Versión gramática de Python que vamos a utilizar (3.6).
- Tipo de intérprete a utilizar, dejamos Default.
- Seleccionamos la opción “Create ‘src’ folder and add it to the PYTHONPATH”. Con ello se creará una carpeta llamada src (sourcecode) donde se guardarán todos los módulos de nuestro proyecto.
- Terminamos pulsando *Finish*.

Si entramos ahora dentro de la carpeta que contiene nuestro proyecto veremos que se han creado dos archivos .project y .pydevproject. y la carpeta src.



Volviendo a Eclipse, observa que la ventana del Package Explorer (zona izquierda) contiene una carpeta con el nombre del proyecto y dentro de ella dos elementos. El primero, una carpeta llamada **src**, contendrá los archivos con código fuente a crear. El segundo permite acceder a las bibliotecas de Python.

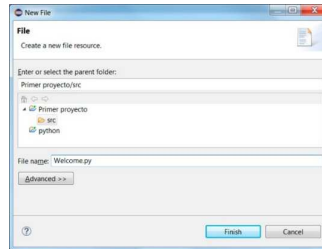
Esta ventana, package explorer, mostrará todos los proyectos creados con Eclipse.

14.2 Pasos previos 2: Crear un nuevo archivo de código fuente

Desde **File>New>File** creamos un archivo nuevo.

Aparece la ventana New File, seleccionamos la carpeta src del proyecto y asignamos un nombre al archivo sin olvidarnos de la extensión .py. (por ejemplo welcome.py)

Otra forma más sencilla: clic con el botón derecho encima del icono de la carpeta src y elegir New>File.



14.3 Comentando tus programas

Comentario: Línea/s de código que el compilador va a ignorar. Su única función es mejorar la legibilidad del programa.

La forma de indicar al interprete el comienzo de un comentario es a través del símbolo #. A partir de ese instante, y hasta que comience una nueva línea de código, el compilador ignorará la información y a efectos prácticos no tendrá ningún valor.

Consejos:

- Iniciar el programa con un comentario que indique la finalidad del programa, el autor, fecha y hora de la última modificación.
- Mantener actualizados los comentarios
- Incluir líneas en blanco para aumentar la legibilidad del código.

Ejercicio: Añade al archivo welcome.py los comentarios indicados en el punto anterior

14.4 Código que genera el texto por pantalla

Utilizamos la función print para escribir por pantalla una cadena de caracteres

```
print('Bienvenido a la programación con Python')
```

Este conjunto recibe el nombre de **orden**. Este programa está formado por una única orden.

14.5 Compilar y ejecutar la aplicación

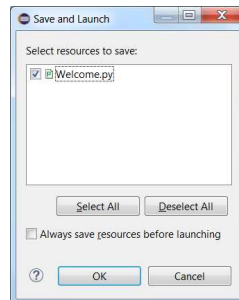
Haz clic en el menú superior en **Run>Run>Python run**. Se muestra la ventana "Save and Launch" que informa de los distintos módulos que se pueden grabar antes de realizar su compilación. Comprobamos que Welcome.py está seleccionado y hacemos clic en OK.

Una vez guardado el módulo, se ejecutará en la parte inferior de la ventana de Eclipse. Esta ventana se llama **Console**.

Podemos comprobar que el módulo se ha guardado en la ruta D:\Code\Mi primer proyecto\src\Welcome.py.

Este archivo podrá ser ejecutado desde la línea de comandos.

Ojo: Si lo ejecutamos desde el lanzador del menú de inicio parecerá no funcionar... espera hasta que veamos la función input()



15. Pasando valores al programa -input()

input() Función que permite introducir información a través del teclado

Cuando es necesario transmitir un valor al programa, Python ofrece la función input(). La forma más habitual de utilizar esta función será:

```
nombreVariable = input('mensaje')
```

Cuando el intérprete de Python llegue a una instrucción como la anterior, mostrará por pantalla el mensaje 'mensaje' y quedará esperando a que introduzcamos a través del teclado una cadena de caracteres o un valor. En el momento que pulsemos la tecla Enter, el valor introducido por teclado quedará almacenado en la variable nombreVariable y continuará con la ejecución de la siguiente orden.

Así, por ejemplo:

```
valor=input('Introduce un número: ')
print('El número introducido es '+valor+'!')
```

El significado de los símbolos + en la orden print se explica más adelante

Se mostrará un texto solicitando un número. Tras introducirlo se generará un mensaje a través de la función print uniendo dos cadenas de caracteres y el número introducido a través de la función input() que ha sido almacenado en la variable valor.

De forma similar se podría introducir una cadena de caracteres:

```
nombre = input("¿Cuál es tu nombre? ")
print("Hola, " + nombre + "!")
```

El resultado obtenido con los dos bloques de código anterior sería:

```
Introduce un número: 1
El número introducido es 1!
¿Cuál es tu nombre? Pedro
Hola, Pedro!
```

15.1 Números e input()

Por defecto input() convierte la respuesta en una cadena de caracteres, así si intentamos hacer un cálculo obtendríamos un mensaje de error. Si se quiere que Python interprete la entrada como un número habrá que utilizar una función para realizar la transformación:

Si queremos que la respuesta se tome como número entero: int()

```
valor=int(input('Introduce tu edad en años: '))
```

Si queremos que la respuesta se tome como un número decimal: float()

```
valor=float(input('Introduce el precio en euros: '))
```

Si intentamos guardar un número con decimales en una variable int se generará un mensaje de error. En cambio si almacenamos un número entero en una variable float no habrá problema. (ej: float(8)=8.0)

15.2 Argumento de la función input()

Esta función sólo admite un único argumento. Por tanto en el código:

```
nombre = input("Dime tu nombre: ")
apellido = input("Dime tu apellido, ", nombre, ": ")
print("Me alegro de conocerte,", nombre, apellido)
```

La primera línea es correcta, sin embargo, la segunda generará un error.

Estamos intentando generar un mensaje complejo formado por dos cadenas de texto y el contenido de una variable. La función input() sólo puede utilizar un argumento. Podemos resolver el problema dos formas:

Escribo la pregunta utilizando la función print() y en la orden siguiente utilizo input().

```
nombre = input("Dime tu nombre: ")
print("Dime tu apellido", nombre+": ")
apellido=input()
print("Me alegro de conocerte,", nombre, apellido)
```

Utilizar en la segunda orden input la concatenación:

Concatenación unir dos elementos (variables o cadenas de texto) en uno único.

Para concatenar dos elementos es escribirán uno a continuación de otro, separados por medio el símbolo de suma (+).

```
nombre = input("Dime tu nombre: ")
apellido = input("Dime tu apellido, "+ nombre+ ": ")
print("Me alegro de conocerte,", nombre, apellido)
```

Ejercicio: Crea un proyecto que sea capaz de leer dos números enteros que introduciremos a través del teclado, realizará la suma de los dos y mostrará el resultado por pantalla. Añade tantos comentarios como sea necesario para conseguir que el código sea fácilmente entendible.

La salida debe ser similar a:

```
Introduce el primer entero: 45
Introduce el Segundo entero: 72
La suma es: 117
```

16. Sentencias condicionales: if... elif...else...

16.1 Sentencia condicional if... (Una opción)

La **orden if** evalúa una condición. Si la condición se cumple se ejecutan las ordenes agrupadas dentro del bloque if. Si no se cumple no se ejecutarán las ordenes agrupadas en el bloque if.

La sintaxis de un bloque if es:

```
if condición:
    bloque de órdenes que se ejecutarán si la condición es cierta.
    Estas órdenes pueden ocupar más de una línea.
```

Condición	es una expresión lógica que el programa evaluará y que generará un valor cierto o falso.
:	Símbolo que marca el final de la expresión lógica. Es obligatoria.
Bloque de órdenes	Conjunto de órdenes que se ejecutan solamente si la condición lógica ha sido evaluada como verdadera.

El bloque de órdenes ha de ir indentado (sangrado). Python utiliza el sangrado para identificar las órdenes que están asociadas al bloque if.

Todas las órdenes asociadas al if han de tener el mismo sangrado.

Normalmente el sangrado en Python es de cuatro espacios. Al escribir el símbolo (:), Eclipse automáticamente realizará el sangrado.

16.1.1 Comparadores lógicos

Realizan una comparación entre dos expresiones. Cuando la expresión es cierta generan un valor booleano True, cuando no lo es generan un valor booleano False.

Símbolo	Comparador	Genera True	Genera False
>	Mayor que	3>2	2>3
<	Menor que	2<3	3<2
>=	Mayor o igual que	2>=1+1	2>=1+2
<=	Menor o igual que	4-2<=1	4-2<=2
==	Igual que	2==1+1	2==1
!=	Distinto de	6/2!=2	6/2!=3

16.1.2 Operadores lógicos

Son tres y permiten construir condiciones más complejas.

Símbolo	Operador	Genera True	Genera False
and	Y lógico	true and true	true and false false and false false and true
or	O lógico	true or true true or false false or true	false or false
not	Negación	not false	not true

16.1.3 Expresiones compuestas

Son aquellas que están formadas por varios componentes. Se recomienda utilizar paréntesis para asegurar el orden en que las expresiones son evaluadas.

Sin embargo, en Python no es obligatorio el uso de paréntesis. El orden de prioridad en este lenguaje es: primero se evalúan los not, luego los and y por último los or.

Utiliza paréntesis!!!!!!

Crea un módulo que pida al usuario un número positivo. La respuesta ha de ser almacenada en un variable a la que llamaremos numero. Comprueba si el número es negativo. Si lo es, el programa debe avisar del error. Finalmente el programa muestra siempre el número por pantalla.

16.2 if ... else ... (dos opciones)

En este caso el if permite que el programa ejecute un bloque de instrucciones cuando se cumple la condición incluida en la línea del if y otras cuando no se cumple.

La sintaxis de un bloque if...else... es:

```
if condición:
    bloque de órdenes que se ejecutarán si la condición es cierta.
else:
    Bloque de órdenes que se ejecutarán si la condición es falsa.
```

Los bloques de órdenes pueden tener más de una línea e irán indentados.

Todas las órdenes de un bloque han de tener el mismo sangrado, cada bloque puede tener un sangrado distinto... aunque eso reduce la claridad del código.

La línea con la orden else solo ha de incluir el else y los dos puntos.

Aunque no es muy habitual, si deseamos dejar uno de los bloques vacío, el bloque de instrucciones deberá contener la orden pass. Esta orden le dice a Python que no haga nada y salga del bloque de órdenes.

