

I.E.S. Arcipreste de Hita

Junio - 2024

Programación

1º CFGS DAW



Contenido

U. T. 1 Introducción	5
Origen de la Informática.....	6
Conceptos Informáticos Fundamentales.....	6
Clasificación del Software.....	6
Qué es la programación.....	7
Qué es un algoritmo	8
Documentación de los programas.....	16
Ciclo de Vida del Software.....	17
Ejercicios.....	18
Entregar	20
Ampliación.....	21
U. T. 2 Elementos de un programa informático.	22
Introducción	23
Historia de Python	23
Características de Python	24
Un vistazo rápido al interior de Python.....	25
Instalación de Python	26
Entornos IDE	26
Elementos de un programa	27
Ejercicios.....	44
U. T. 3 Estructuras de control.....	46
Introducción	47
Condiciones	47
Estructuras de selección o alternativas.....	47
Estructuras de repetición	48
Estructuras de salto	49
Prueba y depuración de programas	50
Documentación del código del programa	53
Cuestiones de Estilo.....	56
Programación modular.....	57
Ejercicios.....	62
U. T. 4 P.O.O.	67
Introducción	68
Orígenes.....	68
Características de la POO	69
Conceptos Fundamentales POO.....	70
Lenguajes.....	89
Ejercicios.....	90

Ejercicio (Repaso)	91
U. T. 5 POO (I).	92
Clases	93
Declaración de una clase	93
Estructura y miembros de una clase	95
Objetos	103
Visibilidad	104
Paquetes	107
Destrucción de objetos y Liberación de memoria.....	109
Librería estándar.....	110
Repaso	120
Ejercicios.....	124
U. T. 6 POO (II).	128
Tipos avanzados: Secuencias.....	129
Tipos avanzados: Listas.....	130
Otros tipos avanzados	133
Recursividad	136
Herencia.....	138
Excepciones	153
Estructuras de datos.....	158
Creación de casos de prueba.....	163
Ejercicios.....	166
U. T. 7 Proyecto.	175
Objetivo	176
Descripción del videojuego	176
Descripción de las etapas	176
Aleatoriedad de las preguntas.....	178
U. T. 8 Gestión de datos.	179
Gestión de ficheros.....	180
Bases de Datos.....	186
XML.....	189
Ejercicios.....	193
U. T. 9 Interfaces gráficos.	195
Introducción al diseño GUI	196
Diseño dirigido a eventos	198
Componentes	199
Librerías	199
Programación GUI Básica	200
Ejercicios.....	216

ANEXO I (Ahora qué)	218
ANEXO II (Bibliografía).....	219
ANEXO III (Nuevas versiones)	220
Python 3.10.....	220
Python 3.11.....	222
Python 3.12.....	222
ANEXO IV (F.A.Qs).....	225
POO avanzado	225
Iteradores	227
Varios.....	228
Cambio de librerías recomendados.....	231
ANEXO V (Diseño POO Robusto)	233
Qué es el software robusto	233
Ejemplo (Cap. 10 Head First Object-Oriented Analysis&Design – Brett D. McLaughlin)	238
ANEXO VI (Escribir código limpio)	244
Por qué de la necesidad de escribir código limpio	244
Nombrado.....	244
Diseño de funciones	245
Comentarios	246
Formato del código.....	246
Objetos y estructuras de datos.....	246
Control de errores	247
Creación de pruebas.....	247
Reglas del diseño sencillo	248
ANEXO VII (Patrones).....	249
Singleton.....	249
Fábrica	250
Director.....	251
Prototipo.....	253
Adaptador.....	254
Cadena	256
Estado	257
Observador / Observado	258
ANEXO VIII (Tecnologías Web)	260
ANEXO IX (Índice completo)	262
ANEXO X (Licencia)	275

U. T. 1 Introducción



Origen de la Informática

La historia del ordenador se remonta a las primeras reglas de cálculo y a las primeras máquinas diseñadas para facilitarle al ser humano la tarea de la aritmética. El ábaco, por ejemplo, fue un importante adelanto en la materia, creado alrededor de 4.000 a. C.

También hubo inventos muy posteriores, como la máquina de Blaise Pascal, conocida como Máquina de Pascal o Pascalina, creada en 1642. Consistía en una serie de engranajes que permitían realizar operaciones aritméticas. Esta máquina fue mejorada por Gottfried Leibnitz en 1671 y se dio inicio a la historia de las calculadoras.

Los intentos del ser humano por automatizar continuaron desde entonces: Joseph Marie Jacquard inventó en 1802 un sistema de tarjetas perforadas para intentar automatizar sus telares, y en 1822 el inglés Charles Babbage empleó dichas tarjetas para crear una máquina de cálculo diferencial.

Otro importante fundador en este proceso fue Alan Turing, creador de una máquina capaz de calcular cualquier cosa, y que llamó “máquina universal” o “máquina de Turing”. Las ideas que sirvieron para construirla fueron las mismas que luego dieron nacimiento al primer computador.

Otro importante avance fue el de ENIAC (Electronic Numeral Integrator and Calculator, o sea, Integrador y Calculador Electrónico Numeral), creado por dos profesores de la universidad de Pensilvania en 1943, considerado el abuelo de los computadores propiamente dicho. Consistía en 18.000 tubos de vacío que llenaban un cuarto entero.

Por último, la historia de los computadores no habría tenido el curso que tuvo sin la invención en 1947 de los transistores, fruto de los esfuerzos de los laboratorios Bell en Estados Unidos, dando origen a lo que conocemos hoy en día.

Conceptos Informáticos Fundamentales

La razón principal por la que una persona utiliza un ordenador es para resolver problemas (en el sentido más general de la palabra) o, en otras palabras, procesar una información para obtener un resultado a partir de unos datos de entrada.

Los ordenadores resuelven los problemas mediante la utilización de programas escritos por los programadores. Los programas de ordenador no son entonces más que métodos para resolver problemas. Por ello, para escribir un programa, lo primero es que el programador sepa resolver el problema que estamos tratando.

El programador debe identificar cuáles son los datos de entrada y a partir de ellos obtener los datos de salida, es decir, la solución, a la que se llegará por medio del procesamiento de la información que se realizará mediante la utilización de un método para resolver el problema denominada algoritmo.

Clasificación del Software

El software es elemento que desarrollan los programadores y se puede clasificar atendiendo a diversas características, pero en un curso de programación haremos dicha clasificación teniendo en cuenta la funcionalidad general en el sistema:

- ∞ **Software de sistema.** Elementos que permiten el mantenimiento del sistema en global: sistemas operativos, controladores de dispositivos, servidores, utilidades, herramientas de diagnóstico, de corrección y optimización, generalmente se usa el lenguaje C para dicha programación con parte en código ensamblador.
- ∞ **Software de programación.** Diferentes alternativas y lenguajes para desarrollar programas de informática: editores de texto, compiladores, intérpretes, enlazadores, depuradores, entornos de desarrollo integrados (IDE).

- ∞ **Software de aplicación.** Permite a los usuarios llevar a cabo una o varias tareas específicas en cualquier campo de actividad: aplicaciones ofimáticas, para control de sistemas y automatización industrial, software educativo, software empresarial, bases de datos, telecomunicaciones (Internet), videojuegos, software médico, software de diseño asistido (CAD), software de control numérico (CAM).

Qué es la programación

Concepto

La programación es el proceso mediante el cual se diseña y codifica un algoritmo mediante un conjunto de instrucciones siguiendo una sintaxis concreta (lenguaje). Este proceso no es sencillo ni rápido y tenemos que ser muy cuidadosos para que el resultado final sea un producto “aceptable”.

La creación de un algoritmo se basa en la resolución de problemas reales por nosotros. Generalmente ante un problema, lo primero que hacemos es observarlo para determinar todos los factores que intervienen, a continuación, desarrollamos un conjunto de soluciones que nos parecen factibles para resolverlo, siendo la última etapa la implementación (llevar a cabo) una de las soluciones para resolver el problema planteado, así como realizaremos pruebas para comprobar su correcto funcionamiento.

La programación ha evolucionado a lo largo de los años, presentando diversas maneras de llevarse a cabo, creando lo que se llaman paradigmas de la programación. Estos paradigmas se siguen utilizando hoy en día y dependiendo del proyecto que estemos desarrollando se usarán unos u otros o mezcla.

Paradigmas de la programación

Como hemos mencionado anteriormente, los paradigmas son reglas aceptadas de programación que han ido evolucionando paralelas a la tecnología y que utilizaremos según el proyecto que desarrollemos.

- ∞ **Programación estructurada.** La programación estructurada se basa en utilizar solamente tres tipos de estructuras de control en nuestros programas. En concreto se usarán solo estructuras secuenciales, alternativas e iterativas. Se evitarán completamente el uso de sentencias de salto incondicionales. ¿Qué es una secuencia de control? Una regla del lenguaje.
 - **Secuencias:** Ejecución de una instrucción a continuación de otra.
 - **Bucles o iterativas:** Repetición de un conjunto de secuencias un número de veces determinado o mientras que se cumpla la condición.
 - **Alternativas o condicionales.** Ejecución de unas secuencias u otras dependiendo de una condición.
- ∞ **Programación modular.** Este paradigma implica la descomposición del problema en varios sub-problemas menos complejos que se puedan implementar más fácilmente. Además, se incluye la posibilidad que los sub-problemas se comuniquen entre sí traspasando datos de unos a otros.
La programación modular se crea desde dos perspectivas, de arriba abajo o viceversa. En la primera aproximación el problema de partida es de gran tamaño y se va dividiendo en más pequeños hasta llegar a problemas unitarios. En la segunda aproximación se crean soluciones para problemas pequeños que se van uniendo para ir creando soluciones mayores hasta dar con la solución completa. Los elementos constituyentes de este paradigma son las funciones y procedimientos, clases y métodos a bajo nivel; unidades, paquetes y librerías a alto nivel.
La programación modular se basa a más bajo nivel en funciones y para crear una buena función tendremos en cuenta los siguientes puntos:
 - Un único punto de entrada y de salida.
 - La función se comportará como una caja negra.
 - El tamaño orientativo estará entre 30 y 50 líneas.
 - Tendrá máxima cohesión o relación con los elementos de la misma unidad funcional.
 - Tendrá mínimo acoplamiento o dependencia de las demás funciones.
- ∞ **Programación orientada a objetos (POO).** Este paradigma se estudiará en una unidad de trabajo posterior. Está basada en los procesos de ingeniería tradicional.

Qué es un algoritmo

Por algoritmo entendemos un conjunto ordenado y finito de operaciones matemáticas que permiten resolver un problema con entradas y salidas definidas que además cumplen las siguientes características:

- ∞ Tiene un número finito de pasos
- ∞ Acaba en un tiempo finito. Si no acabase nunca, no se resolvería el problema.
- ∞ Todas las operaciones deben estar definidas de forma precisa y sin ambigüedad.
- ∞ Puede tener varios datos de entrada y de salida.

Un claro ejemplo de algoritmo es una receta de cocina, donde tenemos unos pasos que hay que seguir en un orden y deben de estar bien definidos, tiene un tiempo finito y tiene unos datos de entrada (ingredientes) y una salida (el plato). Por ejemplo, el algoritmo para freír un huevo podría ser el siguiente:

- ∞ Datos de entrada: Huevo, aceite, sartén, fuego.
- ∞ Datos de salida: huevo frito.

Procedimiento:

1. Poner el aceite en la sartén.
2. Poner la sartén al fuego.
3. Cuando el aceite esté caliente, cascar el huevo e introducirlo.
4. Cubrir el huevo de aceite.
5. Cuando el huevo esté hecho, retirarlo.

La codificación de un algoritmo en un ordenador se denomina programa

Un algoritmo debe cumplir una serie de requerimientos para ser funcional:

- ∞ ¿Produce los resultados esperados?
- ∞ ¿Es la solución óptima?
- ∞ ¿Cómo va a funcionar el algoritmo si incrementamos el número de datos?

Estas preguntas nos las haremos cada vez que hayamos terminado un algoritmo para asegurar la funcionalidad.

Los algoritmos que desarrollemos se engloban según las características del problema en: algoritmos que hacen uso intenso de los datos, algoritmos que hacen uso intenso del procesador y en algoritmos mezcla de los dos anteriores. Esta clasificación nos da una aproximación a los problemas que nos encontraremos a la hora de crearlo.

Representación de los algoritmos

Uno de los principales problemas que nos encontramos a lo hora de crear una aplicación, es que el programador poco experimentado cree que su herramienta principal de trabajo es el ordenador, y se lanza a crear código inmediatamente frente a la pantalla. Esta concepción nos lleva a que el aprendizaje de la programación no se hace de forma correcta y crea vicios muy difíciles de subsanar después.

Si revisamos el ciclo de vida estándar que se muestra más adelante, comprobaremos que solo una de las fases es de codificación, las otras cuatro no, por lo que hay que desbarcar la idea de que la programación se aprende exclusivamente delante de la pantalla programando, la programación se aprende primero planificando y comprendiendo lo que queremos hacer y después codificando, por lo que emerge la necesidad de representar de forma gráfica nuestros algoritmos antes de la codificación.

La codificación primero se planifica en papel y después se pasa al ordenador

Elementos de los algoritmos

- Datos: expresión general que describe con los objetos con los que opera un algoritmo: números (enteros, reales, lógicos, carácter, cadena).

- Constantes: Valores que no cambian a lo largo del programa, almacena un dato.
- Variables: Un objeto que puede cambiar con el desarrollo del programa, almacena un dato.
- Expresiones: combinación de operadores y operandos. Los operadores pueden ser de diversos tipos: aritméticos, lógicos, etc. Los operandos serán datos, constantes o variables.
- **Palabras reservadas o cajas de acción.** Representan el flujo de movimiento del programa.

Diagramas de flujo (ordinogramas)

Un diagrama de flujo es una representación gráfica del algoritmo. Esta representación siempre debe seguir unas reglas básicas y usar unos símbolos predefinidos para poder entenderlos.

- ∞ El comienzo del programa figurará en la parte superior del ordinograma y la finalización abajo.
- ∞ El símbolo de comienzo y de fin deberá aparecer una sola vez en el ordinograma.
- ∞ El flujo de las operaciones será, siempre que sea posible de arriba a abajo y de izquierda a derecha.
- ∞ Se evitarán siempre los cruces de líneas utilizando conectores.
- ∞ Se podrá dividir el algoritmo entre diversas hojas usando conectores de unión.