Diseña un programa que pregunte la edad al usuario y la almacene en la variable 'edad'. A continuación comprueba si la edad es inferior a 18 años. Si la condición es cierta el programa debe avisar por pantalla de que el usuario es menor de edad y si es falsa de que es mayor de edad. Finalmente el programa se despide en cualquier caso.

16.3 if... elif... else... (más de dos opciones)

Esta sentencia condicional permite encadenar varias condiciones. elif es la abreviatura de else if (si no).

La sintaxis de un bloque if...elif...else... es:

```
if condición1:
    bloque de órdenes que se ejecutarán si la condición1 es cierta.
elif condición2:
    bloque de órdenes que se ejecutará si la condición2 es cierta.
else:
    bloque de órdenes que se ejecutarán si todas las condiciones son falsas.
```

Se pueden incluir tantos bloques elif como sean necesarios.

Observa que en este tipo de sentencia las opciones son siempre mutuamente excluyentes. En cualquier caso solamente se ejecutará uno de los bloques.

En este tipo de estructuras hay que ser cuidadoso con el orden con que escribimos las condiciones. Con ello:

- Evitaremos olvidar alguna de las situaciones posibles.
- Evitaremos que el programa ejecute un bloque de instrucciones no deseado.

Veamos el siguiente ejemplo:

Modifica el programa anterior de tal forma que distinga tres situaciones. Si introducimos un valor negativo el programa mostrará un mensaje de error. Si el valor está comprendido entre 0 y 17, mostrará un mensaje indicando que el usuario es menor de edad y si es mayor o igual a 18 el mensaje informará de la mayoría de edad del usuario.

Existen múltiples soluciones sin embargo alguna de ellas nos puede dar resultados extraños como informar de que una persona que tenga -14 años sea menor de edad por ser el valor menor de 18. Investiga la mejor solución.

Complicaremos ahora las cosas un poco. Utilizando un único bloque if..elif..else diseña un programa que indique si un número es múltiplo de dos, de dos y de cuatro o si no es múltiplo de dos. El problema radica en que todos los múltiplos de cuatro son múltiplos de dos pero no todos los múltiplos de dos son múltiplos de cuatro.

16.4 Sentencias condicionales anidadas

Una sentencia anidada condicional puede contener a su vez otra sentencia anidada. Por ejemplo:

```
if condición1:
    if condición2:
        bloque de órdenes que se ejecutarán si la condición1 y la 2 son ciertas.
    else:
        Bloque de órdenes que se ejecutarán si la condición 1 es cierta y 2 es falsa.
else:
    Bloque de órdenes que se ejecutarán si la condición 1 es falsa.
```

En este caso se considera la primera condición. Si es true se considera la segunda condición y si también es cierta se ejecutan las ordenes contenidas en el primer bloque. Si la segunda no lo es se ejecuta el segundo bloque de órdenes. Solo en el caso de que la primera condición sea falsa e independientemente de cómo sea la segunda se ejecuta el tercer bloque de instrucciones.

Las estructuras anidadas pueden ser tan complicadas como se necesite y pueden contener cualquiera de los tres tipos de sentencias if que hemos estudiado.

Diseña un programa que haciendo sólo con dos preguntas al usuario sea capaz de adivinar el número entre 1 y 4 que este está pensando.

16.5 Sintaxis alternativa con valores lógicos

Observa el siguiente código:

```
color=True
talla=False
if (color==True):
    print("Ejemplo sencillo")
```

Hemos definido dos variables tipo booleano. Color toma el valor True y talla el valor False.

La condición incluida dentro del bucle if toma el valor True ya que se cumple la condición (el valor de la variable color es True).

La respuesta del programa es pues:

Ejemplo sencillo

Es decir, lo que hace que se ejecute el contenido incluido dentro del bloque if es el hecho de que el resultado que genera la comparación incluida en él es el valor True.

Eso se puede tener en cuenta de cara a definir un código similar pero más sencillo:

```
color=True
talla=False
if color:
    print("Ejemplo sencillo")
```

En este caso el bloque if analiza el valor de la variable color, como este es True el resultado será el mismo que en el caso anterior.

No ocurriría lo mismo con la variable talla, el código "completo" sería:

```
color=True
talla=False
if (talla==False):
    print("Ejemplo sencillo")
```

Que daría el mismo resultado que en los casos anteriores.

Sin embargo, si intentamos simplificar el código en la forma anterior:

```
color=True
talla=False
if talla:
    print("Ejemplo sencillo")
```

El programa no dará resultado, la línea if esta, recibe el valor False y no se cumple la condición.

Complicando un poco las cosas, supón que necesitamos escribir el código final si la variable color toma el valor True y la variable talla el valor False. El código "completo" sería:

```
color=True
talla=False
if ((color==True) and (talla==False)):
    print("Ejemplo sencillo")
```

Si queremos simplificarlo podríamos hacer:

```
color=True
talla=False
if (color and talla==False):
    print("Ejemplo sencillo")
```


17. El bucle for

En general un bucle en programación se definirá como:

Bucle: Estructura de control que repite un bloque de instrucciones.

En Python existen varios tipos de bucles. Un bucle for es:

Bucle for: Bucle que repite un bloque de instrucciones un número determinado de veces.

Llamaremos:

Iteración: Cada una de las veces que se ejecuta el cuerpo del bucle.

La sintaxis básica del bucle for es:

```
for variableControl in elemento de control  
    Cuerpo del bucle
```

Veamos la orden un poco más a fondo:

variableControl	Variable que tomará en cada iteración un valor definido por 'elemento de control'. La iteración se repetirá tantas veces como valores pueda tomar 'variableControl' y en cada iteración el valor almacenado en la variable es definido por el elemento de control.
Elemento de control	Elemento que define los distintos valores que va a ir tomando la variable 'variableControl'. Puede ser una lista, una cadena de caracteres, un rango.
Cuerpo del bucle	Está indentado y contiene el conjunto de órdenes que se ejecutarán en cada iteración.

Por costumbre se suele utilizar la letra i como nombre de la variable de control

Veamos varios ejemplos:

Bucle for controlado por una lista

En este caso:

- El bucle realizará tantas iteraciones como elementos tenga la lista.
- En cada iteración la variable irá tomando sucesivamente en cada iteración los valores contenidos en la lista.

```
print("Comienzo")  
for i in [1, 1, 1]:  
    print("Hola ", end="")  
    print()  
print("Final")
```

El for del código anterior se ejecutará tres veces, la variable i tomará el valor 1 en las tres iteraciones y como resultados se mostrará la palabra hola tres veces en la misma línea.

En este caso el valor que toma la variable de control no es importante, lo único que nos interesa es que la iteración se realice tres veces.

En otros casos el valor que toma la variable de control en cada iteración es fundamental.

Modifica el código del ejemplo anterior para, utilizando un bucle for con tres iteraciones, calcular los cuadrados de 3, 4 y 5.

Python considerará que pertenece al bucle todo el código indentado que tenga la misma indentación que la línea siguiente a la línea con la orden for.

La lista puede contener cualquier tipo de elementos, no sólo números.

Modifica el código del ejemplo anterior para, utilizando un bucle for con tres iteraciones, crear tres saludos personalizados a tres amigos que se llaman Luis, Mireia y Andrés.

Bucle controlado por una cadena de caracteres

En este caso:

- El bucle realizará tantas iteraciones como caracteres tenga la cadena de caracteres.
- En cada iteración la variable irá tomando sucesivamente en cada iteración cada uno de los caracteres.

```
for i in "AMIGO":  
    print("Dame una ", i)  
    print("¡AMIGO!")
```

Mostrará como resultado:

```
Dame una A  
Dame una M  
Dame una I  
Dame una G  
Dame una O  
¡AMIGO!
```

Escribe un programa que, utilizando un bucle for, muestre la tabla de multiplicar de un número que el usuario introduzca a través del teclado.

Bucle controlado por un range()

Un tipo range se puede definir de tres maneras:

range(a)	El bucle deberá realizar a iteraciones. La variable de control (i) toma el valor 0 en la primera iteración y va aumentando una unidad en cada pasada hasta alcanzar el valor a-1.
range(a,b)	El bucle deberá realizar b-a iteraciones. La variable de control (i) toma el valor a en la primera iteración y va aumentando una unidad en cada pasada hasta llegar al valor b-1.
range(a,b,c)	Similar al anterior pero ahora: <ul style="list-style-type: none">- El valor inicial de la variable de control es a.- El incremento del valor de la variable de control en cada iteración es c.- El bucle termina cuando la variable de control llega al valor b.

Ejemplo: Código para mostrar diez veces la palabra Hola utilizando un bucle for y un tipo range(a).

```
print("Comienzo")
for i in range(10):
    print("Hola ", end="")
    print()
print("Final")
```

Ejemplo: Código muestra los 100 primeros números naturales utilizando un bucle for y un tipo range(a,b).

```
for i in range(1,101):
    print(i)
```

Ejemplo: Múltiplos del 3 contenidos entre 1 y 100.

```
for i in range(3,101,3):
    print(i)
```

17.1 Contadores y acumuladores

En muchas ocasiones se necesitan variables que cuenten cuántas veces ha ocurrido algo o que acumulen valores.

En ambos casos es necesario dar un valor inicial a la variable.

Contador: Variable que lleva la cuenta del número de veces que se ha cumplido una condición.

El código asociado a un contador sería:

```
i = i + 1
```

Crea un programa que determine cuantos números naturales menores de 100 son múltiplos de tres.

Acumulador: Variable que en cada iteración va incrementando su valor en una cantidad (constante o variable).

El código asociado a un acumulador sería:

```
i = i + a
```

Donde a podrá ser un número o una variable.

Crea un programa que, utilizando un bucle for, calcule la suma de los 100 primeros números naturales.

17.2 Bucles anidados

Bucle anidado: Bucle incluido dentro del bloque de instrucciones de otro bucle principal.

Hablaremos de:

Bucle interior: El que se encuentra dentro del otro.

Bucle exterior: Al bucle principal.

Existen dos tipos de bucles anidados. Los de variables independientes y los de variables dependientes.

Bucle anidado con variables independientes

Las dos variables son independientes cuando los valores que toma la variable de control del bucle interior no dependen del valor de la variable de control del bucle exterior. Por ejemplo:

```
for i in [0, 1, 2]:
    for j in [0,1]:
        print('i vale', i, 'y j vale', j)
```

Es más recomendable utilizar tipos range() en lugar de listas. El código anterior quedaría:

```
for i in range(3):
    for j in range(2):
        print('i vale', i, 'y j vale', j)
```

En ambos casos el resultado sería:

```
i vale 0 y j vale 0
i vale 0 y j vale 1
i vale 1 y j vale 0
i vale 1 y j vale 1
i vale 2 y j vale 0
i vale 2 y j vale 1
```

Habitualmente se utiliza la letra i como nombre de la variable de control del bucle exterior y la j como nombre de la variable de control del bucle interior (k si hay un tercer nivel de anidamiento).

Bucle anidado con variables dependientes

Los valores que toma la variable de control del bucle interior dependen del valor de la variable de control del bucle exterior. Por ejemplo:

```
for i in [1, 2, 3]:
    for j in range(i):
        print("i vale", i, "y j vale", j)
```

Daríamos por resultado:

```
i vale 1 y j vale 0
i vale 2 y j vale 0
i vale 2 y j vale 1
i vale 3 y j vale 0
i vale 3 y j vale 1
i vale 3 y j vale 2
```

18. Bucle while

Bucle while: Bucle que repite un bloque de instrucciones *mientras se cumpla una condición*.

La sintaxis básica del bucle while es:

```
while condición:  
    Cuerpo del bucle
```

Veamos la orden un poco más a fondo:

Condición	Expresión lógica que se evalúa antes de cada iteración. Si se evalúa como true se ejecuta el cuerpo del bucle. Una vez ejecutado el cuerpo del bucle se vuelve a ser evaluada. Si se evalúa como false se termina la ejecución del bucle sin ejecutar el cuerpo del mismo.
Cuerpo del bucle	Está indentado y contiene el conjunto de órdenes que se ejecutarán en cada iteración.
Variable(s) de control	Variables que aparecen dentro de la condición. Estas variables pueden definirse antes del bucle while pero también puede modificarse su valor en el interior.

Veamos un ejemplo sencillo:

```
i = 1  
while i <= 3:  
    print(i)  
    i += 1  
print("Programa terminado")
```

Definimos un valor inicial para la variable de control. La condición del bucle es comprobar si el valor de la variable de control es menor o igual que tres. En caso de serlo escribiremos el valor de i en una unidad y volveremos a evaluar la condición. En este caso el bloque del código se ejecutará tres veces mostrando los valores 1, 2 y 3. Tras ello el Python saldrá del bucle y continuará con la siguiente orden, en este caso el print.

Podríamos pensar que un bucle while es equivalente a un bucle for y en algunos casos serán equivalentes pero existe una gran diferencia. Un bucle for se ejecutará un número fijo de veces. Sin embargo un bucle while se ejecutará un número indefinido de ocasiones, hasta que deje de cumplirse una condición. Por ejemplo:

Interpreta el siguiente código:

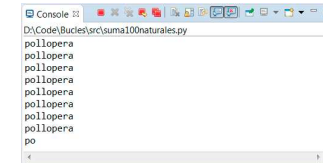
```
numero = int(input("Escriba un número positivo: "))  
while numero < 0:  
    print("¡Ha escrito un número negativo! Inténtelo de nuevo")  
    numero = int(input("Escriba un número positivo: "))  
print("Gracias por su colaboración")
```

18.1 Bucles infinitos

Bucle infinito: Bucle en el que la condición se cumple siempre, el bucle no terminará nunca de ejecutarse.

Aunque a veces es necesario utilizar bucles infinitos en un programa, normalmente se deben a errores que se deben corregir.

Los bucles infinitos no intencionados deben evitarse pues significan perder el control del programa. Para interrumpir un bucle infinito cuando estamos ejecutando un programa desde pyDev es suficiente con hacer clic en la tecla stop (cuadrado rojo) en la ventana de consola:



Cuando se está ejecutando el programa desde otras instancias, la ejecución del mismo se interrumpirá pulsando la combinación de teclas Ctrl+C. En ese caso se mostrará un mensaje tipo:

```
Traceback (most recent call last):  
  File "ejemplo.py", line 3, in <module>  
    print(i)  
Keyboard Interrupt
```

Encuentra la razón por la que se genera un bucle infinito en cada uno de los siguientes ejemplos:

```
i = 1  
while i <= 10:  
    print(i, "", end="")
```

```
i = 1  
while i > 0:  
    print(i, "", end="")  
    i += 1
```

```
i = 1  
while i != 100:  
    print(i, "", end="")  
    i += 2
```

18.2 Ordenes break, continue, pass y else en bucles

Estas órdenes permiten modificar la forma en la que se ejecuta la secuencia de comandos en un bucle for y while. Su función es:

break	Termina la ejecución del bucle más inmediato que la contiene.
continue	Termina la ejecución de la iteración actual pasando a la siguiente.
else	Solamente se ejecuta si el bucle termina normalmente (no por acción del break).
pass	No hace nada. Se utiliza para dar contenido a una orden que no va a hacer nada pero necesita por sintaxis tener algo escrito en ella.

18.2.1 break

Partimos del programa:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    i += 1
print("Programa terminado")
```

El bucle se ejecutará normalmente, realizado diez iteraciones, mostrando el resultado:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - Programa terminado
```

Si simplemente introducimos la orden break:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    i += 1
    break
print("Programa terminado")
```

El resultado tendrá la forma:

```
1 - Programa terminado
```

La orden break actúa en la primera iteración y por lo tanto termina la ejecución del bucle más próximo en la que está incluida (en este caso es el único).

Utilizar la orden break de esta forma no tiene mucho sentido ya que siempre se va a ejecutar, deteniendo el bucle. Normalmente la orden estará incluida dentro de un condicional. Con ello el bucle se saltará cuando se cumpla dicha condición. Un ejemplo sencillo:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    i += 1
    if i == 7:
        break
print("Programa terminado")
```

En este caso la salida sería:

```
1 - 2 - 3 - 4 - 5 - 6 - Programa terminado
```

18.2.2 continue

Si en lugar de break utilizamos continue, el resultado es diferente:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    i += 1
    if i == 7:
        continue
print("Programa terminado")
```

El resultado será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - Programa terminado
```

Parece que continue no ha hecho nada ya que el resultado es el mismo que en caso del programa sencillo. Pero no es así, el problema es que el resultado no es apreciable. Piensa en ello, continue termina la ejecución de la vuelta actual del bucle. Al estar colocado al final de la iteración el que i sea igual a 7 o no lo sea no afecta al resultado del bucle. Debemos cambiar la posición de la orden incremental:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    if i == 7:
        continue
    i += 1
print("Programa terminado")
```

En este caso el programa creará un bucle infinito que no terminará nunca ya que al llegar i al valor 7, la condición del if es cierta, se ejecuta el continue, terminando la vuelta del bucle actual y por lo tanto desde ese momento nunca se vuelve a ejecutar la orden i += 1 con lo que i continúa valiendo 7.

Si queremos evitar este problema podemos utilizar el código:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    if i == 7:
        i += 2
    continue
    i += 1
print("Programa terminado")
```

En este caso la salida será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 9 - 10 - Programa terminado
```

Sin mostrar el valor 8 ya que cuando i vale 7 modificamos su valor sumando dos unidades en lugar de una, la orden continue hace que i+=1 no se ejecute cuando i vale 7.

18.2.3 else

El else asociado a un bucle while o for delimita un conjunto de órdenes que solamente se ejecutan si hemos salido del bucle sin que se haya ejecutado una orden break.

Según el ejemplo anterior:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    if i == 7:
        break
    i += 1
else:
    print("Programa terminado")
```

La salida será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 -
```

Cuando i=7 el bucle se interrumpe, por lo tanto el bloque de ordenes asociado a else no se ejecutará.

En cambio:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    if i == 77:
        break
    i += 1
else:
    print("Programa terminado")
```

La salida en este caso será:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - Programa terminado
```

La orden break no llega a ejecutarse ya que i nunca puede llegar a tomar el valor 77. Por lo tanto el bucle termina normalmente y a continuación se ejecuta el bloque de instrucciones asociados al else.

18.2.4 pass

La mejor forma de entender la funcionalidad de esta orden es ver un ejemplo. Siguiendo con el caso anterior:

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    if i == 77:
        break
    i += 1
else:
    pass
```

Hemos eliminado las ordenes asociadas al bloque else:. El resultado es:

File "D:\Code\8. Ejercicios while 2\src\while201.py", line 9

```
^
SyntaxError: unexpected EOF while parsing
```

Se genera un error de sintáctico ya que else requiere que en su interior exista al menos una orden. La solución es incluir la orden pass que no tiene ninguna utilidad más que la de dar contenido (hacer montón).

```
i = 1
while i <= 10:
    print(i, '- ', end='')
    if i == 77:
        break
    i += 1
else:
    pass
```

La salida es:

```
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
```

19. 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 utilizemos 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**.

20. Funciones

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

20.1 Definición de una función

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

Otra razón para ello es que evidentemente la función tiene que haber sido creada antes de poder ser utilizada. Crearlas al comienzo del código asegura que las podremos 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:
 - o La palabra reservada `def`
 - o El nombre de la función (la guía de estilo de Python recomienda escribir todos los caracteres en minúsculas separando las palabras por guiones bajos).
 - o Paréntesis (pueden incluir los argumentos de la función, lo veremos más adelante)
 - o Símbolo de dos puntos.
- **Bloque de instrucciones:**
 - o Se escriben con sangría con respecto a la primera línea.
- **Finalización del bloque:**
 - o Se puede indicar el final de 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 2016 IES Fco. Grande Covián
Licencia CC-BY-SA 4.0
Programa terminado.
```


20.2 Variables en funciones

20.2.1 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 **alcance** o el **ámbito de las variables**.

Python distingue tres tipos de variables:

- **Variables locales:** Pertenecen al ámbito de la subrutina (pueden ser accesibles a niveles inferiores)
- **Variables libres:** A su vez pueden ser:
 - o **Variables globales:** Pertenecen al ámbito del programa principal.
 - o **Variables no locales:** Pertenecen a un ámbito superior al de la subrutina, pero que 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.

20.2.2 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).

20.2.3 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.

20.2.4 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.

21. 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:

21.1 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 programas o incluso en el mismo programa, 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 admiten **argumentos**.

Argumentos: Valores que se envían junto con la llamada a la función con los que esta podrá trabajar.

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 puede ser utilizada con más flexibilidad que la del ejemplo anterior, puesto que el programador puede elegir a qué valores aplicar la función.

21.2 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. Así las funciones se pueden reutilizar más fácilmente. 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.

21.3 Número variable de argumentos

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

En estos casos 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.

21.4 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.

21.5 Paso por valor o paso por referencia

Está fuera de los objetivos del curso.

22. Módulos, bibliotecas

En el punto 13 de estos apuntes definimos 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 quiere decir dividir el código total en bloques independientes. La aplicación completa se convierte en una especie de puzzle que necesita de todas las pequeñas piezas independientes (módulos) para trabajar.

En ocasiones puede suceder que queramos utilizar 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 poner 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 el 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))
```