La herramienta **Dia** es muy útil para crear este tipo de diagramas

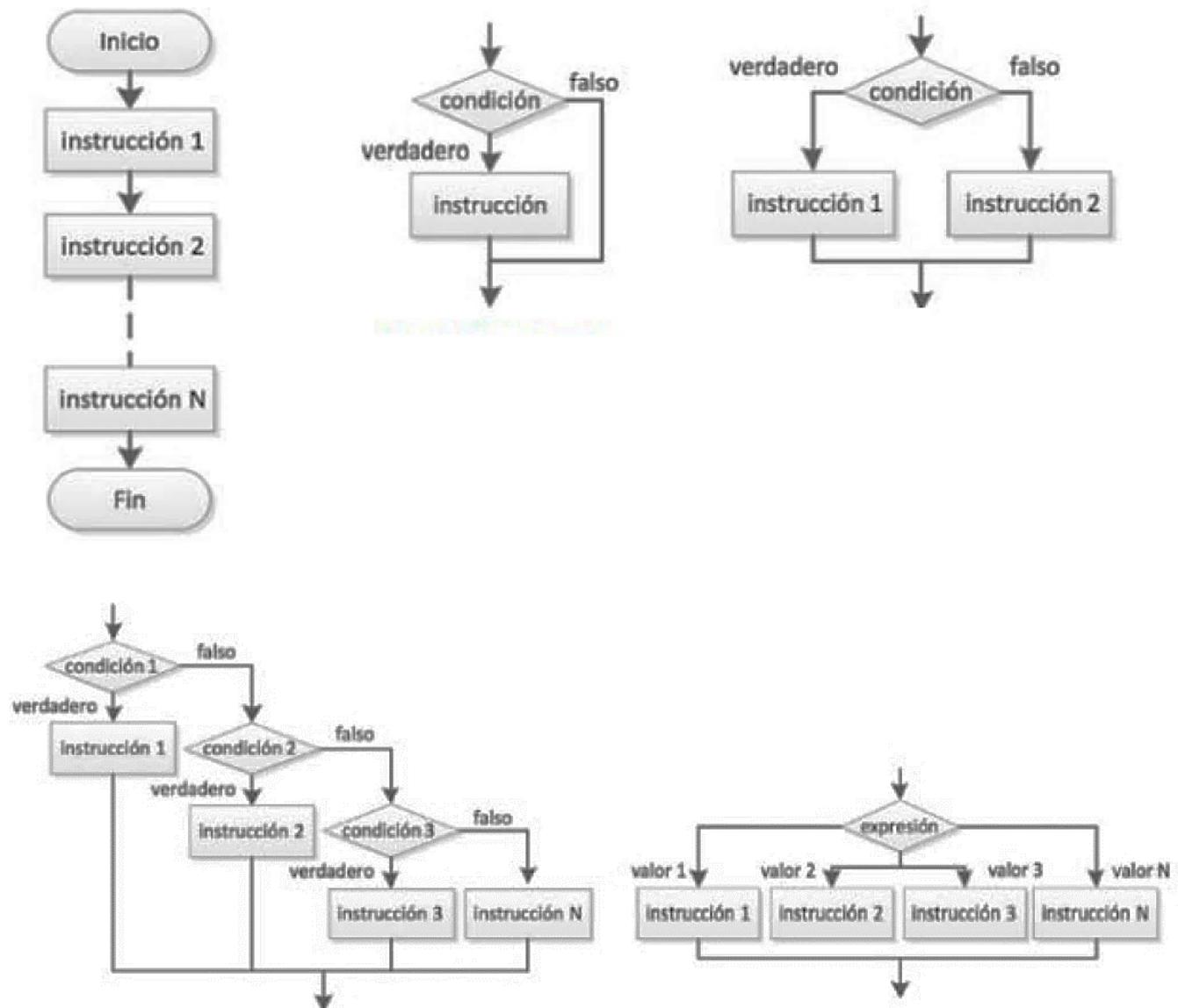
Símbolos de los diagramas de flujo

SÍMBOLO	SIGNIFICADO	SÍMBOLO	SIGNIFICADO
	Terminal. Indica el inicio o la terminación del flujo del proceso		Actividad. Representa una actividad llevada a cabo en el proceso.
	Decisión. Indica un punto en el flujo en que se produce una bifurcación del tipo "SÍ" – "NO"		Documento. Se refiere a un documento utilizado en el proceso, se utilice, se genere o salga del proceso.
	Multidocumento. Refiere a un conjunto de documentos. Por ejemplo, un expediente que agrupa distintos documentos.		Inspección/ firma. Empleado para aquellas acciones que requieren supervisión (como una firma o "visto bueno")
	Base de datos/ aplicación. Empleado para representar la grabación de datos.		Línea de flujo. Proporciona una indicación sobre el sentido de flujo del proceso.

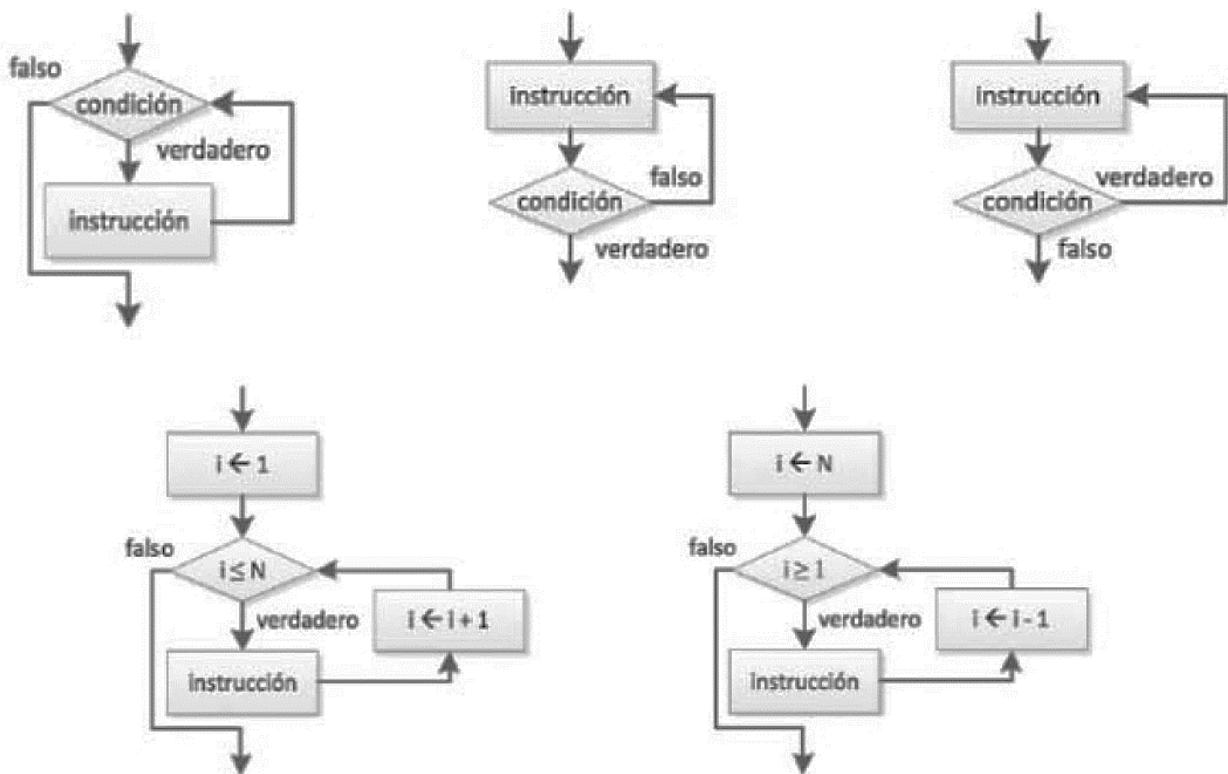
Símbolos básicos



Condicionales



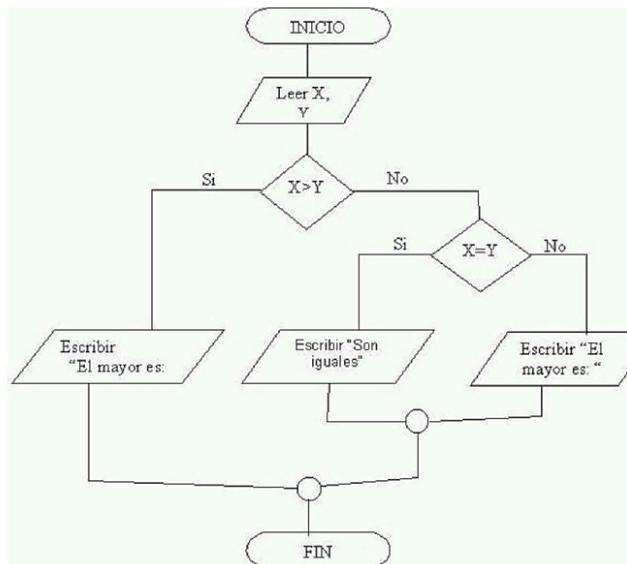
Bucles



Inicio y llamadas a procedimientos



Ejemplo: Algoritmo que lee dos números “X” e “Y”, determina si son iguales, y en caso de no serlo, indica cuál de ellos es el mayor.



Pseudocódigo

Esta técnica de representación utiliza un lenguaje muy cercano a las sintaxis de los lenguajes reales de programación para definir un algoritmo, siendo el paso a la codificación muy sencilla. Se basa en las siguientes estructuras.

- ∞ Operadores matemáticos: +, -, *, /
- ∞ Operadores de comparación: <, <=, >, >=, <>, =
- ∞ Operadores lógicos: and, or, not
- ∞ Operadores de asignación: \leftarrow
- ∞ Estructuras del lenguaje.
 - Inicio y fin: Inicio | Fin.
 - Repetitivas: Para | Desde | Mientras.
 - Condicionales: Si | Sino | Entonces | Según | Caso | Otro caso
 - De entrada / Salida: Leer | Imprimir.
 - Finalización: Fin otra_etiqueta.

Todo algoritmo empieza por la palabra claves Inicio y finaliza por la palabra clave Fin. El algoritmo siempre empieza por arriba y termina por abajo. En su interior utilizaremos tantas palabras claves en secuencia y anidándolas como necesitemos para crear nuestro algoritmo respetando la sintaxis. Las entradas y salidas se realizarán mediante las instrucciones correspondientes. Cada vez que iniciemos un bloque se sangrarán todas las líneas que estén dentro del mismo con cuatro espacios en blanco.

Ejemplo: Crear un algoritmo para sumar dos números pedidos por teclado.

```

Inicio
  Leer(numero_uno)
  Leer(numero_dos)
  suma  $\leftarrow$  numero_uno + numero_dos
  imprimir(suma)
Fin
  
```

Sintaxis del pseudocódigo**Secuencia**

 Inicio
 <instrucción_1>
 <instrucción_2>
 ...
 <instrucción_n>
 Fin

Condicional

 Inicio
 si <condición> entonces
 <instrucción_1>
 sino
 <instrucción_2>
 Fin si
 Fin

Condicional

 Inicio
 según <expresión>
 caso <valor>
 <instrucción_1>
 caso <valor>
 <instrucción_2>
 ...
 caso <valor>
 <instrucción_n>
 otro
 <instrucción_m>
 Fin según
 Fin

Repetitiva

 Inicio
 mientras <condición> hacer
 <instrucción_1>
 ...
 fin mientras
 Fin

Repetitiva

 Inicio
 repetir
 <instrucción_1>
 ...
 hasta que <condición>
 fin repetir
 Fin

Repetitiva

 Inicio
 para n←valor hasta valor incremento
 hacer
 <instrucción_1>
 ...
 rin para
 Fin

Ejemplo: Crear un algoritmo para determinar si un número es par o no.

```

    Inicio
        imprimir("Introduzca un número:")
        leer(numero)
        si (numero % 2) == 0 //% indica el módulo, o resto de la división
            imprimir("par")
        sino
            imprimir("impar")
        fin si
    Fin

```

Ejemplo: Crear un algoritmo para determinar la suma, resta, multiplicación y división de dos números pedidos al usuario teniendo en cuenta que no se pueden dividir números entre cero.

```

    Inicio
        imprimir("Introduce el primer número:")
        Leer(num1)
        imprimir("Introduce el segundo número")
        leer(num2)
        Resultado←0
        Resultado←num1+num2
        imprimir(resultado)
        resultado←num1-num2
        imprimir(resultado)
        resultado←num1*num2
        imprimir(resultado)
        si num2==0 entonces
            imprimir("La división entre num1 y num2 es infinito")
        si no
            resultado←num1/num2
            imprimir(resultado)
        Fin si
    Fin

```

Rendimiento de los algoritmos

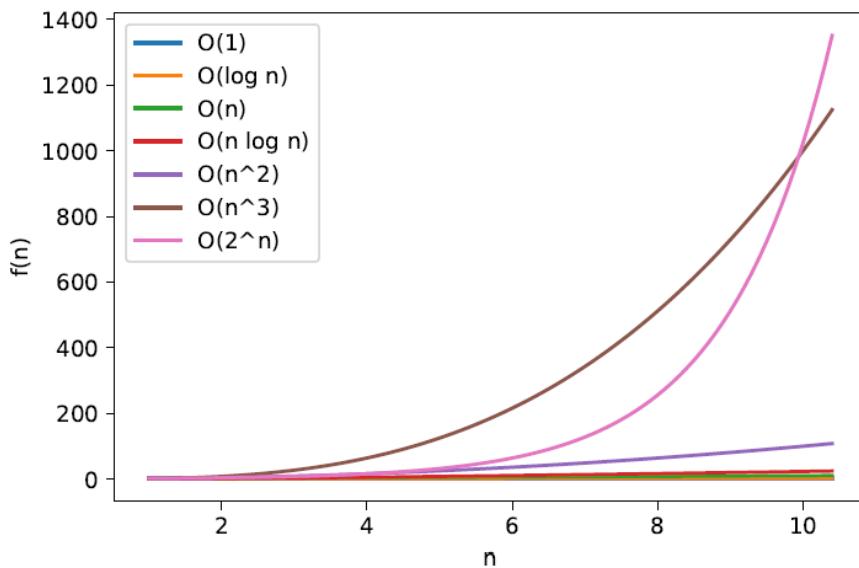
Si queremos crear algoritmo óptimos y eficientes tendremos que tener métricas (indicadores que sean medidos) que nos aporten la información suficiente para determinar el rendimiento. Pero para conseguir estos objetivos, los algoritmos deben ser correctos, deben ser entendibles y deben ser eficientes.

Para determinar la eficiencia hemos dicho que es necesario medir, y las mediciones se pueden llevar a cabo en dos ámbitos: En el ámbito del espacio y en el ámbito del tiempo. En el espacio estimaremos los recursos de almacenamiento necesarios (disco y memoria) para el funcionamiento del programa. En el ámbito del tiempo intentaremos encontrar una medida que nos indique cómo se comportaría el algoritmo en el tiempo y en función de la cantidad de datos. Para esta última medida solemos usar el orden de complejidad de un algoritmo antes de implementarlo y realizamos mediciones en ejecución (**profile**) del tiempo real que tarda en ejecutarse.

Órdenes de complejidad

El orden de complejidad es una expresión matemática que indica cómo se comportará un algoritmo si la cantidad de datos aumenta. Esta expresión es del tipo $O(x)$, en la que x es una función que indicará dicho comportamiento. Así nos encontraremos con órdenes constantes $O(1)$, órdenes lineales $O(n)$, órdenes polinómicos $O(n^x)$, órdenes logarítmicos $O(n \log(n))$ $O(\log(n))$, etc.

Comprenderemos mucho mejor lo que significa si vemos las gráficas correspondientes.



Si interpretamos los valores del eje x como incrementos en la cantidad de datos y el eje y el tiempo que tarda en realizar la tarea, lo ideal es que esté por debajo de la función constante (en azul en la gráfica) o se acerque lo más posible a ella. Como vemos los únicos órdenes de complejidad que se acercan son los órdenes logarítmicos.

No estudiaremos cómo determinar el orden de complejidad de un algoritmo, pero debemos de saber elegir entre varios algoritmos si nos dan un orden de complejidad para cada uno de ellos.

Codificación y ejecución

Al final, los algoritmos tienen que estar implementados en un lenguaje y la elección de este dependerá de varios factores, entre los que destacan el utilizado en el proyecto en el que estemos asignados o por una elección propia.

El proceso de codificación se realiza a través de herramientas de edición de texto simples o complejas (IDEs) con las que se facilita la tarea del programador. En estos entornos se lleva a cabo todo el proceso de creación de los algoritmos: codificación, prueba, empaquetado y distribución.