Daríamos 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)
```

Daríamos como resultado:

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

22.1 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).

22.2 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.

Los módulos pueden importar otros módulos.

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.

22.3 El camino de búsqueda de los módulos - Organización

Cuando un archivo intenta importar un módulo, Python lo busca en la lista de directorios dada por la variable sys.path. Esta lista comienza con el directorio que contiene al script de entrada (o el directorio actual), los directorios definidos en PYTHONPATH, y el directorio default dependiente de la instalación.

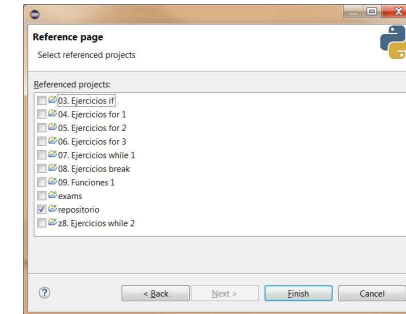
Si guardamos el módulo en el mismo directorio que el programa principal que lo va a utilizar no tendremos ningún problema. Sin embargo, en muchas ocasiones nos interesará que el módulo sea accesible a todos los programas de nuestros proyectos.

Trabajando con Eclipse podremos vincular nuestro módulo principal a los módulos contenidos en un directorio externo de dos formas diferentes:

22.3.1 Asociando un directorio externo en el momento de la creación del proyecto

Esta forma de trabajar permite asociar un proyecto nuevo a todas las carpetas que queramos. Sólo es posible realizar este proceso en el momento en que estamos definiendo las características del proyecto. El procedimiento es el siguiente:

1. Creamos un proyecto con el nombre “repositorio” (a modo de ejemplo). Este será el proyecto que contendrá los módulos que luego podremos importar.
2. Guardaremos cualquier módulo que queramos sea accesible por varios proyectos en el directorio src del proyecto “repositorio”.
3. Para crear un proyecto con acceso a la carpeta donde están guardados los módulos seleccionaremos el icono que lo define en el explorador de paquetes PyDev (ventana de la izquierda de la pantalla) seguiremos el proceso habitual para crear un nuevo proyecto pero una vez que en la ventana PyDev Project hayamos definido el nombre y versión de Python a utilizar haremos clic en el botón “Next” en lugar del botón Finish:
4. Se mostrará la ventana Reference page. En ella activaremos la casilla correspondiente a la carpeta repositorio:

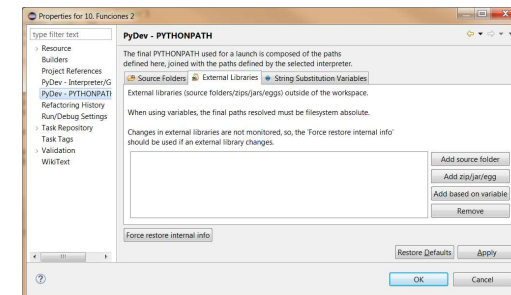


5. Hacemos clic en Finish y ya podemos trabajar.

A partir de ese momento ya podemos invocar los módulos contenidos en dicho directorio en nuestro proyecto al haber añadido su carpeta a las direcciones contenidas en sys.path.

22.3.2 Añadir un directorio con módulos a un proyecto ya existente

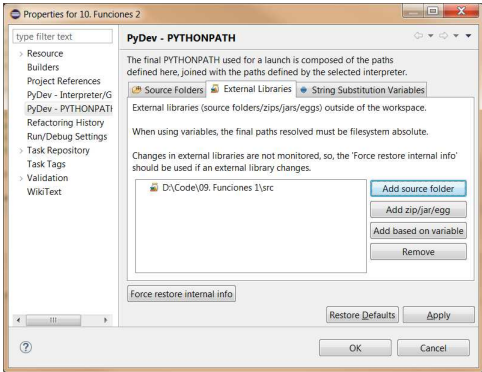
1. En este caso deberemos ir al explorador de paquetes PyDev (ventana de la izquierda de la pantalla) y hacer clic con el botón derecho del ratón sobre el icono que contiene el nombre del proyecto que quiere utilizar funciones externas. En el menú emergente seleccionamos la opción “Propiedades”.
2. Se abrirá la ventana Properties for (nombre del proyecto). En la columna de la izquierda de esta ventana seleccionamos la opción PyDev – PYTHONPATH. Seleccionamos la pestaña central “External Libraries” para añadir el directorio que van a contener los módulos externos.



3. Hacemos clic en el botón “Add source folder” y buscamos el directorio donde están guardados los módulos:

OJO: Has de definir la carpeta en la que está el módulo a importar (en nuestro caso hay que seleccionar la carpeta src del proyecto del que vas a importar).

El problema es saber dónde se están guardando los proyectos que creamos con Eclipse. Este directorio se define en el momento de realizar la instalación de Eclipse. Si no sabes cual es, comienza a crear un nuevo proyecto y en la ventana de configuración puedes ver donde se va a guardar. En el caso de los ordenadores del centro el directorio es C:\Usuarios\alumno\workspace.



4. Hacemos clic en Apply y a continuación en OK.

23. 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 proyecto con Eclipse también añadimos la referencia al directorio en que está guardada la biblioteca estándar.

23.1 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 sample 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)
```

23.2 Información sobre los módulos de la biblioteca estándar

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

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

24. 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.

24.1 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 una variable, c. La cuarta línea muestra el resultado y la quinta un mensaje final. En principio 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
```

Por otro lado ejecución de programa termina al generarse el error y ninguna de las dos órdenes print() finales llega a ejecutarse.

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

- Pedir al programa que intente hacer una instrucción.
- En caso de que no pueda hacerse esa 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 el 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.

24.2 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 cueros\\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("mensaje")
```

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.

24.3 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.

24.4 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("Cuidate...")
```

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("Cuidate...")
```

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 cueros\\src\\print.py", line 8, in <module>
    print(calculaRaiz(opt1))
  File "...\\Ensayos cueros\\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
```

25. Diccionarios

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

Con un ejemplo lo entenderemos mejor. Un diccionario podría 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.

25.1 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 (no tienen por qué serlo): 'x', 'y', 'z'.
- En este caso el **valor guardado** en el término con clave '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

Para acceder al valor asociado a una clave, utilizamos el mismo método que para acceder a un elemento de una lista, pero utilizando la clave elegida en lugar del índice.

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.

25.2 Método 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 término 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 método **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 otro valor incluyéndolo como segundo parámetro en la llamada al método:

```
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”, el método 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
```

25.3 Formas de acceder a todos los elementos de un diccionario

Hasta ahora para acceder a un término del diccionario es necesario conocer y utilizar 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
```

25.4 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:

Funciones:

len(name)	Devuelve el número de elementos en el diccionario.
-----------	--

Métodos:

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 no existe, 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

25.5 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.

26. 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 método:

```
import os
```

Veamos un par de aplicaciones muy útiles.

Averiguar el tipo de sistema operativo utilizado

La sentencia 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)
```

27. 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

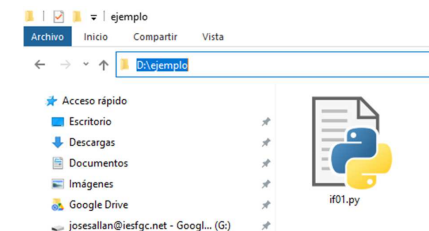
Vamos a utilizar la aplicación pyinstaller. En primer lugar tenemos que instalarla en nuestro ordenador. Para ello abrimos una consola (Windows), tecleando cmd desde el botón de inicio de Windows y ejecutando la orden:

```
pip install pyinstaller
```

Ejecutamos pyinstaller

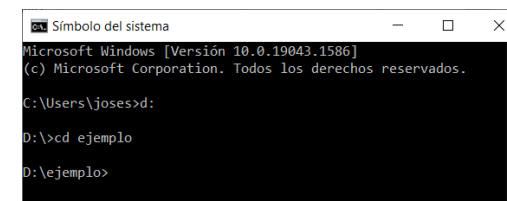
Debemos navegar en la consola hasta situarnos dentro del directorio en que está el módulo del cual queremos crear el ejecutable.

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

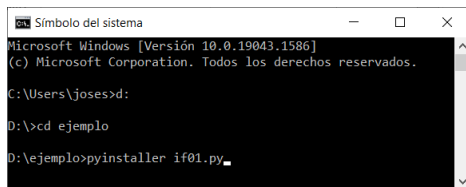


En este caso la dirección del directorio que contiene el módulo es: D:\ejemplo

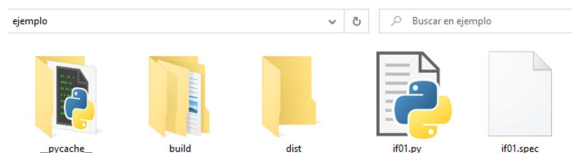
Vamos a la consola y nos movemos a esa dirección:



Ejecutamos ahora la orden pyinstaller seguida de un espacio y ifdel 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). Entramos dentro de ella y 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
```

28. Manejo de archivos con Python

Archivo: Información en forma de bits 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 soporte de almacenamiento dos archivos con igual nombre, extensión y path.

Cuando hablamos de manejo de archivos nos referimos cualquier tipo de operación que permite crear, modificar o eliminar un archivo o la información que hay almacenada en él.

Las operaciones básicas a realizar con un archivo son la lectura y escritura de datos en el mismo.

28.1 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 dentro del código.

Una vez hayamos terminado de trabajar con el archivo lo cerraremos utilizando el método close() del objeto file.

```
fichero.close()
```

28.2 Leer un archivo

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

28.2.1 Lectura a través de un for

la forma más sencilla de leer la información contenida en él es utilizando un for que vaya leyendo cada una de sus líneas. Veamos un ejemplo.

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

```
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 código necesario es:

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

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

El for utiliza la variable línea para ir leyendo en cada iteración una línea del archivo, la línea print muestra la información almacenada en la variable "línea" en cada pasada.

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" "línea".

Método rstrip()

Aplicados 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 .rstrip() 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**.

28.2.2 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\n\ncan do that can't be done.\n", "Nothing you can sing that can't be sung.\n",\n'Nothing you can say but you can learn how to play the game.\n', "It's easy.\n", '\n',\n"Nothing you can make that can't be made.\n", "No one you can save that can't be\nsaved.\n", 'Nothing you can do but you can learn how to be you in time.\n', "It's\neasy.\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.
```

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

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

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

Leerá la canción completa.

En cambio:

```
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) leerá "numero" caracteres dentro de la cadena desde la posición actual (relacionado con el punto siguiente seek y tell),

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

En este caso si el fichero file2.txt no existía previamente y había sido incluido en el proyecto, no se mostrará dentro del pyDev Package Explorer. Sin embargo si entramos dentro de la carpeta que lo contiene (en nuestro caso incluida dentro de D:\Code) podremos abrirlo y trabajar con él normalmente. 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).

28.4 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".

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

28.5 seek y tell. Definiendo la 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 ls
Posición final 15
Love -
```

- El primer `fh.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 ls).
- 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

28.6 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.

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.

28.7 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:

28.7.1 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.

28.7.2 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...)

28.8 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.

28.9 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.
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).

29. Introducción a la 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) o tipos de lenguajes de programación:

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

Programación orientada a procedimientos

Fueron los primeros lenguajes de programación de alto nivel que se utilizaron. 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

La idea es simplificar el proceso de programación creando elementos a los que llamaremos **objetos**. Un objeto en POO tiene la misma esencia que tendría 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.

Lenguajes que siguen este paradigma 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 fundamental de este tipo de lenguajes es que requiere que nos tendremos que familiarizar con un vocabulario (Clase, objeto, ejemplar de clase, modularización...) que puede resultar inicialmente un tanto complejo.

29.1 Conceptos previos

1. Clase:

Una clase es un **modelo** donde se redactan **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 que vienen 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.
- Porque 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 lo que llamaremos **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** (miCoche)
- **Para acceder a sus propiedades**, escribimos el nombre del objeto seguido de un punto y del nombre de la propiedad:
miCoche.color="rojo"
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()
```

30. Pasando a código lo visto hasta ahora

30.1 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():
```

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

30.1.1 Añadir 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
```

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, tener cuatro ruedas e inicialmente no estar en marcha.

30.1.2 Añadir comportamientos, procedimientos o métodos:

Un método va a ser un bloque de código una “función” que pertenece a una clase que permite hacer una o varias acciones.

El procedimiento será el siguiente, supongamos que queremos crear un método que permita arrancar a los coches que creemos más adelante:

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 el procedimiento arrancar. Este método permitirá cambiar el estado enmarcha en los objetos que creemos en un futuro. Para ello crearemos un método que cambie el valor de la propiedad enmarcha de False a True:

```
class Coche():
```

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

30.2 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 deberemos **ejecutar el comportamiento** arrancar:

```
miCoche.arrancar()
```

Al leer esta orden, Python llama al método arrancar. El método recibe como parámetro (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())

```

30.2.1 Creando un segundo objeto de la misma clase

Seguimos con el ejemplo anterior. 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().

30.3 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 (al igual que ocurría con las funciones). La diferencia es que en la cláusula que define el nombre del método tendrá que contener el nombre del parámetro, pero sin olvidar el nombre del parámetro por defecto self, que hacía 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()
```

31. 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
```

31.1 Crear objetos utilizando 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:

```
miCoche=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.

32. 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()
```

33. 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 vamos mejor un poco nuestro ejemplo del coche.

Pretendemos introducir la novedad de que cuando llamemos 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 justo antes de arrancar. Así, el método chequeo_interno ha de ser llamado antes de realizar la acción. Para ello 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 "
```

Es decir, 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 no hayamos enviado el valor True como parámetro en el método arrancar, el coche está 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 es suficiente con encapsular el método. Al hacerlo su uso quedará restringido al interior de la clase.

La sintaxis es sencilla:

- Comenzar con dos guiones bajos su nombre en la definición.
- Al llamar el método más adelante, recordar que debemos añadir los dos guiones bajos que forman parte del nombre.

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 (chequeo==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.

34. Método __str__

El método __str__ se ejecutará cuando llamemos desde el código el nombre del objeto:

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 asociar un procedimiento al 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
```

35. 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 lo tanto la clase 2 es **superclase** de ellas y 3, 4 y 5 son subclases de 2.

35.1 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 plantearemos 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).

35.2 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 Vehículo

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 añadido 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
```

35.3 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, esta 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 a 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).

35.4 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
```

35.5 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 cargarEnergia(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 Vehículo pero también las de la clase VElectrico.

35.5.1 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 de 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 parámetros, si hubiera llamado primero a VElectricos no habría que poner ninguno.

35.6 Función super()

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

Vamos a partir de un ejemplo nuevo. Su código es:

```
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():
    def __init__(self,salario,antigüedad):
        self.salario=salario
        self.antiuedad=antigüedad
```

Hemos definido dos clases:

- La clase Persona tiene cuatro parámetros (self, nombre, edad y lugar_residencia). El constructor define tres propiedades y añadimos un método que muestra por pantalla las características de los objetos de esta clase.
- La clase Empleado tiene tres parámetros (self, salario y antigüedad). El constructor define dos propiedades.

Sin embargo, todos los Empleados son Persona por lo tanto la clase Empleado debe de heredar de la clase Persona, así la definición de la clase Empleado debe incluir la llamada a la clase 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,
              "Residencia:",self.lugar_residencia)

class Empleado(Persona):
    def __init__(self,salario,antigüedad):
        self.salario=salario
        self.antiuedad=antigüedad
```

Construyamos un objeto de tipo persona:

```
Antonio=Persona("Antonio",17,"Zaragoza")
Antonio.descripcion()
```

El resultado por pantalla será:

```
Nombre: Antonio Edad: 17 Residencia: Zaragoza
```

De momento todo funciona de forma correcta.

Eliminemos ahora el objeto Antonio como persona y creemos ahora un objeto Antonio tipo Empleado. Si ejecuto:

```
Antonio=Empleado("Antonio",17,"Zaragoza")
Antonio.descripcion()
```

Se genera un error:

```
Traceback (most recent call last):
  File "...\\Ensayos curos\\src\\herencia super.py", line 13, in <module>
    Antonio=Empleado("Antonio",17,"Zaragoza")
TypeError: __init__() takes 3 positional arguments but 4 were given
```

Ya que estamos enviando cuatro argumentos (recuerda que siempre enviamos también self) y esta clase solo tiene tres, que además no son los que estamos enviando.

Sin en lugar de esto, pasamos los parámetros esperados (salario y antigüedad):

```
Antonio=Empleado(1500,15)
Antonio.descripcion()
```

También generamos un error:

```
Traceback (most recent call last):
  File "...\\Ensayos curos\\src\\herencia super.py", line 14, in <module>
    Antonio.descripcion()
  File "...\\Ensayos curos\\src\\herencia super.py", line 7, in descripcion
    print("Nombre:", self.nombre, "Edad:", self.edad, "Residencia:",
    self.lugar_residencia)
AttributeError: 'Empleado' object has no attribute 'nombre'
```

En este caso el método descripción no reconoce el atributo nombre (por ser el primero) ya que no lo hemos enviado al crear el objeto.

Sin embargo un objeto Empleado ha de tener nombre, edad y lugar de residencia.

La solución a este problema está en la función super():

Dentro del constructor que está heredando, en este caso de la clase Empleado, introducimos la orden:

```
super().__init__("Antonio", 17, "Zaragoza")
```

Con lo cual el código completo quedaría:

```
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,antigüedad):
        super().__init__("Antonio", 17, "Zaragoza")
```

```
self.salario=salario
self.antigüedad=antigüedad

Antonio=Empleado(1500,15)
Antonio.descripcion()
```

Dando por resultado:

```
Nombre: Antonio Edad: 17 Residencia: Zaragoza
```

La función super() llama al método __init__ de la clase padre (persona), enviándole los tres parámetros necesarios.

Ya es algo, pero este resultado es incompleto por dos razones:

- Los datos de persona se han metido como constantes, no parámetros, al crear el objeto.
- El método descripción no nos informa de los datos de Empleado.

Enviar los parámetros:

Habrà que:

- Incluir los nuevos parámetros en el método __init__ de la clase hijo.
- Incluir estos nuevos parámetros en la llamada al método __init__ de la clase padre.
- Incluir el valor de los cinco parámetros en la llamada al método descripción.

```
class Empleado(Persona):
    def __init__(self,salario,antigüedad, nombre_empleado, edad_empleado,
residencia_empleado):
        super().__init__(nombre_empleado, edad_empleado, residencia_empleado)
        self.salario=salario
        self.antiüedad=antigüedad

Antonio=Empleado(1500,15,"Antonio",17,"Zaragoza")
Antonio.descripcion()
```

Mostrar el estado completo:

- Crearemos un nuevo método descripción para esta clase, pero aprovecharemos lo ya utilizado en el método descripción de la clase padre. Añadiremos la nueva información:

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

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,antigüedad, nombre_empleado, edad_empleado,
residencia_empleado):
```

```
super().__init__(nombre_empleado, edad_empleado, residencia_empleado)
self.salario=salario
self.antigüedad=antigüedad
def descripcion(self):
    super().descripcion()
    print("Salario:", self.salario,"Antigüedad:",self.antigüedad)
Antonio=Empleado(1500,15,"Antonio",17,"Zaragoza")
Antonio.descripcion()
```

Dando como resultado:

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

Modifica el código del ejemplo de los puntos anteriores (choches) para que el objeto moto eléctrica pueda mostrar todas sus propiedades por pantalla (las de Vehículo y las de VElectrico)

35.7 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:

```
print(isinstance(Antonio,Empleado))
```

Generará por pantalla:

```
True
```

Si comprobamos si el objeto Antonio es también un Persona:

```
print(isinstance(Antonio,Persona))
```

Generará por pantalla:

```
True
```

Ya que todos los objetos Empleados son objetos Persona.