Una vez codificado el programa se tiene que compilar (pasar al lenguaje del ordenador) y ejecutar.

La compilación se realiza a través de programas especializados (compiladores o intérpretes), la ejecución es el Sistema Operativo. el encargado de realizarla.

Clasificación de los lenguajes

Los lenguajes se pueden clasificar según atendiendo a varias características:

- o **Según el código:** binario, ensamblador, alto nivel. Esta clasificación a la cercanía del lenguaje usado al microprocesador. Cuanto más cerca esté el lenguaje de la máquina más difícil será su uso y la programación, pero más eficiente a su vez será. Hay ciertas partes de los sistemas operativos que deben ser programados en lenguaje ensamblador para ser eficientes.

Del mismo modo, los lenguajes de más alto nivel son más fáciles de usar y presentan más herramientas al programador a cambio de una menor eficiencia.

En última instancia, independientemente del lenguaje utilizado, el algoritmo debe ser “traducido” a código binario (código máquina) para que se pueda ejecutar, esa transformación se llama **compilación**.

- ∞ **Según la ejecución:** compilados, interpretados o mixtos. Una vez programado el algoritmo se tiene que ejecutar dentro de un microprocesador y este solo entiende ceros y unos (código binario o código máquina), por lo que es necesaria la traducción a ese lenguaje. Este proceso de traducción se puede hacer desde varios puntos de vista.

Compilado: El código fuente se compila directamente en un fichero ejecutable ya en código máquina. Esta aproximación es la más rápida de las tres, pero el ejecutable es dependiente de la arquitectura y del sistema operativo para el que se haya compilado.

Interpretado: Un programa intermedio lee el código fuente, lo traduce a código máquina y lo ejecuta.

Intermedio o mixto: El código fuente se lee a un código intermedio llamado código objeto (**byte code**) en un lenguaje propio, posteriormente un segundo programa llamado intérprete leerá el byte code y lo traducirá a código máquina justo antes de ejecutarse. Esta aproximación pierde eficiencia, pero gana en portabilidad.



Documentación de los programas

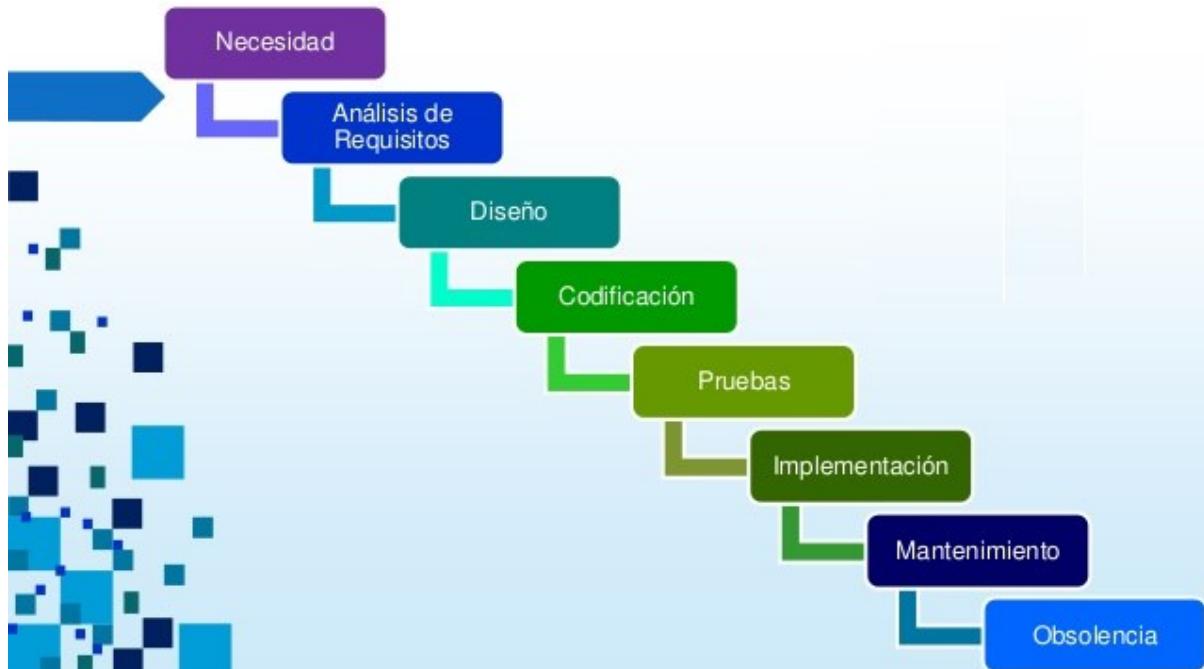
La mayor parte de los proyectos exigen la realización de una planificación previa. Esta planificación debe determinar el modelo de ciclo de vida a seguir, los plazos para completar cada fase y los recursos necesarios en cada momento. Todo esto se debe plasmar en una documentación completa y detallada de toda la aplicación.

La documentación asociada al software puede clasificarse en interna y externa. La documentación interna corresponde a la que se incluye dentro del código fuente de los programas. Nos aclaran aspectos de las propias instrucciones del programa. La documentación externa es la que corresponde a todos los documentos relativos al diseño de la aplicación, a la descripción de la misma y sus módulos correspondientes, a los manuales de usuario y los manuales de mantenimiento.

Ciclo de Vida del Software

Cuando hablamos de ciclo de vida del software nos referimos a las fases por las que pasa el desarrollo de una aplicación o programa desde su concepción hasta su sustitución. Ciclos de vida hay muchos, pero nos vamos a centrar el ciclo clásico que incluye las siguientes fases:

- 1º. Análisis. En la fase de análisis recogeremos información sobre el funcionamiento del programa y sus diferentes partes, veremos los actores que están implicados y las entradas salidas de datos que necesitaremos para llevar a cabo el desarrollo.
- 2º. Diseño. En el diseño crearemos las estructuras visuales y de datos necesarias para llevar a cabo la tarea, plasmaremos en documentos la estructura de la aplicación, las librerías y los datos que van a compartir, si es necesario se determinarán las estructuras de los flujos de comunicación, así como de todas las estructuras de datos. Definiremos normas de codificación y estándares para nombrado y almacenamiento, se crearán guían de estilo.
- 3º. Codificación y pruebas. En esta fase se realiza la codificación de la aplicación, se divide el trabajo en grupos y se realiza en paralelo, avanzaremos desde lo más sencillo a lo más complejo, creando pruebas para cada bloque de código que creamos. Nos aseguraremos que todas las partes funcionan y encajan perfectamente entre ellas.
- 4º. Implementación o implementación. Se pasa en producción la aplicación, se instala en los servidores y se hacen pruebas de integración, se forma a los usuarios y comienza el uso de la aplicación.
- 5º. Mantenimiento. En esta fase repararemos aquellos errores que sean notificados por los usuarios generando nuevas versiones de la aplicación, así como daremos asistencia a los usuarios sobre nuestra aplicación. El mantenimiento hará que nuestro software termine por deteriorarse y que tenga que ser sustituido por una nueva versión desarrollada por completo que nos llevará a la primera fase de nuevo.



Ejercicios

1. Escribe un algoritmo para cambiar la rueda de un coche.
2. Escribe un algoritmo para cocinar un plato de pasta.
3. Dadas dos variables numéricas A y B, que el usuario debe teclear, se pide realizar un algoritmo que intercambie los valores de ambas variables y muestre cuánto valen al final las dos variables.
4. Algoritmo que lea dos números y nos diga cuál de ellos es mayor o bien si son iguales.
5. Algoritmo que lea tres números distintos y nos diga cuál de ellos es el mayor.
6. Hacer un algoritmo que imprima los números pares entre 0 y un número solicitado por pantalla.
7. Un colegio desea saber qué porcentaje de niños y qué porcentaje de niñas hay en el curso actual. Diseñar un algoritmo para este propósito.
8. Una tienda ofrece un descuento del 15% sobre el total de la compra durante el mes de octubre. Dado un mes y un importe, calcular cuál es la cantidad que se debe cobrar al cliente.
9. Escribir un algoritmo que pida al usuario que escriba N o S y lo repita hasta que el usuario pulse otra letra. Debe funcionar tanto en minúsculas como en mayúsculas.
10. Hacer un algoritmo que pida una cadena, la imprima en mayúsculas, después en minúsculas, nos diga la longitud y escriba de una en una sus letras. Cada vez que termina debe preguntar al usuario si desea introducir otra.
11. Hacer un algoritmo que cuente las veces que aparece una determinada letra en una frase que introduciremos por teclado y nos pregunte si deseamos continuar.
12. Desarrollar un algoritmo que nos calcule el cuadrado de los 9 primeros números naturales.
13. Introducir dos números por teclado y mediante un menú, calcule su suma, su resta, su multiplicación o su división, finalizará al pulsar un valor de salida ('s').
14. Teniendo en cuenta que la clave es "eureka", escribir un algoritmo que nos pida una clave. Solo tenemos 3 intentos para acertar, si fallamos los 3 intentos nos mostrará un mensaje indicándonos que hemos agotado esos 3 intentos. Si acertamos la clave, saldremos directamente del programa.
15. Se ha establecido un programa para estimular a los alumnos, el cual consiste en lo siguiente: si el promedio global obtenido por un alumno en el último periodo es mayor o igual que 5, se le hará un descuento del 30% sobre la matrícula y no se le cobrará IVA; si el promedio obtenido es menor que 5 deberá pagar la matrícula completa, la cual debe incluir el 10% de IVA. Hacer un algoritmo que calcule el valor a pagar si se conocen las notas finales de las 6 materias que cursaron.
16. Hacer un algoritmo para ayudar a un trabajador a saber cuál será su sueldo semanal. Se sabe que, si trabaja 40 horas o menos, se le pagará 20€ por hora, pero si trabaja más de 40 horas entonces las horas extras se le pagarán a 25€ por hora.
17. Realiza un reloj digital que pida la hora, minutos y segundos en los que tiene que empezar y a partir de ahí muestre el resto de horas, minutos y segundos.
18. Realizar un juego simple que pide al usuario que adivine un número en 10 intentos.
19. La siguiente tabla muestra un algoritmo paso a paso (lista de instrucciones). Utiliza tres variables A, B y C que inicialmente valen 4, 2 y 3 respectivamente. Calcula el valor de las variables tras ejecutar cada instrucción. Las tres primeras están hechas a modo de ejemplo.

	Instrucción	A	B	C
1	$A \leftarrow B$			
2	$C \leftarrow A$			
3	$B \leftarrow (A + B + C) / 2$			
4	$A \leftarrow A + C$			

Recuerda que $X = Y$ significa que el valor de Y se copia en X.

5	$C \leftarrow B - A$			
6	$C \leftarrow C - A$			
7	$A \leftarrow A * B$			
8	$A \leftarrow A + 3$			
9	$A \leftarrow A \% B$			
10	$C \leftarrow C + A$			

20. Evalúa las siguientes expresiones:

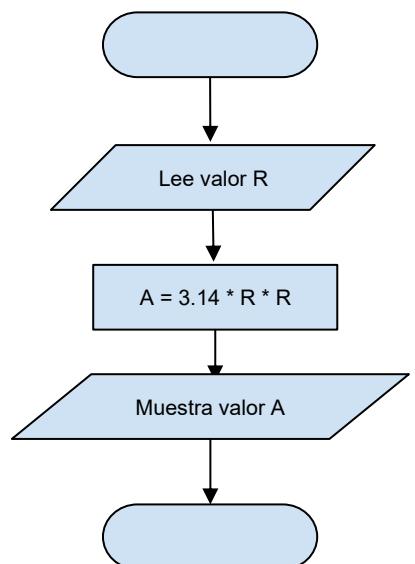
$((3 + 2)^2 - 15) / 2 * 5$	$5 - 2 > 4 \text{ AND NOT } 0.5 == 1 / 2$
Dado $x = 1, y = 4, z = 10, \pi = 3.14, e = 2.71$ $2 * x + 0.5 + y - 1 / 5 * z$	Dado $x = 1, y = 4, z = 10, \pi = 3.14, e = 2.71$ $(\pi * x ^ 2 > y) \text{ OR } (2 * \pi * x \leq z)$

21. Evalúa las siguientes expresiones:

1. $24 \% 5$
2. $7 / 2 + 2.5$
3. $10.8 / 2 + 2$
4. $(4 + 6) * 3 + 2 * (5 - 1)$
5. $5 / 2 + 17 \% 3$
6. $7 \geq 5 \text{ OR } 27 \neq 8$
7. $(45 \leq 7) \text{ OR NOT } (5 \geq 7)$
8. $27 \% 4 + 15 / 4$
9. $37 / 4 * 4 - 2$
10. $(25 \geq 7) \text{ AND NOT } (7 \leq 2)$
11. $('H' < 'J') \text{ AND } ('9' \neq '7')$
12. $25 > 20 \text{ AND } 13 > 5$
13. $10 + 4 < 15 - 3 \text{ OR } 2 * 5 + 1 > 14 - 2 * 2$
14. $4 * 2 \leq 8 \text{ OR } 2 * 2 < 5 \text{ AND } 4 > 3 + 1$
15. $10 \leq 2 * 5 \text{ AND } 3 < 4 \text{ OR NOT } (8 > 7) \text{ AND } 3 * 2 \leq 4 * 2 - 1$

22. Dado el siguiente algoritmo descrito en forma de ordinograma, explica brevemente qué hace y cuál sería el resultado mostrado si el valor R leído fuera 2.

23. Dibuja un ordinograma que dé los “buenos días”.
24. Dibuja un ordinograma que calcule y muestre el área de un cuadrado de lado igual a 5.
25. Dibuja un ordinograma que calcule el área de un cuadrado cuyo lado se introduce por teclado.
26. Dibuja un ordinograma que lea dos números, calcule y muestre el valor de sus suma, resta, producto y división.
27. Dibuja un ordinograma que toma como dato de entrada un número que corresponde a la longitud de un radio y nos escribe la longitud de la circunferencia, el área del círculo y el volumen de la esfera que corresponden con dicho radio.
28. Dibuja un ordinograma que dado el precio de un artículo y el precio de venta real nos muestre el porcentaje de descuento realizado.
29. Dibuja un ordinograma que lea un valor correspondiente a una distancia en millas marinas y escriba la distancia en metros. Sabiendo que una milla marina equivale a 1.852 metros.
30. Dibuja un ordinograma de un programa que pide la edad por teclado y nos muestra el mensaje de “Eres mayor de edad” solo si lo somos.