36. 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)
```

Daríamos por resultado:

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

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

37. POO. Métodos de cadenas

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 modo de ejemplo como por ejemplo las cadenas de texto, Python las considera objetos y como utilizar los métodos asociados a su clase nos va a facilitar mucho tareas que de otra forma serían más complejas.

Las cadenas de caracteres son objetos de la clase **String**.

La clase String tiene multitud de métodos que nos permitirán hacer diferentes tareas. Los más habituales podrían ser:

Método	Descripción
upper()	Convierte en mayúsculas todos los caracteres de la cadena.
lower()	Convierte en minúsculas todos los caracteres de la cadena.
capitalize()	La primera letra de la cadena pasa a ser mayúscula, el resto serán minúsculas.
count()	Cuenta cuantas veces aparece un carácter o un grupo de caracteres en 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.
isalnum()	Devuelve True/False en función de que la cadena sea 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.

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
```

38. Módulos que incluyen clases

En el punto 22 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.

Creemos 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 el punto 0 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
```

39. 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: XXXXXXXX_XX.py
# Autor: XXXXX XXXXXXXX (nombre y apellidos)
# Fecha: XX de XXXXXX de XXXX
# Descripción: Ejercicio XXXXXXXX. Este programa blablabla
# Si la descripción es larga, escriba varias líneas
```

40. Funciones

Listado de las funciones más habituales.

[illegible]

40.1 Formateando la presentación de números

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'))

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

# Justificación 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'))
```


41. Métodos

Listado de métodos de interés.

[illegible]

42. Tabla de contenido

1.	Introducción	1
2.	Algoritmos	2
3.	Diagramas de flujo	3
3.1	Principales elementos en un Diagrama de Flujo simple	3
	Terminal	3
	Datos	3
	Proceso	3
	Decisión	3
	Proceso Predefinido	4
	Líneas de flujo	4
	Anotación	4
	Entrada Manual	4
	Documentación	4
	Base de Datos	4
3.2	Reglas para la elaboración de un Diagrama de Flujo	4
3.3	Como hacer un diagrama de flujo	5
3.4	Ejemplos de Diagramas de Flujo	6
4.	Tipos de lenguajes de programación	8
4.1	¿Por qué Python?	9
5.	Instalación Python	10
5.1	Instalación en Windows 7-10	10
6.	Ejecutar un programa Python	12
7.	Utilizar Python directamente desde la línea de comandos	15
8.	Tipos de datos	16
9.	Operadores. Trabajando con números	17
10.	Comenzando a utilizar variables	18
11.	Cadenas de caracteres – función print()	20
11.1	print() con más de un argumento	21
11.2	Opciones sencillas cadenas de caracteres y print()	22
11.3	Índices en las cadenas de texto	24
12.	Listas	25
12.1	Opciones muy interesantes trabajando con listas	26
13.	Ejecutar un archivo .py	27
13.1	Ejecutar el archivo desde la línea de comandos	27
13.1.1	Nuestro primer programa	27
13.1.2	Ejecutar desde la línea de comandos	28
13.2	Ejecutar haciendo clic en el icono del módulo	28
13.3	Interfaz de usuario IDLE	29
13.4	Eclipse y PyDev	30
13.5	Sublime text	30
13.6	PyCharm	30
14.	Programar con Eclipse. Primer programa	31
14.1	Pasos previos 1: Crear un proyecto	31
14.2	Pasos previos 2: Crear un nuevo archivo de código fuente	32

14.3	Comentando tus programas.....	32
14.4	Código que genera el texto por pantalla.....	32
14.5	Compilar y ejecutar la aplicación.....	32
15.	Pasando valores al programa –input().....	33
15.1	Números e input().....	33
15.2	Argumento de la función input().....	34
16.	Sentencias condicionales: if... elif...else.....	35
16.1	Sentencia condicional if... (Una opción).....	35
16.1.1	Comparadores lógicos.....	35
16.1.2	Operadores lógicos.....	36
16.1.3	Expresiones compuestas.....	36
16.2	if ... else ... (dos opciones).....	36
16.3	if... elif... else... (más de dos opciones).....	37
16.4	Sentencias condicionales anidadas.....	38
16.5	Sintaxis alternativa con valores lógicos.....	38
17.	El bucle for.....	40
17.1	Contadores y acumuladores.....	42
17.2	Bucles anidados.....	43
18.	Bucle while.....	44
18.1	Bucles infinitos.....	45
18.2	Ordenes break, continue, pass y else en bucles.....	46
18.2.1	break.....	46
18.2.2	continue.....	47
18.2.3	else.....	48
18.2.4	pass.....	48
19.	Concepto de subrutina.....	50
20.	Funciones.....	51
20.1	Definición de una función.....	51
20.2	Variables en funciones.....	52
20.2.1	Conflictos de nombres de variables.....	52
20.2.2	Variables locales.....	52
20.2.3	Variables libres globales o no locales.....	54
20.2.4	Variables declaradas global o nonlocal.....	55
21.	Funciones: Argumentos y devolución de valores.....	57
21.1	Función con argumentos.....	57
21.2	Utilizando return. Devolviendo valores.....	58
21.3	Número variable de argumentos.....	58
21.4	Funciones que devuelven más de un valor.....	59
21.5	Paso por valor o paso por referencia.....	59
22.	Módulos, bibliotecas.....	60
22.1	Importando directamente las funciones.....	61
22.2	Más sobre los módulos.....	62
22.3	El camino de búsqueda de los módulos - Organización.....	62
22.3.1	Asociando un directorio externo en el momento de la creación del proyecto.....	62
22.3.2	Añadir un directorio con módulos a un proyecto ya existente.....	63
23.	Biblioteca estándar – Módulo random.....	65
23.1	Módulo random.py.....	65

23.2	Información sobre los módulos de la biblioteca estándar.....	65
24.	Excepciones y su gestión.....	66
24.1	Gestión del error división por cero.....	66
24.2	Gestión del error tipo de dato – While, break.....	67
24.3	Clausula finally.....	69
24.4	Lanzar excepciones. Instrucción Raise.....	69
25.	Diccionarios.....	72
25.1	Crear un diccionario.....	72
25.2	Método get().....	73
25.3	Formas de acceder a todos los elementos de un diccionario.....	74
25.4	Funciones y métodos a utilizar con diccionarios.....	76
25.5	Usos de diccionarios.....	76
26.	Módulo os.system() – Borrar pantalla de consola	77
27.	Crear archivos ejecutables.....	79
28.	Manejo de archivos con Python.....	81
28.1	Abrir en modo lectura y cerrar archivos.....	81
28.2	Leer un archivo.....	82
28.2.1	Lectura a través de un for.....	82
28.2.2	Método readlines(). Leer un fichero como lista.....	83
28.2.3	Método read(). Leer un archivo como cadena de caracteres.....	84
28.3	Abrir en modo escritura y escribir en un archivo.....	84
28.4	Añadir información a un archivo ya existente.....	85
28.5	seek y tell. Definiendo la posición del puntero en el archivo.....	86
28.6	Leer y escribir en el mismo archivo.....	87
28.7	Módulo pickle.....	88
28.7.1	Método dump.....	88
28.7.2	Método load.....	89
28.8	Módulo shelve.....	90
28.9	Resumen modo de apertura de archivos.....	91
29.	Introducción a la programación orientada a objetos.....	92
29.1	Conceptos previos.....	93
1.	Clase:.....	93
2.	Ejemplar de clase = instancia de clase = objeto perteneciente a una clase:.....	93
3.	Modularización.....	93
4.	Encapsulación.....	93
5.	Nomenclatura del punto.....	93
30.	Pasando a código lo visto hasta ahora.....	94
30.1	Crear una clase.....	94
30.1.1	Añadir propiedades y estados:.....	94
30.1.2	Añadir comportamientos, procedimientos o métodos:.....	94
30.2	Creando objetos.....	95
30.2.1	Creando un segundo objeto de la misma clase.....	96
30.3	Métodos que reciben parámetros.....	98
31.	Constructor, definir el estado inicial.....	99
31.1	Crear objetos utilizando un constructor con parámetros.....	99
32.	Encapsulación de variables.....	101
33.	Encapsulación de métodos.....	103

34.	Método __str__	108
35.	Herencia	109
35.1	Finalidad de la herencia en programación	109
35.2	Sintaxis	110
35.3	Añadir propiedades y métodos a una subclase.....	111
35.4	Sobrescribir un método heredado	111
35.5	Herencia múltiple	112
35.5.1	Sintaxis	112
35.6	Función super().....	114
35.7	Función isinstance()	117
36.	Polimorfismo	118
37.	POO. Métodos de cadenas.....	120
38.	Módulos que incluyen clases	122
39.	Libro de estilo.....	124
40.	Funciones	127
40.1	Formateando la presentación de números	127
41.	Métodos	128
42.	Tabla de contenido	129
43.	Instalar Eclipse y PyDev –Anexo I.....	133
43.1	Instalar el complemento pyDev en Eclipse.....	136
43.2	Configurar Eclipse	138
43.2.1	Tildes y ñ	138
43.2.2	Corrector ortográfico en español	138
43.2.3	Tamaño y tipo fuente	139
43.2.4	Colores. Syntax highlighting.....	139
43.2.5	Línea guía 80 caracteres	139
44.	Exportar e importar código a Eclipse –Anexo II.....	140
44.1	Exportar código Eclipse a un archivo comprimido.zip.....	140
44.2	Importar código Eclipse desde un archivo comprimido.zip	141
44.2.1	Archivos con un único proyecto en ellos.....	141
44.2.2	Archivos con más de un proyecto en ellos	142
45.	Materiales referencia.....	144

43. Instalar Eclipse y PyDev –Anexo I

Eclipse no se instala como tal. Simplemente se descomprime en una carpeta y se crea en el escritorio un acceso directo que apunta a ella.

Por defecto Eclipse se “instala” en la carpeta del usuario, quedando asociada a él.

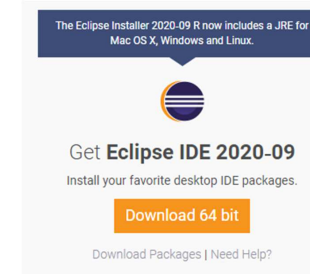
Instalación en Windows 7 – versión 2020 09

Es posible acceder a todos los enlaces descritos en este y los siguientes apartados de la instalación de Eclipse en Windows desde el sitio web del centro: [Enlace](#).

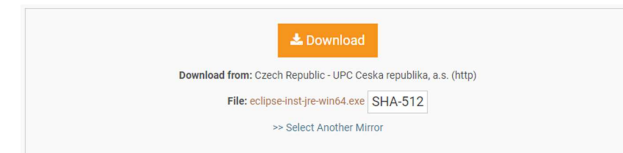
Desde cualquier navegador accedemos a la página de descarga de Eclipse:

<https://www.eclipse.org/downloads/>

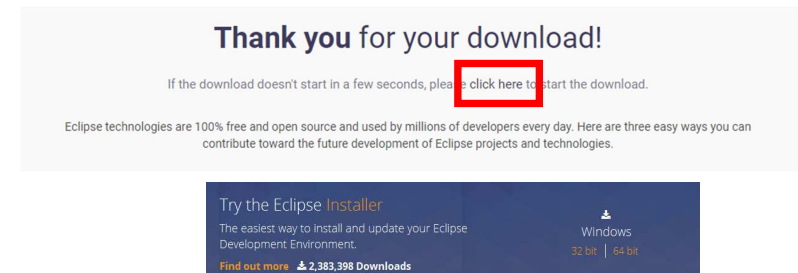
Eclipse necesita para poder funcionar que en el equipo esté instalado un **entorno de desarrollo de Java (JRE)**. Desde la página anterior se ofrece una versión de instalador que incluye el paquete JRE necesario para esta versión. Para iniciar el proceso de instalación hacemos clic en el botón “Download 64 bit”



El navegador nos redirigirá a una nueva ventana en la que hacemos clic en el botón download:



Se mostrará una nueva ventana en la que verás la siguiente sección:



En función del navegador que estés utilizando, es posible que no se inicie la descarga del archivo. En ese caso haz clic con el botón derecho del ratón sobre el enlace “click here” que se indica marcado en un rectángulo rojo en la imagen anterior y selecciona la opción “Abrir enlace en una pestaña nueva”.



Descargamos el archivo en nuestro disco duro y hacemos clic sobre su icono para iniciar la instalación.

En la primera ventana seleccionamos la opción Eclipse IDE for Java Developers.



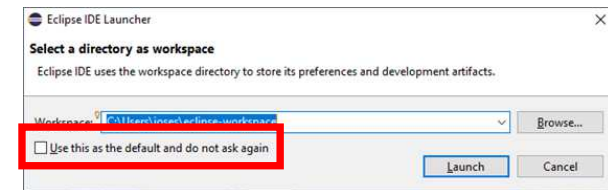
En la siguiente ventana debemos definir el directorio de instalación para el JVM de Java y para Eclipse. Podemos dejar las opciones por defecto:



Se iniciará el proceso de instalación. Al terminar se mostrará:

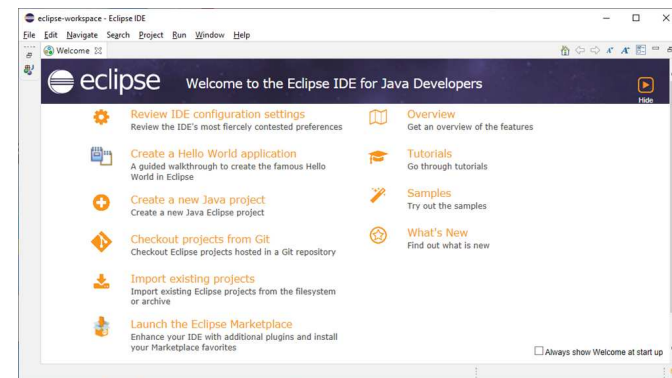


Clic en el botón Launch y se mostrará una ventana preguntando por la dirección del directorio de trabajo. En mi caso dejo el directorio por defecto. Activamos el tic “Use this as the default and do not ask again”:



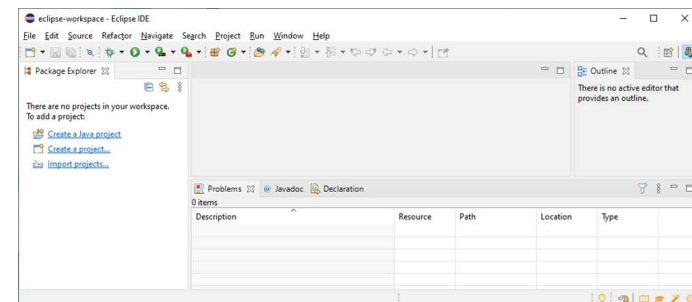
Aviso: Ten en cuenta que este directorio está congelado en el ordenador del aula.

Clic en Ok y se abrirá el programa:



Si no queremos que se muestre esta ventana de inicio cada vez que abra el programa, comprobamos que está desactivado el Tic “Always show Welcome at start up” situado en la esquina inferior derecha y a continuación hacemos clic en el botón “Hide” situado en la esquina superior derecha de la ventana de Eclipse.

Tras cerrar esta ventana de inicio se mostrará la ventana de trabajo:



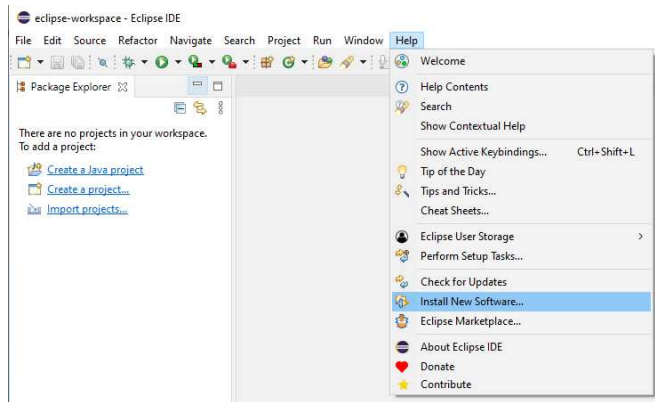
En un punto siguiente describiremos las partes principales de esta ventana. De momento vamos a configurar las características que nos interesan para poder trabajar con Python.

43.1 Instalar el complemento pyDev en Eclipse

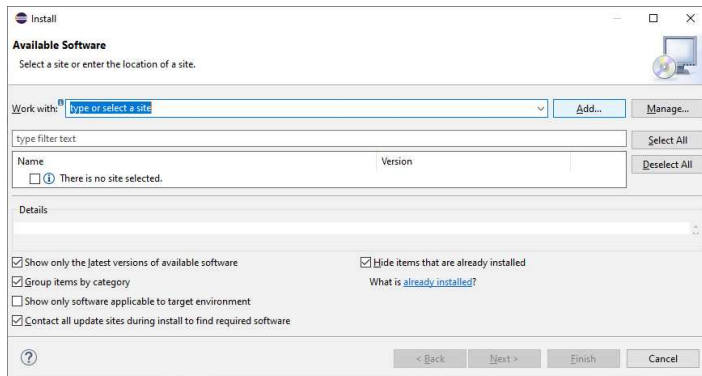
Instalación siguiendo el proceso indicado en: http://www.pydev.org/manual_101_install.html

Tal y como lo hemos instalado, Eclipse solamente permitiría programar en el lenguaje Java. Para poder trabajar con Python es necesario instalar el complemento pyDev.

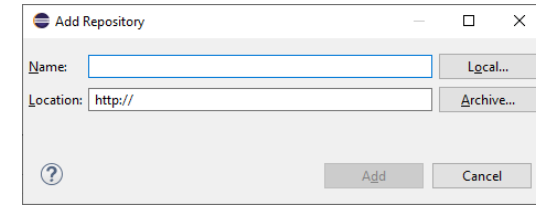
Para instalar el complemento pyDev debemos tener abierto Eclipse. Seleccionamos la opción Help del menú superior y en ella la opción “Install New Software...”



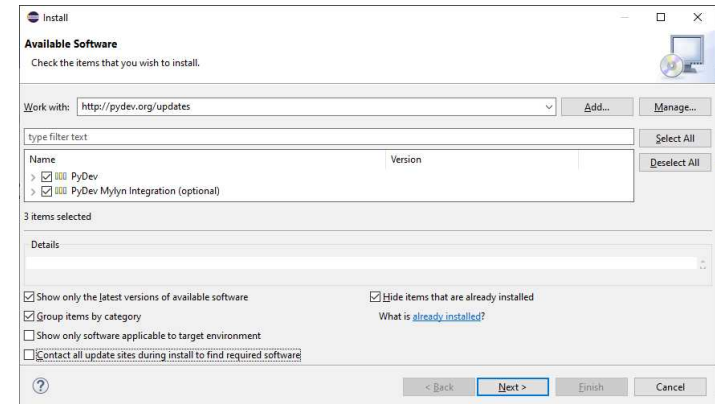
En la siguiente ventana pulsaremos el botón Add...



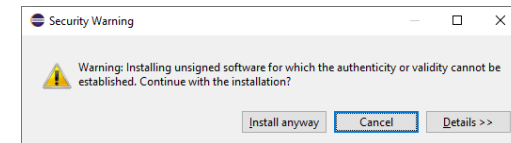
Se mostrará la ventana en la que tenemos que permite añadir repositorios de descarga:



Añadimos en la ventana “Location” el enlace <http://pydev.org/updates> y pulsamos el botón Add.



Seleccionamos pyDev y pyDev Mylyn Integration. Desactivamos la opción “Contact all update sites during install to find required software” y pulsamos Next dos veces. En la pantalla siguiente seleccionamos “I accept the terms of the license agreements”. Pulsamos Finish y esperamos a que progrese la instalación. Se mostrará:

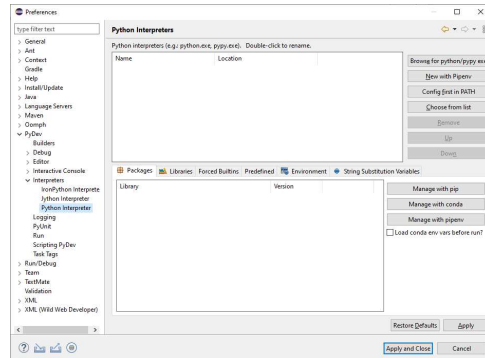


Pulsamos en “Install anyway”. Al terminar la instalación el programa solicitará reiniciar Eclipse, pulsamos Yes.

Todavía nos queda un paso más. Indicar a Eclipse donde está instalado Python.

Para ello en el menú superior seleccionamos la opción Window y en el menú que se abrirá a continuación seleccionamos Preferences.

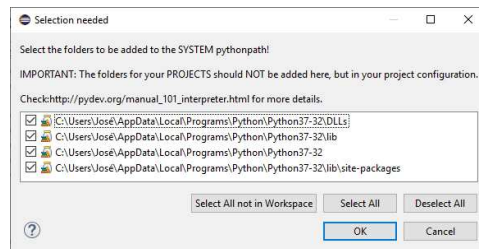
En el menú de la izquierda seleccionamos PyDev, Interpreters, Python Interpreter:



Hacemos clic en el botón Choose from list. Eclipse detectará todas las versiones de intérprete Python que tengamos instaladas en nuestro equipo, por ejemplo:

Seleccionamos todos los elementos y pulsamos OK. A continuación, clic en “Apply and Close” en la ventana de preferencias.

Ya tenemos Eclipse casi listo...



43.2 Configurar Eclipse

43.2.1 Tildes y ñ

Por defecto Eclipse codifica los archivos utilizando codificación ANSI por lo que no es capaz de procesar caracteres “extraños” como las tildes y las ñ. Para solucionar este problema cambiaremos el sistema de codificación a UTF-8. Para ello hay que seguir los siguientes pasos:

Desde el menú superior seleccionamos Window>Preferences.

En la lista de la izquierda, en la ventana Preferences, seleccionamos General>Content Types.

En la zona de la derecha seleccionamos Text y en default encoding (parte inferior de la ventana) escribimos UTF-8, clic en Update y en Apply and Close.

43.2.2 Corrector ortográfico en español

El corrector ortográfico instalado por defecto en Eclipse es el correspondiente al idioma Inglés. Para instalar un diccionario español (útil si vamos a utilizar expresiones en español en el código, comentarios, textos...) hay que seguir los siguientes pasos:

En primer lugar, descargamos un archivo con el diccionario en español. En la carpeta de software podemos encontrar en el sitio web del centro:

<https://grandecovian.es/files/D. Tecnología/TIC II/Python/diccionario.rar>

Descomprimos el archivo en el directorio donde lo vayamos a almacenar y desde el menú principal de Eclipse vamos a:

Window>Preferences>General>Editors>Text Editors>Spelling.

- Seleccionamos la opción “None” en el desplegable Platform dictionary.
- Seleccionamos UTF-8 como Encoding.
- Hacemos clic en el botón **Browse** y seleccionamos el archivo de texto “ES.txt”.
- Hacemos clic en Apply y Apply and Close.

43.2.3 Tamaño y tipo fuente

Para cambiar las características de tamaño, tipo de fuente, debemos seguir la siguiente ruta:

Menu Window → Preferences → General → Appearance → Colors and fonts → Basic → Text Font.

Clic en el botón “Edit...” y configuramos las características de la fuente principal de Eclipse.

Una buena opción sería utilizar la fuente DejaVu Sans Mono de 14 puntos.

43.2.4 Colores. Syntax highlighting

Syntax highlighting es una característica que ofrecen muchos editores de código que permite mostrar en diferentes colores y tipos de fuente los distintos elementos del código. Esta característica facilita la escritura del programa. Los colores utilizados para los distintos elementos en Eclipse son tan vez poco diferenciados.

Si quieres modificarlos, basta con que sigas la siguiente ruta:

Menu Window → Preferences → PyDev → Editor:

Te sugerimos la siguiente configuración (a modo de ejemplo):

- Comments: 255,127,0
- Numbers: 192,0,0
- Matching brackets: 255,0,255
- Keywords: 0,128,192 bold
- Code: 0,67,191
- Function name: 128,0,255 bold
- Unicode: 51,160,44

Puedes crear la combinación que más se adapte a tu gusto personal.

43.2.5 Línea guía 80 caracteres

Es costumbre no utilizar más de ochenta caracteres por línea de código. En Eclipse podemos hacer visible una línea que nos indique esa anchura máxima. El procedimiento es:

Menu Window → Preferences → General → Editors → Text editors, activar el checkbox “Show print margin” y escribir el número deseado en la caja de texto “Print margin column:”.

También es posible seleccionar el color de dicha línea cambiando el color del elemento “Print margin” en la caja “Appearance color option” situada en la parte inferior.

44. Exportar e importar código a Eclipse –Anexo II

Los equipos del aula 122 se encuentran congelados. Eso quiere decir que cualquier proyecto que realices con Eclipse desaparecerá del disco duro una vez que apagues tu ordenador.

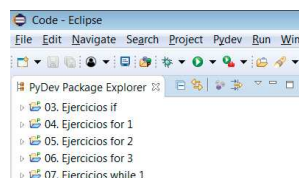
Para evitar este problema cada vez que vayamos a terminar nuestra sesión de trabajo realizaremos una exportación del proyecto en el que estemos trabajando a un archivo comprimido que guardaremos en nuestra carpeta compartida de Drive.

Cuando queramos volver a trabajar en un proyecto de un día anterior será necesario importarlo desde el archivo comprimido en que lo tengamos guardado.

Esta forma de operar nos permitirá también transportar nuestro código de forma sencilla entre cualquiera de los equipos en los vayamos a trabajar.

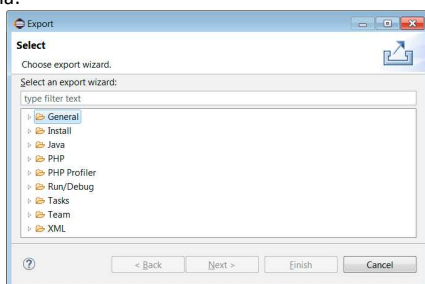
44.1 Exportar código Eclipse a un archivo comprimido.zip

Supón que tenemos nuestro Eclipse la siguiente estructura de proyectos:



Y que queremos exportar el proyecto correspondiente al título “03. Ejercicios if”.

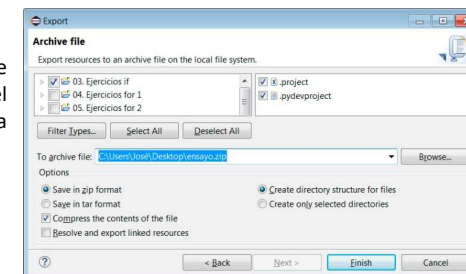
- En primer lugar haremos clic con el botón derecho del ratón sobre cualquier punto de la ventana PyDev Package Explorer (ventana izquierda), seleccionamos la opción “Export...”. Se mostrará la ventana:



- Abrimos la carpeta General y seleccionamos la opción Archive File. Hacemos clic en el botón Next>.
- Se mostrará una ventana que permite elegir los archivos y carpetas que queremos exportar. En este caso seleccionaremos el proyecto “03. Ejercicios if” haciendo clic en el selector situado a la izquierda de su nombre.
- Haciendo clic en el botón “Browse...” seleccionamos el directorio en el que guardaremos el archivo comprimido.
- En la ventana de texto situada a la izquierda del botón “Browse...” escribimos el nombre que queremos dar al archivo exportado:

- En este caso:

- Hacemos clic en el botón “Finish” y se creará el archivo comprimido en el directorio especificado. Sólo queda copiar este archivo a un lugar seguro.

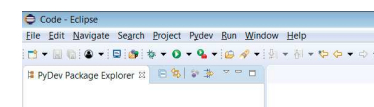


44.2 Importar código Eclipse desde un archivo comprimido.zip

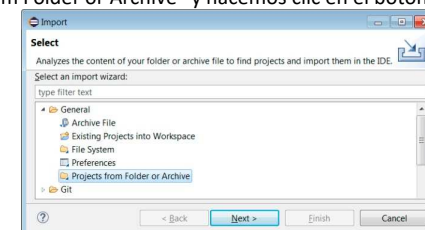
44.2.1 Archivos con un único proyecto en ellos

A modo de ejemplo importaremos el archivo comprimido en el apartado anterior a un Eclipse vacío. El procedimiento sería el mismo si hubiera algún proyecto preexistente.

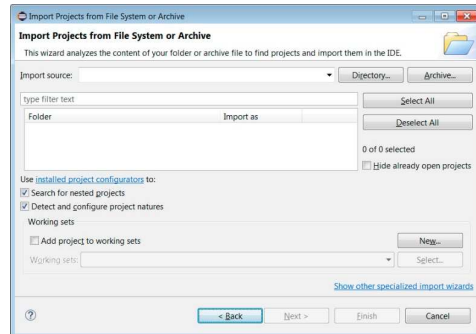
- Partiremos de la siguiente ventana:



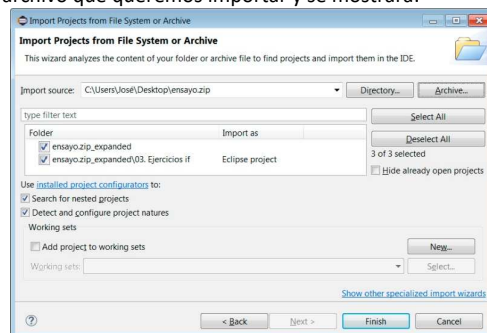
- Hacemos clic con el botón derecho en la ventana “PyDev Package Explorer” y seleccionamos la opción “Import...”. Se mostrará la ventana emergente “Import” en ella seleccionamos la opción “Projects from Folder or Archive” y hacemos clic en el botón “Next>”:



- Se mostrará la ventana “Import Projects from File System or Archive”, hacemos clic en el botón “Archive...” situado a la derecha de la ventana:



- Seleccionamos el archivo que queremos importar y se mostrará:

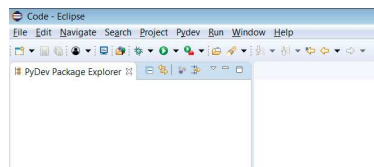


- Desactivamos el primer tick (ensayo.zip_expanded). Esto es algo que debemos hacer siempre, de lo contrario se importan dos copias del archivo aumentando la confusión.
- Hacemos clic en Finish y el proyecto estará listo para trabajar.

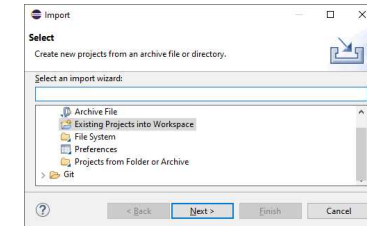
44.2.2 Archivos con más de un proyecto en ellos

Por ejemplo, los ejercicios que utilizan un proyecto externo con módulos en un programa principal que se conecta con ellos.

- Partiremos de la siguiente ventana:



- Hacemos clic con el botón derecho en la ventana "PyDev Package Explorer" y seleccionamos la opción "Import...". Se mostrará la ventana emergente "Import" en ella seleccionamos la opción "Existing Projects into Workspace" y hacemos clic en el botón "Next>":



- Se mostrará la ventana "Import Projects", hacemos clic en el selector "Select archive file:". A continuación, en el botón Browse...
- Esto nos permitirá buscar y seleccionar el archivo que queremos importar.
- En la ventana central se mostrará el nombre de todos los proyectos incluidos en el archivo.
- Será suficiente con seleccionarlos y hacer clic en el botón "Finish".

45. 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/>