31. Dibuja un ordinograma de un programa que pide la edad por teclado y nos muestra el mensaje de “eres mayor de edad” o el mensaje de “eres menor de edad”.
32. Dibuja un ordinograma que lee dos números, calcula y muestra el valor de su suma, resta, producto y división. (Ten en cuenta la división por cero).
33. Dibuja el ordinograma de un programa que lee 2 números y muestra el mayor.
34. Dibuja el ordinograma y escribe el pseudocódigo de un programa que lee un número y me dice si es positivo o negativo, consideraremos el cero como positivo.
35. Dibuja el ordinograma y escribe el pseudocódigo de un programa que lee dos números y los visualiza en orden ascendente.
36. Escribe el pseudocódigo que lea una calificación numérica entre 0 y 10 y la transforma en calificación alfabética, escribiendo el resultado.
 - de 0 a <3 Muy Deficiente.
 - de 3 a <5 Insuficiente.
 - de 5 a <6 Bien.
 - de 6 a <9 Notable
 - de 9 a 10 Sobresaliente
37. Dibuja un ordinograma que recibe como datos de entrada una hora expresada en horas, minutos y segundos que nos calcula y escribe la hora, minutos y segundos que serán, transcurrido un segundo.
38. Dibuja un ordinograma y escribe un pseudocódigo que calcula el salario neto semanal de un trabajador en función del número de horas trabajadas y la tasa de impuestos de acuerdo a las siguientes hipótesis:
 - Las primeras 35 horas se pagan a tarifa normal.
 - Las horas que pasen de 35 se pagan a 1,5 veces la tarifa normal.
 - Las tasas de impuestos son:
 - Los primeros 500 euros son libres de impuestos.
 - Los siguientes 400 tienen un 25% de impuestos.
 - Los restantes un 45% de impuestos.Escribir nombre, salario bruto, tasas y salario neto.
39. Dibuja un ordinograma y escribe un pseudocódigo de un programa que muestre los números pares comprendidos entre el 1 y el 200. Utiliza un contador sumando de 1 en 1.
40. Dibuja un ordinograma y escribe un pseudocódigo de un programa que muestre los números desde el 1 hasta un número N que se introducirá por teclado.
41. Dibuja un ordinograma y escribe un pseudocódigo de un programa que lea un número positivo N y calcule y visualice su factorial N!. Siendo la factorial:
 - $0! = 1$
 - $1! = 1$
 - $2! = 2 * 1$
 - $3! = 3 * 2 * 1$
 - $N! = N * (N-1) * (N-2) * \dots * 1$
42. Dibuja un ordinograma y escribe un pseudocódigo de un programa que lea 100 números no nulos y luego muestre un mensaje indicando cuántos son positivos y cuantos negativos.
43. Dibuja un ordinograma y escribe un pseudocódigo de un programa que lea una secuencia de números no nulos hasta que se introduzca un 0, y luego muestre si ha leído algún número negativo, cuantos positivos y cuantos negativos.
44. Dibuja un ordinograma y escribe un pseudocódigo de un programa que calcula y escribe la suma y el producto de los 10 primeros números naturales.

Entregar

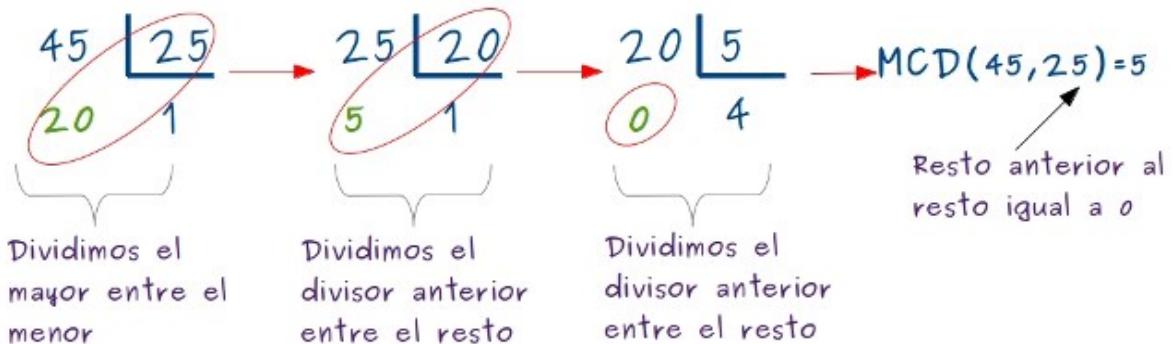
45. Dibuja un ordinograma y escribe un pseudocódigo de un programa que lee una secuencia de notas (con valores que van de 0 a 10) que termina con el valor -1 y nos dice si hubo o no alguna nota con valor 10.
46. Dibuja un ordinograma y escribe un pseudocódigo de un programa que suma independientemente los pares y los impares de los números comprendidos entre 100 y 200, y luego muestra por pantalla ambas sumas.

47. Dibuja un ordinograma y escribe un pseudocódigo de un programa que calcule el valor A elevado a B (A^B) sin hacer uso del operador de potencia (^), siendo A y B valores introducidos por teclado, y luego muestre el resultado por pantalla.
48. Dibuja un ordinograma y escribe un pseudocódigo de un programa donde el usuario "piensa" un número del 1 al 100 y el ordenador intenta adivinarlo. Es decir, el ordenador irá proponiendo números una y otra vez hasta adivinarlo (el usuario deberá indicarle al ordenador si es mayor, menor o igual al número que ha pensado).
49. Dibuja un ordinograma y escribe un pseudocódigo de un programa que dada una cantidad de euros que el usuario introduce por teclado (múltiplo de 5 €) mostrará los billetes de cada tipo que serán necesarios para alcanzar dicha cantidad (utilizando billetes de 500, 200, 100, 50, 20, 10 y 5). Hay que indicar el mínimo de billetes posible. Por ejemplo, si el usuario introduce 145 el programa indicará que será necesario 1 billete de 100 €, 2 billetes de 20 € y 1 billete de 5 € (no será válido por ejemplo 29 billetes de 5 que, aunque sume 145 € no es el mínimo número de billetes posible).

Ampliación

- Implementar el algoritmo de Euclides para el cálculo del máximo común divisor.

Calculamos $MCD(45,25)$ usando algoritmo de Euclides



- Implementar un ordinograma que escriba en la pantalla tantos signos # como el valor del cociente de sus dos argumentos, y tantos signos \$ como el resto de la división entera.

U. T. 2 Elementos de un programa informático.



Introducción

La elección de un lenguaje de programación para aprender los entresijos de la programación es una opción muy importante y que no se debe tomar a la ligera. Tradicionalmente se usaban lenguajes muy estructurados para el aprendizaje como era Pascal o Basic, pero con el tiempo se impuso java como lenguaje de aprendizaje. Si bien, java está ampliamente extendido, no es un lenguaje sencillo ni fácil de aprender, la curva de aprendizaje al principio es muy elevada, lo que hace difícil la aproximación a la programación a través de este entorno. No quiero decir ni mucho menos que no sea posible, sino que no es el lenguaje para aprender para neófitos.

Por el contrario, Python incorpora muchas características que hacen que su curva de aprendizaje sea más plana y por tanto más fácil de dominar por parte de los nuevos programadores. De hecho, una gran mayoría de centros de estudios y universidades están utilizando ya Python como el lenguaje de aprendizaje para la programación.

Después de más de veinte años programando, y pasando por muchos lenguajes diferentes: C, C++, Lisp, Cobol, Pascal, Delphi, Basic, C#, Java, PHP, Python y Perl entre otros, me parece que Python reúne todas las características para que un alumno aprenda programación de manera sencilla, e impone restricciones que evitarán vicios difícilmente subsanables posteriormente, añadiendo una gran cantidad de utilidades y herramientas.

Un último dato para afianzar más que es el lenguaje actual que mejor se adapta a la enseñanza, en el año 2020 ha pasado a ser el segundo lenguaje más utilizado en proyecto, por delante de Java, continuando con su ascenso meteórico desde su creación.

Historia de Python

Guido Van Rossum es el creador y responsable de que Python exista. Se trata de un informático de origen holandés que fue el encargado de diseñar Python y de pensar y definir todas las vías posibles de evolución de este popular lenguaje de programación.

En las navidades de 1989 Van Rossum decidió empezar un nuevo proyecto como pasatiempo personal. Pensó en darle continuidad a ABC, un lenguaje de programación que se desarrolló en el mismo centro en el que estaba trabajando. ABC fue desarrollado a principios de los ochenta como alternativa a BASIC. Se trata de un lenguaje pensado para principiantes por su facilidad de aprendizaje y uso. Su código era compacto pero legible. Sin embargo, el proyecto no llegó mucho más lejos por las limitaciones del hardware de la época, así que Van Rossum decidió darle una segunda vida a su idea y partiendo de la base que tenía, empezó a trabajar en Python.

¿Por qué se llama Python? El nombre de este lenguaje de programación es en honor a los Monty Python, el famoso grupo de cómicos británicos.

Versión

La versión 1.0, que se publicó en enero de 1994, la versión 2.0 se publicó en octubre de 2000 y la versión 3.0 se publicó en diciembre de 2008. Esta primera versión de Python ya incluía clases con herencias, manejo de excepciones, funciones y una de sus características fundamentales: funcionamiento modular. Esto permitía que fuese un lenguaje mucho más limpio y accesible para la gente con pocos conocimientos de programación, una característica que se mantiene hasta el día de hoy. Hasta el año 2018, el desarrollo de este popular lenguaje de programación estaba dirigido personalmente por Van Rossum, pero decidió apartarse y, desde 2019, son cinco las personas que deciden cómo evoluciona y se desarrolla Python. Un consejo que se renueva de forma anual.

Versión 1.0

Python es un lenguaje de programación que Van Rossum empezó a desarrollar mientras trabajaba en CWI. Fue este centro de investigación quien liberó, en 1995 la versión 1.2 de Python. A partir de este momento,

ya desvinculado de CWI, Van Rossum hizo aún más accesible el código y para el año 2000, el equipo principal de desarrolladores de Python se cambió a BeOpen.com para formar el equipo de BeOpen Python Labs.

Python 1.6.1 es exactamente lo mismo que 1.6, con algunos bugs arreglados y una nueva licencia compatible con GPL. La versión 1.6 de Python tuvo algunos problemas con su tipo de licencia hasta que la Free Software Foundation (FSF) consiguió cambiar Python a una licencia de Software Libre, que lo haría compatible con GPL.

Versión 2.0

En octubre del año 2000 se publica la segunda versión de Python. Una nueva versión en la que se incluyó la generación de listas, una de las características más importantes del lenguaje.

En 2001, se crea la Python Software Foundation, la cual a partir de Python 2.1 es dueña de todo el código, documentación y especificaciones del lenguaje. A mayores de esta nueva característica, esta nueva versión de Python también incluyó un nuevo sistema gracias al cual los programadores eran capaces de hacer referencias cíclicas y, de esta manera, Python podía recolectar basura dentro del código.

Versión 3.0

La última gran actualización de la historia de Python se produjo en el año 2008 con el lanzamiento de la versión 3.0, que venía a solucionar los principales fallos en el diseño de este lenguaje de programación.

Aunque Python mantiene su filosofía en esta última versión, como lenguaje de programación ha ido acumulando formas nuevas y redundantes de programar un mismo elemento. De ahí la necesidad de nuevas versiones que eliminen estos constructores duplicados.

Python 3.0 rompe la compatibilidad hacia atrás del lenguaje, ya que el código de Python 2.x no necesariamente debe correr en Python 3.0 sin modificación alguna. La última actualización de la versión 3 de Python a la hora de escribir estos apuntes se trata de la **versión 3.9**.

Características de Python

- ∞ Es un lenguaje fuertemente tipado, en concreto todos los elementos son objetos: los tipos, las funciones, etc.
- ∞ Es un lenguaje con tipado dinámico, lo que significa que el tipo de una variable se puede cambiar durante la ejecución. Esto implica que la definición de una variable no tendrá nunca un tipo, pero será necesaria su inicialización antes de usarla.
- ∞ El lenguaje incorpora un gestor de paquetes propio: **pip** que se utiliza para añadir funcionalidades al sistema. Se accederá desde una sesión de consola.

pip search cadena ¹	→ buscar paquetes
pip install paquete	→ instalar un paquete
pip install –U paquete	→ actualizar un paquete
pip uninstall paquete	→ desinstalar un paquete

```
C:\Users\arcipreste>pip search numpy
ERROR: XMLRPC request failed [code: -32500]
RuntimeError: PyPI no longer supports 'pip search' (or XML-RPC search). Please use https://pypi.org/search (via a browser) instead. See https://warehouse.pypa.io/api-reference/xml-rpc.html#deprecated-methods for more information.
```

Hay que tener en cuenta que se debe ejecutar este comando dentro del entorno de ejecución si no queremos que se realice en todo el sistema. Dentro de un IDE este paso es sencillo.

¹ Esta función está desactivada por la sobrecarga que conlleva en los servidores, <https://warehouse.pypa.io/api-reference/xml-rpc.html#deprecated-methods>, en su defecto usar el interfaz web directamente y traspasar a pip install: <https://pypi.org/search/>

- ∞ Es un lenguaje interpretado, pero que se compila a una representación intermedia: byte code. Es un concepto similar a Java.
- ∞ El lenguaje está dirigido a todos los sistemas. Web, desktop, Programación de sistemas, Big data, programación científica.
- ∞ Es un lenguaje que se ejecuta en diversos sistemas operativos: Unix, Windows, etc.
- ∞ Se caracteriza por tener una gran librería. Es imprescindible estudiar la documentación.
- ∞ El lenguaje incorpora una gestión automática de recursos y de memoria, con un recolector de la misma.
- ∞ Es un lenguaje libre con varias implementaciones, la más conocida es la basada en C.
- ∞ El lenguaje impone pocas restricciones al programador.
- ∞ El lenguaje incorpora las facilidades de docstring para documentación del código generado.
- ∞ Por último, se asegura la compatibilidad a nivel de código dentro de la misma rama. Es decir, todo código de la versión 3.1 se ejecutará en cualquier 3.X, pero no bajo 4.X.
- ∞ La curva de aprendizaje para los no conocedores de la programación es muy suave, por el contrario, para aquellos con conocimientos arraigados puede costar cambiar la metodología.

```
Terminal: Local +  
Microsoft Windows [Versión 10.0.17134.471]  
(c) 2018 Microsoft Corporation. Todos los derechos reservados.  
F:\web\wamp\wamp64\www\python\001>pip
```

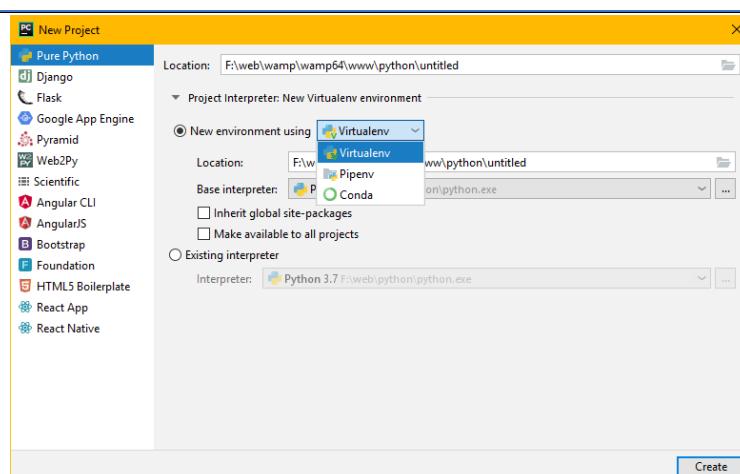
Un vistazo rápido al interior de Python

El lenguaje está compuesto por los siguientes elementos: Gramática y sintaxis, implementaciones, librería estándar, librerías de terceros y Frameworks. Todo programa cuando se ejecuta pasa por las siguientes fases:

- ∞ Carga de la máquina virtual (Python.exe).
- ∞ Compilación en Byte Code. Se compila cada módulo y se guarda en caché dicha compilación (.pyc) no recompilándose nada a menos que sea necesario (La caché se guarda dentro del directorio: **__pycache__**).
- ∞ Ejecución del programa.
 - Se puede crear un entorno virtual de ejecución para cada proyecto, de tal manera que no interfieran unos en otros.

Es recomendable que cada proyecto tenga su propio entorno de ejecución. A la hora de crear el proyecto se debe inicializar este entorno y después instalar las librerías necesarias solo en el proyecto, no en el sistema.

En Python existen utilidades para crear estos entornos virtuales: `venv`, `pyvenv` o `virtualenv`. En caso de usar un IDE será en el momento de la creación del proyecto cuando se determine si se usa el entorno general o se crea uno virtual. Veremos más adelante el uso y creación de estos entornos virtuales.



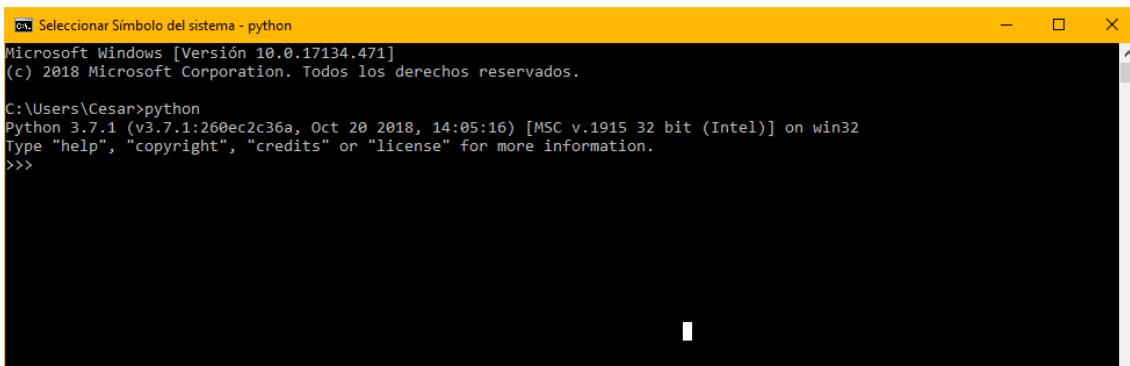
Instalación de Python

La instalación está dirigida sobre todo a Windows sabiendo que la mayoría del alumnado usa este tipo de sistemas, pero para otros sistemas operativos la instalación es igual o más sencilla.

Desde la página <https://www.python.org/downloads/> nos descargamos el ejecutable correspondiente (generalmente x64) y lo instalamos. Una vez finalizado sería recomendable establecer las variables de entorno (PATH del sistema) de forma adecuada si no se hace en la instalación:

Variables de entorno	
Variables de usuario para Cesar	
Variable	Valor
MOZ_PLUGIN_PATH	
OneDrive	
Path	F:\web\python\Scripts\;F:\web\python\

Con estas configuraciones, podremos acceder a una consola Python desde el símbolo de sistema o un terminal ejecutando simplemente **Python**.



The screenshot shows a Windows Command Prompt window titled "Seleccionar Símbolo del sistema - python". The window displays the following text:
Microsoft Windows [Versión 10.0.17134.471]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Cesar>python
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:05:16) [MSC v.1915 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

En caso que no deseemos instalar todo desde cero, se puede utilizar la distribución Anaconda de Python, si bien está orientada a la programación científica se puede usar como plataforma de aprendizaje: <https://www.anaconda.com/>

En un entorno Unix, la propia distribución incluirá los paquetes necesarios para la instalación. Así en sistemas Debian/Ubuntu será tan sencillo como **apt-get install python**

Entornos IDE

Para el desarrollo de un proyecto Python es altamente recomendable utilizar un IDE que nos proporcione todas las características, si bien no es imprescindible y es posible usar: atom, sublime, etc. como entornos de desarrollo, lo más indicado es utilizar uno. Los IDEs más extendidos a día de hoy son: PyCharm y Microsoft Visual Studio. Este curso usaremos PyCharm.

Si nos encontramos que PyCharm no muestra ni llaves, corchetes, etc. hay que añadir "actionSystem.force.alt.gr=true" al final del fichero **bin\idea.properties** en el directorio de instalación.

La instalación de PyCharm es bastante sencilla no siendo necesaria ninguna configuración si tenemos instalado antes Python en el sistema.

Elementos de un programa

Estructura General de un programa

Un programa de Python es un fichero de texto (normalmente guardado con el juego de caracteres UTF-8) que contiene expresiones y sentencias del lenguaje Python. Esas expresiones y sentencias se consiguen combinando los elementos básicos del lenguaje.

Conceptos básicos a recordar

- ∞ Los ficheros bajo Python se codifican en formato UTF-8 de forma general, pero se puede cambiar añadiendo en la primera línea del fichero fuente la codificación deseada: **# -*- coding: cp-1252**
- ∞ Cada línea representará una instrucción, no hace falta un delimitador final (como en otros lenguajes) ya que la indentación es parte imprescindible de la sintaxis. Así para crear un bloque sintáctico simplemente se indentarán todas las líneas del bloque el mismo número de espacios, marcando la línea principal de bloque con (:) dos puntos. A la hora de indentar no se pueden mezclar tabuladores y espacios, se recomienda usar cuatro espacios para indentar.
- ∞ Python usa comentarios de fin de línea exclusivamente, no existe ningún otro tipo de comentario (#). Es posible simular comentarios multilíneas con la notación extendida de representación de cadenas en varias líneas: """ """"
- ∞ El lenguaje diferencia entre mayúsculas y minúsculas tanto en las palabras clave como en los nombres de variables y funciones.

```
print("Hola Mundo") # mi primer comentario

# Los comentarios son sólo de fin de línea

for cnt in [1, 2, 3, 4]:
    print(cnt)

"""

Aunque de esta manera se puede simular
un comentario de varias líneas sin serlo.

"""
```

Funciones introductorias

Para poder desarrollar los ejemplos y avanzar de forma adecuada en el conocimiento del lenguaje debemos introducir varias funciones que nos servirán de ayuda.

- ∞ **print(,,end="")**. Esta función imprimirá un conjunto de parámetros en una línea por la salida estándar del programa, generalmente la pantalla. El parámetro **end** indica el último carácter a añadir a la línea en la salida, por defecto \n.
- ∞ **input("texto")**. Esta función se utiliza para recoger una cadena de texto del usuario. En caso que el valor a tratar sea numérico es imprescindible realizar la conversión. Utilizará por defecto la entrada estándar del programa.
- ∞ **str(valor)**, **int(valor)** e **float(valor)** convierten respectivamente a cadena un objeto cualquiera, a entero cualquier valor no numérico y a decimal. En el último caso no convertirá si el valor contiene alguna letra.
- ∞ **dir(objeto)** devuelve la lista de los métodos y propiedades existentes en el objeto.
- ∞ **type(objeto)** devuelve el tipo del objeto.
- ∞ **pass**. No ejecución. Se utiliza para crear bloques vacíos ya que no es posible que se deje un bloque sin instrucciones.

Ejemplo: Ejecutar un programa

Vamos a crear nuestro primer programa para familiarizarnos con todos los procedimientos.

Ejemplo: Crear el siguiente programa.

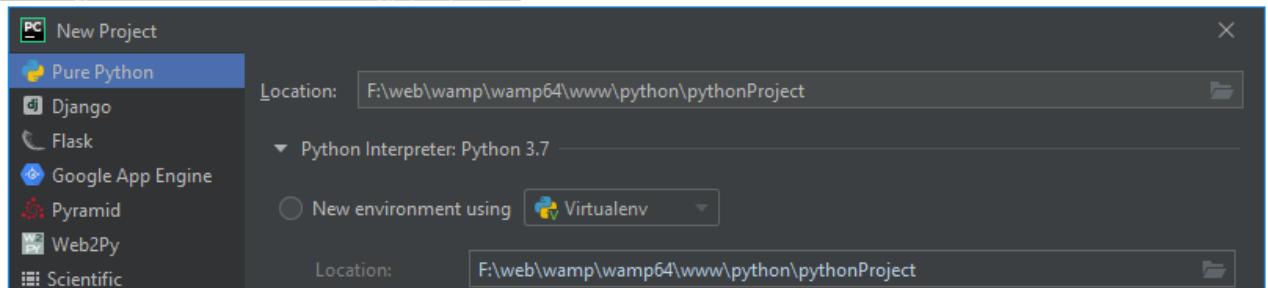
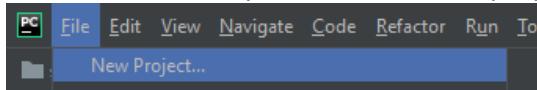
```
print("Prueba de entrada")

num1 = int(input("Primer Valor:"))
num2 = int(input("Segundo Valor:"))

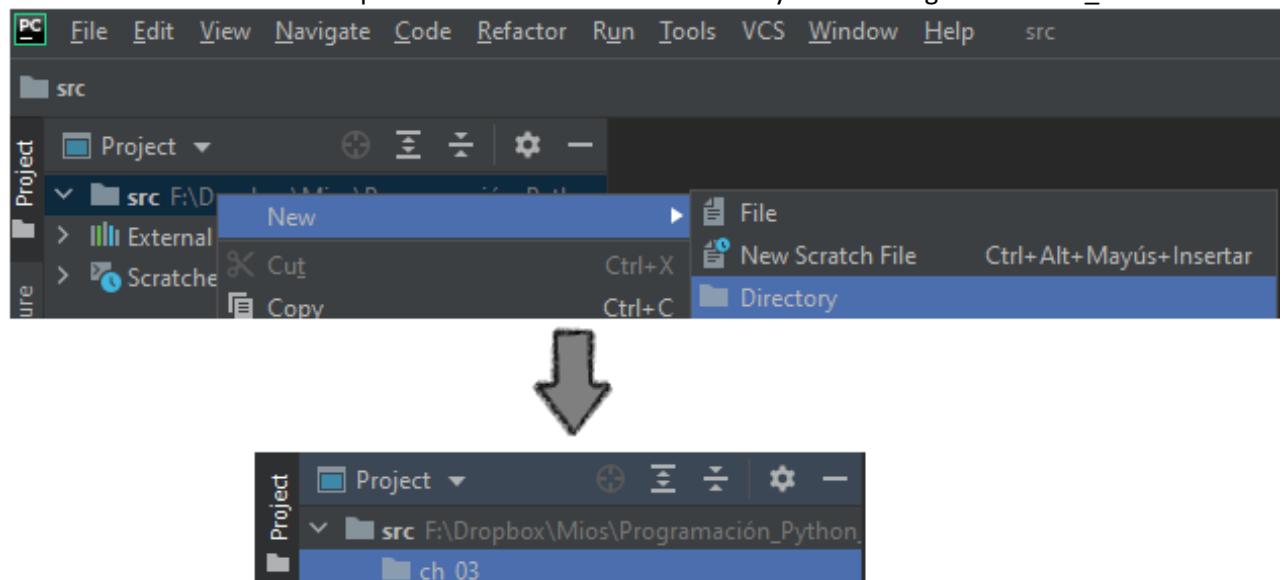
print("La suma es:", str(num1 + num2))
```

code_0000.py

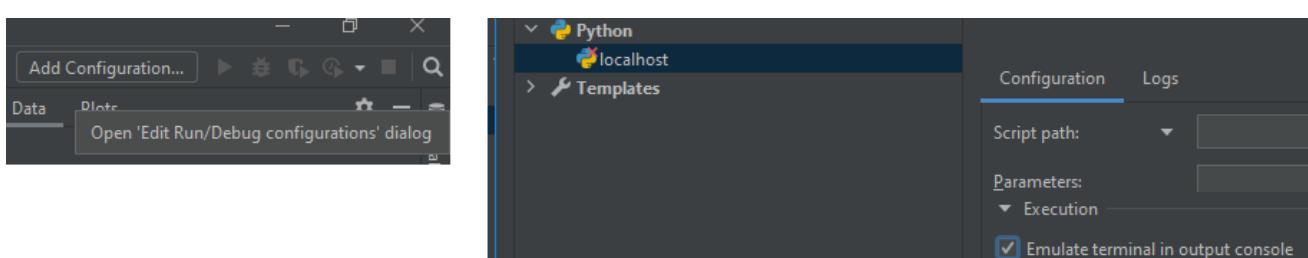
1. Abrimos el entorno y creamos un nuevo proyecto, estableciendo el directorio de trabajo.



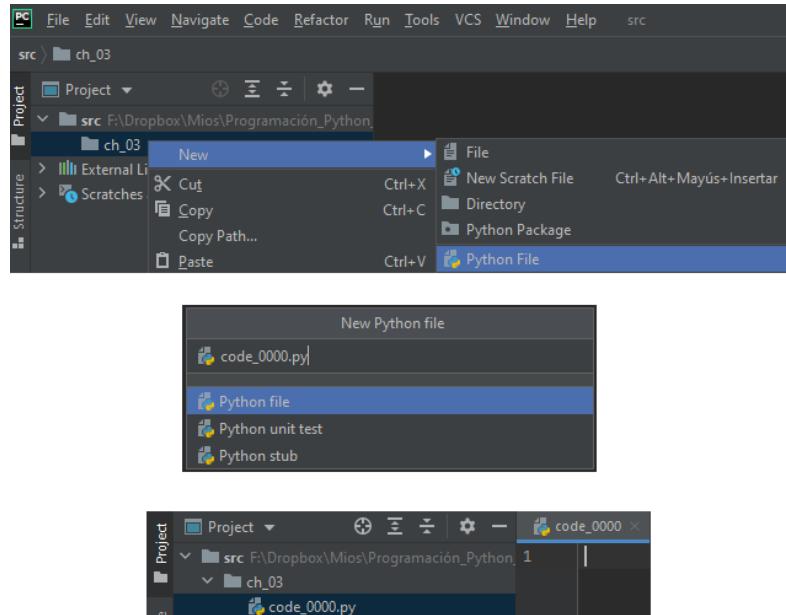
2. Creamos un nuevo directorio para contener los ficheros fuentes y tenerlos organizados: ch_03.



3. Añadimos una configuración de ejecución al entorno, para poder ejecutar ficheros. Se pulsará en la parte superior el botón más (+) y de la lista desplegable seleccionaremos Python.

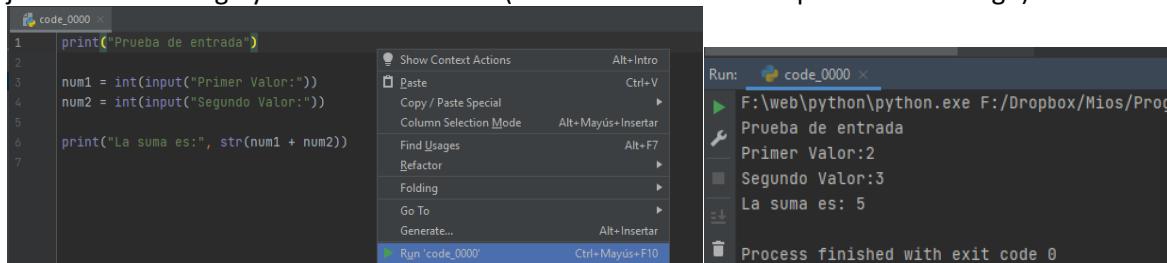


4. Creamos el fichero.



5. Introducimos el código fuente tal y como aparece.

6. Ejecutamos el código y vemos el resultado (botón derecho sobre la pestaña del código).



Hemos procedido a crear un fichero y todas las estructuras necesarias para la ejecución del código, pero no es necesario utilizar un IDE tal y como dijimos anteriormente. Se puede hacer la programación en cualquier editor de texto y ejecutarlo llamando a la orden **python nombre_archivo**, veámoslo.

1. Se abre el editor de texto preferido (write.exe, Notepad.exe, vi, joe, etc.) y se escribe el fichero fuente, almacenándolo en formato UTF-8. Esto último es obligatorio si no queremos que de un error de sintaxis.
2. Se abre una consola del sistema (cmd.exe en Windows, bash en Linux).
3. Se localiza el fichero fuente con la ruta completa.
4. Se ejecuta pasándolo como parámetro a la orden **python**.

```
Microsoft Windows [Versión 10.0.18363.1379]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Cesar>python F:\Dropbox\Mios\Programación_Python_DAW\libro\src\ch_03\code_0000.py
Prueba de entrada
Primer Valor:2
Segundo Valor:3
La suma es: 5

C:\Users\Cesar>
```

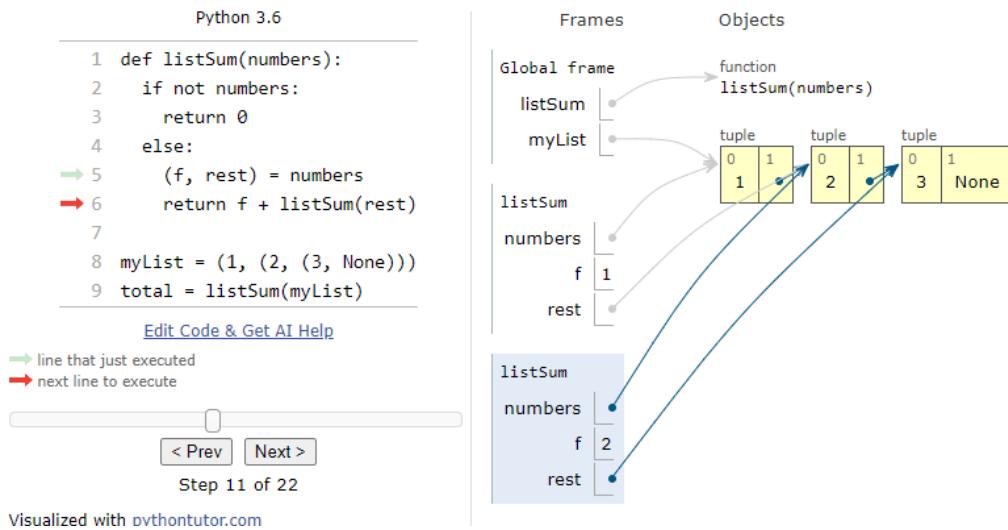
Explicación del código

- ∞ **print("Prueba de entrada")**. Muestra por pantalla un texto.
- ∞ **num1 = int(input("Primer Valor:"))**. Crea una variable llamada num1 y le asigna el valor entero que el usuario introduzca por teclado, si no puede convertirlo dará error.

- ∞ `num2 = int(input("Segundo Valor:"))`. Similar pero llamada num2.
- ∞ `print("La suma es:", str(num1 + num2))`. Imprime un texto y el resultado de la operación de sumar las dos variables.

Ejemplo: Crear un programa con el típico: Hola mundo.

Herramienta de depuración: <https://pythontutor.com/>



Líneas y espacios en blanco

- ∞ Un programa Python se divide en un número de **líneas lógicas**. El final de una línea lógica está representado por el token NEWLINE (nueva línea, no hace falta ningún otro terminador como el punto y coma). Una línea lógica se construye a partir de una o más líneas físicas siguiendo las reglas de unión de líneas.
- ∞ Una **línea física** es una secuencia de caracteres terminada por una secuencia de final de línea. En los archivos fuente y las cadenas, se puede utilizar cualquiera de las secuencias de terminación de línea de la plataforma estándar: Unix utiliza ASCII LF, Windows utiliza la secuencia ASCII CR LF. Todas estas formas pueden ser utilizadas por igual, independientemente de la plataforma.
- ∞ **Dos o más líneas físicas pueden unirse en líneas lógicas** utilizando caracteres de barra invertida (\), de la siguiente manera: cuando una línea física termina en una barra invertida que no es parte de literal de cadena o de un comentario, se une con la siguiente formando una sola línea lógica

```

if 1900 < year < 2100 and 1 <= month <= 12 \
    and 1 <= day <= 31 and 0 <= hour < 24 \
    and 0 <= minute < 60 and 0 <= second < 60:
    return 1

```

Una línea que termina en una barra invertida no puede llevar un comentario. No se puede usar una barra invertida para continuar un comentario.

- ∞ Las expresiones entre paréntesis, entre corchetes o entre llaves pueden dividirse en más de una línea física sin usar barras invertidas. Por ejemplo:

```

month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',       'Juni',      # Dutch names
               'Juli',    'Augustus',  'September', # for the months
               'Oktober', 'November', 'December'] # of the year

```

Las líneas continuas implícitas pueden llevar comentarios. No es importante la sangría de las líneas de continuación.

- ∞ Una línea lógica que contiene sólo espacios, tabulaciones, saltos de página y posiblemente un comentario, es ignorada.
- ∞ A excepción del comienzo de una línea lógica o en los literales de cadenas, los caracteres de espacio en blanco, tabulación y formfeed pueden utilizarse indistintamente para separar tokens. Los espacios en blanco se necesitan entre dos tokens sólo si su concatenación podría interpretarse de otra manera como un token diferente.
- ∞ **Un bloque en Python viene determinado por el nivel de sangría aplicada al código**, así como del delimitador de comienzo de bloque (:). De tal manera que todas aquellas líneas lógicas que tengan la misma sangría estarán dentro del mismo bloque. El sangrado es obligatorio y en caso de utilizarlo mal generará un error de sintaxis. Las recomendaciones PEP indican que el sangrado de cada bloque se debe hacer con cuatro espacios en blanco más que el bloque anterior, pero nada nos impide usar tabuladores o cualquier otro número de espacios en blanco siempre que no se mezclen ni se cambie el número de espacios.

```
def perm(l):
    # Compute the list of all permutations of l
    if len(l) <= 1:
        return [l]
    r = []
    for i in range(len(l)):
        s = l[:i] + l[i+1:]
        p = perm(s)
        for x in p:
            r.append(l[i:i+1] + x)
    return r
```

Comentarios

Un comentario comienza con un carácter de almohadilla (#) que no es parte de un literal de cadena y termina al final de la línea física. Los comentarios son ignorados por la sintaxis y se utiliza para documentar o explicar el código fuente.

Identificadores

Un identificador es una secuencia de uno o más caracteres asignada por el programador a un elemento del programa (constante, variable, método, clase, paquete...).

Un identificador en Python comienza con una letra (de la A a la Z o de la a a la z) o con un guion bajo (_) seguido de cero o más letras, guiones bajos y números. Python distingue mayúsculas de minúsculas en el nombrado de los identificadores y Python **NO** permite signos de puntuación como @, \$ y %, excepto el guion bajo (_).

Se recomienda nombrar las clases comenzando por una letra mayúscula cada palabra sin espacios y el resto de los identificadores por una letra minúscula separando las palabras con guiones bajos, excepto aquellos que funcionen como constantes que se escribirán en mayúsculas y los identificadores que se definan en las clases como privados empezarán por el guion bajo. Ejemplos:

```
numero_palabras, encontrar_media, imprimir_linea
NUMERO_ALUMOS, LINEAS POR PAGINA
_nombre_persona,
```

Palabras reservadas

Los siguientes identificadores se utilizan como palabras reservadas, o palabras clave del lenguaje, y no pueden utilizarse como identificadores ordinarios. Deben escribirse exactamente como están escritas aquí:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try

as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Operadores y expresiones

Un operador es un token que realiza alguna operación sobre uno o más operandos. Los operadores se clasifican según su tipo.

Operador de concatenación de cadenas de caracteres

Una de las operaciones más básicas cuando se trabaja con cadenas de caracteres es la concatenación. Esto consiste en unir dos cadenas en una sola, siendo el resultado una nueva cadena.

La forma más simple de concatenar dos cadenas en Python es utilizando el operador de concatenación +:

```
>>> hola = 'Hola'
>>> python = 'Pythonista'
>>> hola_python = hola + ' ' + python # concatenamos 3 strings
>>> print(hola_python)
Hola Pythonista
```

Operadores lógicos o booleanos

A la hora de operar con valores booleanos, tenemos a nuestra disposición los operadores and, or y not.

Las operaciones and, or y not realmente no devuelven True o False, sino que devuelven uno de los operandos como veremos en el cuadro de abajo.

Operación	Resultado	Descripción
a or b	Si a se evalúa a falso, entonces devuelve b, si no devuelve a	Solo se evalúa el segundo operando si el primero es falso
a and b	Si a se evalúa a falso, entonces devuelve a, si no devuelve b	Solo se evalúa el segundo operando si el primero es verdadero
not a	Si a se evalúa a falso, entonces devuelve True, si no devuelve False	Tiene menos prioridad que otros operadores no booleanos

Ejemplos:

```
>>> x = True
>>> y = False
>>> x or y
True
>>> x and y
False
>>> not x
False
>>> x = 0
>>> y = 10
>>> x or y
10
>>> x and y
0
>>> not x
True
```

Operadores de comparación

Los operadores de comparación se utilizan, como su nombre indica, para comparar dos o más valores. El resultado de estos operadores siempre es True o False.

Operador	Descripción
----------	-------------

>	Mayor que. True si el operando de la izquierda es estrictamente mayor que el de la derecha; False en caso contrario.
>=	Mayor o igual que. True si el operando de la izquierda es mayor o igual que el de la derecha; False en caso contrario.
<	Menor que. True si el operando de la izquierda es estrictamente menor que el de la derecha; False en caso contrario.
<=	Menor o igual que. True si el operando de la izquierda es menor o igual que el de la derecha; False en caso contrario.
==	Igual. True si el operando de la izquierda es igual que el de la derecha; False en caso contrario.
!=	Distinto. True si los operandos son distintos; False en caso contrario.

Ejemplos:

```
>>> x = 9
>>> y = 1
>>> x < y
False
>>> x > y
True
>>> x == y
False
```

Consideraciones sobre los operadores de comparación

Los objetos de diferentes tipos, excepto los tipos numéricos, nunca se comparan igual. El operador == siempre está definido, pero para algunos tipos de objetos (por ejemplo, objetos de clase) es equivalente a **is**.

Las instancias no idénticas de una clase normalmente se comparan como no iguales a menos que la clase defina el método **__eq__()**. Las instancias de una clase no se pueden ordenar con respecto a otras instancias de la misma clase u otros tipos de objeto, a menos que la clase defina los métodos **__lt__()**, **__gt__()**.

Los operadores de comparación se pueden concatenar.

Ejemplo:

```
>>> x = 9
>>> 1 < x and x < 20
True
>>> 1 < x < 20
True
```

Operadores aritméticos en Python

En cuanto a los operadores aritméticos, estos permiten realizar las diferentes operaciones aritméticas del álgebra: suma, resta, producto, división, ... Estos operadores Python son de los más utilizados. El listado completo es el siguiente:

Operador	Descripción
+	Suma dos operandos.
-	Resta al operando de la izquierda el valor del operando de la derecha. Utilizado sobre un único operando, le cambia el signo.
*	Producto/Multiplicación de dos operandos.
/	Divide el operando de la izquierda por el de la derecha (el resultado siempre es un float).
%	Operador módulo. Obtiene el resto de dividir el operando de la izquierda por el de la derecha.
//	Obtiene el cociente entero de dividir el operando de la izquierda por el de la derecha.
**	Potencia. El resultado es el operando de la izquierda elevado a la potencia del operando de la derecha.

Ejemplos:

```
>>> x = 7
>>> y = 2
>>> x + y # Suma
9
>>> x - y # Resta
5
>>> x * y # Producto
14
>>> x / y # División
3.5
>>> x % y # Resto
1
>>> x // y # Cociente
3
>>> x ** y # Potencia
49
```

Operadores a nivel de bits

Los operadores a nivel de bits actúan sobre los operandos como si fueran una cadena de dígitos binarios. Como su nombre indica, actúan sobre los operandos bit a bit. Son los siguientes:

Operación	Descripción
x y	or bit a bit de x e y.
x ^ y	or exclusivo bit a bit de x e y.
x & y	and bit a bit de x e y.
x << n	Desplaza x n bits a la izquierda.
x >> n	Desplaza x n bits a la derecha.
~x	not x. Obtiene los bits de x invertidos.

Ejemplo: Supongamos que tenemos el entero 2 (en bits es 00010) y el entero 7 (00111). El resultado de aplicar las operaciones anteriores es:

```
>>> x = 2
>>> y = 7
>>> x | y
7
>>> x ^ y
5
>>> x & y
2
>>> x << 1
4
>>> x >> 1
1
>>> ~x
-3
```

Operadores de asignación

El operador de asignación se utiliza para asignar un valor a una variable. Como se ha mencionado en otras secciones, este operador es el signo =.

Además del operador de asignación, existen otros operadores de asignación compuestos que realizan una operación básica sobre la variable a la que se le asigna el valor. Por ejemplo, $x += 1$ es lo mismo que $x = x + 1$. Los operadores compuestos realizan la operación que hay antes del signo igual, tomando como operandos la propia variable y el valor a la derecha del signo igual.

Operador	Ejemplo	Equivalencia
$+=$	$x += 2$	$x = x + 2$
$-=$	$x -= 2$	$x = x - 2$
$*=$	$x *= 2$	$x = x * 2$
$/=$	$x /= 2$	$x = x / 2$
$%=$	$x %= 2$	$x = x \% 2$
$//=$	$x //= 2$	$x = x // 2$
$**=$	$x **= 2$	$x = x ** 2$
$&=$	$x &= 2$	$x = x \& 2$
$ =$	$x = 2$	$x = x 2$
$^=$	$x ^= 2$	$x = x ^ 2$
$<<=$	$x <<= 2$	$x = x << 2$
$>>=$	$x >>= 2$	$x = x >> 2$

Operador morsa

Python ha creado un operador de morsa (:=) en la versión 3.8 que permite asignar valores a una variable como parte de una expresión. Se puede realizar gran cantidad de cosas interesantes, pero el mejor aspecto es la reducción de líneas necesarias para tareas muy comunes.

Ahora, en vez de escribir esto:

```
line = f.readline()
while line:
    line = f.readline()
```

Puedes escribir esto:

```
while line := f.readline():
```

Operadores de pertenencia

Los operadores de pertenencia se utilizan para comprobar si un valor o variable se encuentran en una secuencia (**list, tuple, dict, set o str**).

Operador	Descripción
in	Devuelve True si el valor se encuentra en una secuencia; False en caso contrario.
not in	Devuelve True si el valor no se encuentra en una secuencia; False en caso contrario.

Ejemplos:

```
>>> lista = [1, 3, 2, 7, 9, 8, 6]
>>> 4 in lista
False
>>> 3 in lista
True
>>> 4 not in lista
True
```

Operadores de identidad

Por último, los operadores de identidad se utilizan para comprobar si dos variables son, o no, el mismo objeto.

Operador	Descripción
is	Devuelve True si ambos operandos hacen referencia al mismo objeto; False en caso contrario.
is not	Devuelve True si ambos operandos no hacen referencia al mismo objeto; False en caso contrario.

Ejemplos:

```
>>> x = 4
>>> y = 2
>>> lista = [1, 5]
>>> x is lista
False
>>> x is y
False
>>> x is 4
True
```

Prioridad de los operadores en Python

Al igual que ocurre en las matemáticas, los operadores en Python tienen un orden de prioridad. Este orden es el siguiente, de menos prioritario a más prioritario: asignación; operadores booleanos; operadores de comparación, identidad y pertenencia; a nivel de bits y finalmente los aritméticos (con el mismo orden de prioridad que en las matemáticas). Este orden de prioridad se puede alterar con el uso de los paréntesis ()

Expresiones

Una expresión es una combinación de valores, variables y operadores. La evaluación de una expresión produce un valor, esta es la razón por la que las expresiones pueden aparecer en el lado derecho de las sentencias de asignación. Un valor, por sí mismo, se considera como una expresión, lo mismo ocurre para las variables.

Expresa en Python las siguientes expresiones

- Mayor de edad y cinco años o más cotizados
- Una manzana es seleccionada si el color es amarillo o rojo y además el diámetro está entre 15 y 25 cm.
- La presión de un balón debe estar entre 3 y 5 bares y la circunferencia no debe pasar de 45 cm
- El número de bolas rojas serán tres si el de amarillas es dos, debe ser cinco si el de amarillas es cinco.
- Para seleccionar un jugador debe medir entre 180 y 210 cm. Si es más alto de 210cm tiene que pesar menos de 120kg. Si es menor de 180cm tiene que saltar más de 50cm.
- Para comprar un vehículo debe ser azul si es un deportivo o verde si es un 4x4, pero si quiero una moto la compraré roja o negra.
- La jubilación se produce a los 67 años con 35 cotizados. a partir de 65 con 40 cotizados y 20 o más continuos

Representación de los datos (Literales)

Literales

Los literales son las formas de escribir (notaciones) que utilizamos para los valores constantes de algunos tipos, son la representación escrita de los datos.

Literales numéricos

Hay tres tipos de literales numéricos: números enteros, números de punto flotante y números imaginarios. No hay literales complejos (los números complejos pueden formarse sumando un número real y un número imaginario: $4 + 3j$).

- ∞ **Enteros.** (enteros, octales 0o, binarios 0b y hexadecimales 0x)


```
7 2147483647 0o177 0b100110111 3 79228162514264337593543950336 0o377
Oxdeadbeef 1000000000000 100_000_000_000 0b_1110_0101
```
- ∞ **Coma flotante.** (notación tradicional o notación científica)


```
3.14 10. .001 1e100 3.14e-10 0e0 3.14_15_93
```
- ∞ **Imaginarios.**

```
3.14j 10.j 10j .001j 1e100j 3.14e-10j 3.14_15_93j
```

Literales de bytes

Los literales de bytes siempre se prefijan con 'b' o 'B'; y producen una instancia del tipo **bytes** en lugar del tipo **str**, por lo que solo pueden contener caracteres ASCII. Los bytes con un valor numérico de 128 o mayor deben ser expresados con secuencias numéricas (\xxxx). Además, pueden ser prefijados con una letra 'r' o 'R', tales cadenas se llaman raw strings y consideran las barras inversas como caracteres literales.

Ejemplos:

```
b'Espa\xc3\xb1a'.decode("UTF-8")      # España
b'Espa\xc3\xb1a'.decode("Latin1")    # España
rb'spam\xddeg'                      # spam\xddeg
```

Literales Cadena

Las cadenas (tipo **str**) en Python pueden ser encerrados entre comillas simples ('') o dobles ("") pero no se pueden mezclar los delimitadores. También pueden estar encerrados en grupos de tres comillas simples o dobles (a las que generalmente se les llama cadenas de tres comillas) para dividir la cadena en varias líneas físicas. El carácter de la barra inversa (\) se utiliza para escapar los caracteres que de otra manera tienen un significado especial, como la línea nueva, la barra inversa en sí misma, o el carácter de comillas. Son valores inmutables. Se puede acceder a un carácter mediante [] indexados en cero.

```
print('Hola Mundo')
mi_cadena = """Clase que representa
una Persona"""
print(mi_cadena)
print("hola \n \" mundo \" ")
print('hola \n \" mundo \" ')

cadena = "Hola"
cadena[2] = "a" # error son inmutables
```

Los literales de cadena pueden ser prefijados con una letra 'r' o 'R'; tales cadenas se llaman raw strings y consideran las barras inversas como caracteres literales. A menos que un prefijo 'r' o 'R' esté presente, las secuencias de escape en literales de cadena y bytes se interpretan según reglas similares a las usadas por C estándar. Las secuencias de escape reconocidas son:

Secuencia de escape	Significado
\newline	Barra inversa y línea nueva ignoradas
\\\	Barra inversa (\)
\'	Comilla simple (')
\"	Comilla doble ("")
\a	ASCII Bell (BEL)
\b	ASCII Retroceso (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)

\r	ASCII Retorno de carro (CR)
\t	ASCII Sangría horizontal (TAB)
\v	ASCII Sangría vertical (VT)
\ooo	Carácter con valor octal <i>oo</i>
\xhh	Carácter con valor hexadecimal <i>hh</i>

Las secuencias de escape que sólo se reconocen en los literales de cadena son:

Secuencia de escape	Significado
\N{name}	El carácter llamado <i>name</i> en la base de datos de Unicode
\xxxxx	Carácter con valor hexadecimal de 16 bits xxxx
\xxxxxxxx	Carácter con valor hexadecimal de 32 bits xxxxxxxx

Se permiten múltiples literales de cadenas adyacentes (delimitados por espacios en blanco), utilizando diferentes convenciones y su significado es el mismo que su concatenación. Por lo tanto, "hola" 'mundo' es equivalente a "holamundo". Esta característica puede ser utilizada para reducir el número de barras inversas necesarias, para dividir largas cadenas convenientemente a través de largas líneas, o incluso para añadir comentarios a partes de las cadenas, por ejemplo:

```
re.compile("[A-Za-z_]"           # letter or underscore
          "[A-Za-z0-9_]*"        # letter, digit or underscore
      )
```

Téngase en cuenta que esta característica se define a nivel sintáctico, pero se implementa en el momento de la compilación, por lo que el operador "+" debe ser usado para concatenar **expresiones** de cadena al momento de la ejecución.

Cadenas formateadas

Una cadena formateada o f-string es un literal de cadena que se prefija con 'f' o 'F'. Estas cadenas pueden contener campos de reemplazo, que son expresiones delimitadas por llaves {}. Mientras que otros literales de cadena siempre tienen un valor constante, las cadenas formateadas son realmente expresiones evaluadas en tiempo de ejecución. Ver más adelante en la sección cadenas el nuevo modelo **format**.

Tipos de datos

En cualquier lenguaje de programación de alto nivel se manejan tipos de datos. Los tipos de datos definen un conjunto de valores que tienen una serie de características y propiedades determinadas. Los tipos de datos básicos de Python son los booleanos, los numéricos (enteros, punto flotante y complejos), el tipo nulo y las cadenas de caracteres. Los tipos de datos complejos los veremos en unidades posteriores.

Tipado fuerte vs Tipado débil vs No tipado

- Tipado fuerte indica que el lenguaje no permite operaciones entre tipos que no sean los especificados, así no se podrá pasar un float a un parámetro int o sumar una cadena y un número.
- Por contra, débil permite este tipo de operaciones, en la que se pasa o usa otro tipo al especificado.
- Por último, no tipado indica que el tipo del parámetro se determina en tiempo de ejecución.

Tipado estático vs Tipado dinámico

- El tipado estático establece en la definición de una variable o constante el tipo y es inamovible.
- El tipado dinámico hace que el tipo del contenido de una variable o constante pueda cambiar durante la ejecución.

Booleanos

Para tratar las expresiones booleanas, el lenguaje incorpora dos objetos que representarán verdadero (True) y falso (False), pero no existe un tipo como tal.

Nulo

El lenguaje no tiene un valor u objeto que represente el concepto de nulo en una Base de datos, o en otros lenguajes como C, pero existe el objeto **None** que representará la no existencia o el vacío dependiendo del contexto.

Numéricos

Dentro de los números que puede manejar Python se encuentra los enteros (**int**: -5), los decimales (**float**: 2.3) y los irracionales: 3j.

Podemos realizar conversiones entre los diferentes tipos con las funciones: **hex()** para convertir a hexadecimal, **oct()** para octal, **bool()** para booleana, **int()** para entero y **float()** para números decimales y **complex()** para complejos.

La precisión matemática que proporciona el tipo **float** no es adecuada para todos los entornos, por lo que si es necesario se puede utilizar una librería con una precisión mayor: **decimal**.

Se puede hacer uso de funciones matemáticas de forma directa: **abs**, **round**, **min**, **max**, **sum** o de la librería **math** (**ceil**, **floor**, **trunc**, etc.) importándola antes con **import math**.

```
c1 = 4 + 3j
print(type(c1))
c2 = complex(2, -3)
print(c1 + c2)
print(c1 - c2)
print(3 * c1)
print(c1 * c2)
print(c1 == c2)
print(c1.conjugate())
print(c1 / c2)
print(abs(c1))
print(c1.real)
print(c1.imag)

# comparadores en secuencia: a mayor que dos y menor que cinco
print(2 < a < 5)
```

Cadenas

Las cadenas en Python (tipo **str**) se inicializan con cualquiera de los siguientes métodos: "Cadena", 'Cadena' o """ Cadena """. En este último caso se podrá partir la cadena en varias líneas y se tendrá en cuenta los retornos de carro, en los otros dos métodos deberemos escapar los retornos de carro.

Hay que tener en cuenta que Python trabaja de base con UTF-8 con lo que la representación interna de toda cadena se realiza en esta codificación, por lo que es prescriptivo convertirlo a otra codificación si lo requiriese la salida.

En caso que necesitemos caracteres Unicode con la secuencia de escape deberemos anteponer a la definición de la cadena u, y si necesitamos acceso a los bytes utilizaremos antes de la cadena r (raw) para cadenas en crudo Unicode o b (byte) para ASCII.

Bajo Python no existe el concepto de carácter como otros lenguajes, se implementa con una cadena de longitud uno. El acceso a un carácter individual se realizará mediante índice y la notación corchete, sabiendo que los índices empiezan en 0 y terminan en **len(cadena)-1**.

Las cadenas en Python son objetos **inmutables**, por lo que no permiten modificaciones directas del contenido y cualquiera que se realice con una de sus funciones será necesario asignarla a una variable para no perder dicho cambio.

Podemos conseguir una subcadena simplemente usando la sintaxis corchete [inicio: fin], teniendo en cuenta que fin nunca se incluirá en el resultado y que si usamos valores negativos significa que empezaremos desde el

final. También consideraremos que la omisión de uno de los parámetros significa: si omitimos el primero indica índice cero y si omitimos el último, indica **len(cadena)**.

```
print("la" in "!Hola")      # True
pal="Una palabra"
print(pal, pal[0],pal[-2])  # Una palabra U r
print(pal[0:2])            # Un
print(pal[4:])              # palabra
print(pal[-3:-1])          # br
print("Una" "Palabra")     # UnaPalabra
pal[3]="b"                  # 'str' object does not support item assignment
print(len(pal))             # Una PalabraUna Palabra
print(pal*2)                 # Una PalabraUna Palabra
```

Para finalizar, los operadores que podemos usar con las cadenas con la concatenación (+), innecesaria para solo cadenas ya que el lenguaje concatena dos cadenas consecutivas, y el operador repetición (*) que repite una cadena un número de veces.

Formato de cadenas

Para formatear una cadena podemos usar dos métodos, uno más antiguo y desaconsejado similar a las cadenas de formato de la función **printf** del C, y el nuevo mecanismo basado en la función **format()**.

Así el mecanismo antiguo sería: "%s es una %s" % (a, b). Este formato utiliza las reglas de sintaxis de **printf** de C añadiendo la posibilidad de que entre el % y el símbolo correspondiente de tipo exista un nombre que se use como clave de un diccionario para el valor en vez de la posición. Podemos aprender más sobre este formato en la siguiente dirección:

<https://docs.python.org/es/3/library/stdtypes.html#old-string-formatting>

El nuevo mecanismo implica el uso de llaves ({}) para indicar en la cadena la posición del parámetro, anteponiendo la letra f a la cadena o llamando al método **format**. Se puede utilizar números, claves o dejar vacías las llaves, pero no se deben mezclar. En caso de usar números se puede utilizar el orden que deseemos, no siendo obligatorio que sea secuenciales o incrementales. También permite centrar el valor anteponiendo: ^Tamaño, justificarlo a la derecha :>tamaño o a la izquierda :<tamaño. Para la impresión de números podemos llenar con ceros anteponiendo el carácter que necesitamos.

```
pal="Una palabra"
print("{}".format(pal))
print("{} es {}".format("A", "b"))
print("{0} es {1}".format("A", "b"))
print("{1} es {0}".format("A", "b"))
print("{:<10} {:^10} {:>10} {:>5}".format("Ab", "CDE", "eF", 23))
# Ab           CDE           eF 00023
# Formato recomendado con el prefijo f
width = 10
precision = 4
value = decimal.Decimal("12.34567")
print(f"result: {value}")
print(f"result: {value = }") # escribe number_var = valor
print(f"result: {value:{width}.{precision}}") # nested fields
# 'result: 12.35'
today = datetime(year=2017, month=1, day=27)
print(f"{today:%B %d, %Y}") # using date format specifier
# 'January 27, 2017'
```

Forzado de tipos

Python es un lenguaje con tipado fuerte pero dinámico, con lo que el tipo de cada variable se establece en tiempo de ejecución, pero tiene un tipo. Una de las mayores críticas que sufren los lenguajes con tipado dinámico son los problemas que introducen al poder el variar el tipo de una variable durante el desarrollo de la aplicación y no detectarse en el momento del análisis sintáctico errores en los usos de los tipos. Cada lenguaje a introducido sus propios mecanismos, pero bajo Python se han creado las anotaciones.

Las anotaciones permiten a otras herramientas (IDEs, analizadores estáticos como mypy o pylint) realizar las mismas tareas que en los lenguajes tradicionales se realizan en el análisis sintáctico. Pero hay que tener claro que, si no usamos esas herramientas, las anotaciones no serán más que meras clarificaciones, ya que el lenguaje no impone restricciones en el tipo de las variables durante la ejecución, pudiendo cambiar si así lo deseamos.

La estructura general de una anotación es:

```
nombre_variable: tipo [valor]
```

Veamos ejemplos a usar.

```
age: int = 1
x_1: float = 1.0
x_2: bool = True
x_3: str = "test"
x_4: bytes = b"test"
```

Los tipos anteriores los usaremos inmediatamente en nuestros programas, los siguientes se utilizarán a lo largo de diferentes capítulos, pero se introducen aquí para tener una referencia completa en un único lugar.

```
x: list[int] = [1]      # Listas
x: set[int] = {6, 7}    # Conjuntos
x: dict[str, float] = {"field": 2.0}  # Diccionarios
x: tuple[int, str, float] = (3, "yes", 7.5) # Tuplas

x: list[int | str] = [3, 5, "test", "fun"] #Valores optativos
```

Las funciones y métodos que veremos más adelante también deben usar las anotaciones.

```
def plus(num1: int, num2: int) -> int:    # Tipos y devolviendo
    return 0

def show(value: str, excitement: int = 10) -> None:
    pass          # En este caso la función no devuelve nada
```

Cuando se vea el capítulo de POO, las clases son otro tipo más y las clases también pueden ser anotadas.

```
class BankAccount:
    def __init__(self, account_name: str,
                 initial_balance: int = 0) -> None:
        self.account_name: str = account_name
        self.balance: int = initial_balance

    def deposit(self, amount: int) -> None:
        self.balance += amount

account: BankAccount = BankAccount("Alice", 400)
def transfer(src: BankAccount,
            dst: BankAccount,
```

```
amount: int) -> None:
src.withdraw(amount)
dst.deposit(amount)
```

Variables y Constantes

En programación, una variable o una constante está formada por un espacio en memoria y un nombre simbólico (un identificador) que está asociado a dicho espacio. Esa área contiene una cantidad de información, es decir un valor. El nombre de la variable es la forma usual de referirse al valor almacenado: esta separación entre nombre y contenido permite que el nombre sea usado independientemente de la información exacta que representa. La diferencia principal es que la variable permite cambiar el contenido almacenado, mientras que la constante una vez definida, su valor nunca cambiará.

- ∞ Una variable es un contenedor de valores (generalmente un puntero a una dirección de memoria).
 - ∞ Las variables son objetos.
 - ∞ Las reglas de nombrado dictan que deben empezar por letra siempre y que pueden contener números y subrayado (_). Las reglas de estilo de nombrado de variables indican que serán siempre en minúsculas y separadas por un subrayado.
 - ∞ Las variables no son tipadas, el tipo del contenido de una variable viene determinado en tiempo de ejecución por el contenido, no en tiempo de compilación. Por tanto, no es necesario especificar un tipo a la hora de definir una variable.
 - ∞ No es necesario definir de forma explícita una variable, se define en el momento del primer uso cuando se le asigne valor, pero vamos a inicializar todas las variables que necesitemos antes de usarlas al comienzo del bloque correspondiente.
 - ∞ Una variable puede ser eliminada con la función **del nombre_variable**.
 - ∞ No se pueden usar las palabras clave de la sintaxis como nombres de variables.
 - ∞ El contenido de una variable es inmutable, por lo que cuando cambiamos su valor se crea un nuevo objeto con el nuevo valor, se cambia el puntero al nuevo objeto y se libera la memoria del valor (objeto) anterior.
- Existen tipos de datos inmutables: números, cadenas, tuplas que una modificación implicará una creación del nuevo objeto y una asignación a la variable; y tipos mutables que se modifica el objeto directamente: diccionario, lista, conjunto.
- ∞ Se pueden definir varias variables en una misma línea usando el operador coma (,) y usar el operador empaquetar (*) en la asignación.
 - ∞ No existen los valores constantes, se usa la nomenclatura en mayúsculas para indicar un valor que no cambia, pero no se implementa ningún control.

```
a, b = 3.5, "Hola"
c = d = 3
a, c = c, a
e, *f = 3, 4, 5, 6    # el asterisco es el resto de valores
*g, h = 7, 8, 9      # el asterisco es el resto de valores
del b
print(a, c, d, e, f, g, h) # 3 3.5 3 3 [4, 5, 6] [7, 8] 9
a, b = b, a # intercambio de variables
VALOR_CTE = 23 # es solo nomenclatura no se controla
print(VALOR_CTE)
VALOR_CTE = 22
print(VALOR_CTE)
```

Bibliotecas

La biblioteca estándar de Python es muy amplia y ofrece una gran cantidad de facilidades. La biblioteca contiene módulos incorporados que brindan acceso a las funcionalidades del sistema como entrada y salida

de archivos que serían de otra forma inaccesibles para los programadores en Python, así como módulos escritos en Python que proveen soluciones estandarizadas para los diversos problemas que pueden ocurrir en el día a día en la programación. Veremos elementos de esta biblioteca a lo largo de todo el curso.

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

Ejercicios

1. Escribe un programa que dé los “buenos días”.
2. Escribe un programa que calcule y muestre el área de un cuadrado de lado igual a 5.
3. Escribe un programa que calcule el área de un cuadrado cuyo lado se introduce por teclado.
4. Escribe un programa que lea dos números, calcule y muestre el valor de sus suma, resta, producto y división.
5. Escribe un programa que toma como dato de entrada un número que corresponde a la longitud de un radio y nos escribe la longitud de la circunferencia, el área del círculo y el volumen de la esfera que corresponden con dicho radio.
6. Escribe un programa que dado el precio de un artículo y el precio de venta real nos muestre el porcentaje de descuento realizado.
7. Escribe un programa que lea un valor correspondiente a una distancia en millas marinas y escriba la distancia en metros. Sabiendo que una milla marina equivale a 1.852 metros.
8. Calcular el consumo de un coche a los 100 kilómetros dados el gasto en dinero realizado, el precio del litro y el número de kilómetros recorridos.
9. Calcular el coste de la vida dados los precios de tres productos este año y el año pasado.
10. Escribir un programa que pregunte al usuario por el número de horas trabajadas y el coste por hora. Después debe mostrar por pantalla la paga que le corresponde.
11. Escribir un programa que lea un entero positivo, introducido por el usuario y después muestre en pantalla la suma de todos los enteros desde 1 hasta el número. La suma de los primeros enteros positivos puede ser calculada de la siguiente forma $n(n+1)/2$.
12. Escribir un programa que pida al usuario su peso (en kg) y estatura (en metros), calcule el índice de masa corporal, lo almacene en una variable y muestre por pantalla la frase Tu índice de masa corporal es <imc> donde <imc> es el índice de masa corporal calculado redondeado con dos decimales.
13. Escribir un programa que pida al usuario dos números enteros y muestre por pantalla la <n> entre <m> da un cociente <c> y un resto <r> donde <n> y <m> son los números introducidos por el usuario, y <c> y <r> son el cociente y el resto de la división entera respectivamente.
14. Una juguetería tiene mucho éxito en dos de sus productos: payasos y muñecas. Suele hacer venta por correo y la empresa de logística les cobra por peso de cada paquete así que deben calcular el peso de los payasos y muñecas que saldrán en cada paquete a demanda. Cada payaso pesa 112 g y cada muñeca 75 g. Escribir un programa que lea el número de payasos y muñecas vendidos en el último pedido y calcule el peso total del paquete que será enviado.
15. Imagina que acabas de abrir una nueva cuenta de ahorros que te ofrece el 4% de interés al año. Estos ahorros debido a intereses, que no se cobran hasta finales de año, se te añaden al balance final de tu cuenta de ahorros. Escribir un programa que comience leyendo la cantidad de dinero depositada en la cuenta de ahorros, introducida por el usuario. Después el programa debe calcular y mostrar por pantalla la cantidad de ahorros tras el primer, segundo y tercer años. Redondear cada cantidad a dos decimales.
16. Una panadería vende barras de pan a 3.49€ cada una. El pan que no es el día tiene un descuento del 60%. Escribir un programa que comience leyendo el número de barras vendidas que no son del día. Después el programa debe mostrar el precio habitual de una barra de pan, el descuento que se le hace por no ser fresca y el coste final total.

17. Escribir un programa que solicite al usuario ingresar un número con decimales y almacenarlo en una variable. A continuación, el programa debe solicitar al usuario que ingrese un número entero y guardararlo en otra variable. En una tercera variable se deberá guardar el resultado de la suma de los dos números ingresados por el usuario. Por último, se debe mostrar en pantalla el texto “El resultado de la suma es [suma]”, donde “[suma]” se reemplazará por el resultado de la operación.
18. Escribir un programa que solicite al usuario el ingreso de una temperatura en escala Fahrenheit (debe permitir decimales) y le muestre el equivalente en grados Celsius. La fórmula de conversión que se usa para este cálculo es: Celsius = $(5/9) * (Fahrenheit - 32)$.
19. Escribir un programa que solicite al usuario el ingreso de dos palabras, las cuales se guardarán en dos variables distintas. A continuación, almacenará en una variable la concatenación de la primera palabra, más un espacio, más la segunda palabra. Mostrará este resultado en pantalla.
20. Escribir un programa que solicite al usuario el ingreso de un texto y almacene ese texto en una variable. A continuación, mostrar en pantalla la primera letra del texto ingresado. Luego, solicitar al usuario que ingrese un número positivo menor a la cantidad de caracteres que tiene el texto que ingresó (por ejemplo, si escribió la palabra “HOLA”, tendrá que ser un número entre 0 y 4) y almacenar este número en una variable llamada índice. Mostrar en pantalla el carácter del texto ubicado en la posición dada por índice.
21. Escribir un programa que solicite al usuario que ingrese cuántos shows musicales ha visto en el último año y almacene ese número en una variable. A continuación, mostrar en pantalla un valor de verdad (True o False) que indique si el usuario ha visto más de 3 shows.
22. Escribir un programa que le solicite al usuario ingresar una fecha formada por 8 números, donde los primeros dos representan el día, los siguientes dos el mes y los últimos cuatro el año (DDMMAAAA). Este dato debe guardarse en una variable con tipo int (número entero). Finalmente, mostrar al usuario la fecha con el formato DD / MM / AAAA.
23. Escribir un programa para solicitar al usuario el ingreso de un número entero y que luego imprima un valor booleano dependiendo de si el número es par o no. Recordar que un número es par si el resto al dividirlo por 2, es 0.
24. Escribir un programa que le solicite al usuario su edad y la guarde en una variable. Que luego solicite la cantidad de artículos comprados en una tienda y la guarde en otra variable. Finalmente, mostrar en pantalla un valor de verdad (True o False) que indique si el usuario es mayor de 18 años de edad y además compró más de 1 artículo.
25. Escribir un programa que, dada una cadena de texto por el usuario, imprima True si la cantidad de caracteres en la cadena es un número impar, o False si no lo es.
26. Escribir un programa para pedir al usuario su nombre y luego el nombre de otra persona, almacenando cada nombre en una variable. Luego mostrar en pantalla un valor de verdad que indique si: los nombres de ambas personas comienzan con la misma letra o si terminan con la misma letra. Por ejemplo, si los nombres ingresados son María y Marcos, se mostrará True, ya que ambos comienzan con la misma letra. Si los nombres son Ricardo y Gonzalo se mostrará True, ya que ambos terminan con la misma letra. Si los nombres son Florencia y Lautaro se mostrará False, ya que no coinciden ni la primera ni la última letra.

U. T. 3 Estructuras de control.



Introducción

En esta unidad de trabajo vamos a conocer todas las estructuras de control de lenguaje, ya estamos familiarizadas con ellas a través de pseudocódigo y de diagramas de flujo, pero vamos a implementarlas en código.

**A la hora de programar, nos centraremos en hacer el código fácil de leer no más corto
La programación se empieza siempre en papel desarrollando el algoritmo**

Condiciones

Las condiciones forman parte de gran cantidad de estructuras de control, con lo que es imprescindible dominarlas correctamente para una adecuada programación.

Para crear las condiciones tenemos los siguientes operadores:

- ∞ Operadores de pertenencia: **in, not in**.
- ∞ Operadores de ser lo mismo: **is, not is**.
- ∞ Operadores de expresiones: **and** y **or** son operadores en cortocircuito, o que no siguen evaluando la expresión si no es necesario.
- ∞ Operador de negación: **not**.
- ∞ Operadores de comparación: **<, >, >=, <=, !=, ==**.
- ∞ Operador morsa a partir de la versión 3.8 (**:=**).

Una condición tiene siempre la estructura: **operando operador operando [[and| or] ...]** para aquellos operandos binarios. O tiene la forma **operador operando [[and| or] ...]** para los operadores unarios. El único operador unario que hemos visto es **not**.

En Python las condiciones se pueden encadenar de izquierda a derecha para formar una condición compleja: **5 < variable < 15** con el mismo significado que **variable > 5 and variable < 15**.

Ejemplo: Crear las siguientes condiciones.

- La variable numero mayor o igual que cinco
- La variable numero mayor de tres y menor o igual que uno
- La variable numero
- La variable numero no es verdadera y no es mayor de cinco
- La variable numero no es igual a "Hola"
- La variable numero no es igual a la variable saltos

Estructuras de selección o alternativas

Las estructuras alternativas se usan para realizar un conjunto de acciones en función de condiciones.

Condicional simple

```
if <condición>:  
    sentencias
```

Condicional simple

```
if <condición1>:  
    sentencias  
else:  
    sentencias
```

Condicional compuesta

```
if <condición1>:  
    sentencias  
elif <condición2>:  
    sentencias  
elif <condición3>:  
    sentencias  
else:  
    sentencias
```

Esta estructura de control nos permite realizar acciones en función de una o varias condiciones, que pueden ser simples o complejas uniéndolas a través de los operandos **and** y **or**. En caso de no ser verdadera ninguna condición se ejecutará la parte **else** si está presente.

Los bloques **elif** y **else** son optativos y pueden aparecer ambos o solo uno de ellos. Además, se puede repetir el bloque **elif** tantas veces como requiramos para testear las condiciones necesarias, **elif** es la abreviatura de **else if**. Python no tiene una estructura de control **switch-case**

como otros lenguajes, se implementa mediante la condicional compuesta.

Operador ternario

```
variable = Valor if <condición> else Valor2
```

En muchos lenguajes encontraremos el operador ternario (**:**), en Python no existe tal operador, pero la estructura **if** nos facilita un uso similar. En este caso

la variable adquirirá “Valor” si la <condición> es verdadera, si es falsa se le asignará “Valor2”.

```
if compra <= 100:  
    print("Pago en efectivo")  
elif compra > 100 and compra < 300: # 100 < compra < 300  
    print("Pago con tarjeta de débito")  
else:  
    print("Pago con tarjeta de crédito")
```

!!!!!! Estudiar Anexo III – Patrón estructural !!!!!!

Hacer los Ejercicios: 1 al 5

Estructuras de repetición

Introducción

Las estructuras de repetición realizan un número finito de vueltas sobre el código que contienen. Es muy importante esa característica, debe ser finito, ya que si no controlamos correctamente la condición de finalización podemos hacer que nuestro programa entre en un bucle cerrado o infinito.

```
fin=False  
while not fin:  
    print("Pago en efectivo")  
print("Fin") # Esta instrucción no se ejecutará nunca
```

Función range

La función **range** se usa para devolver una lista de números secuenciales. Esta función se utiliza para simular los bucles secuenciales de otros lenguajes. La sintaxis de la función es **range(inicio, fin, paso)**. Los parámetros se explican por sí solos, pero hay que tener cuenta que el parámetro **fin** no estará incluido en la lista final.

range(5)	# 0 1 2 3 4 5
range(5, 10)	# 5 6 7 8 9
range(0, 10, 3)	# 0 3 6 9
range(-10, -100, -30)	# -10 -40 -70

Bucles

Bucle for

```
for variable in lista:
    sentencias
else:
    sentencias
```

En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia **for** de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia, por eso se hace

imprescindible el uso de **range**. La parte **else** se ejecuta cuando termina la sentencia **for** siempre y cuando no se haya salido del bucle con una instrucción **break** que veremos a continuación, este bloque también admite la sentencia **continue**. Por último, indicar que la lista que se recorre se puede modificar dentro del bucle sin problemas.

```
for valor in range(5):
    print("El valor es", valor)
```

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print(w, len(w))
```

Bucle while

```
while <condicion>:
    sentencias
else:
    sentencias
```

La sentencia **while** se usa para la repetición la ejecución del bloque siempre que una expresión sea verdadera. Como en el caso anterior, la parte **else** se ejecutará siempre que no se salga del bucle a través de **break**. Es en este bucle donde hay que tener especial cuidado con la terminación, por lo que **siempre** modificaremos la condición dentro del bucle de manera que se llegue a algún fin. Este bucle

también admite las sentencias **break** y **continue** como el anterior.

```
fin, numero = False, 0
while not fin:
    print(numero)
    if numero == 5:
        fin = True
    numero += 1
```

Python no incorpora ningún otro tipo de bucle. Los bucles **for** tradicionales tipo C, o los tipos **do-while**, **until** o **do-until** no están presentes en la sintaxis del lenguaje. Es importante remarcar que ninguno de los bucles nombrados es necesario y en todas las situaciones se puede rescribir el código con los bucles que hay en Python.

Hacer los Ejercicios: 13, 16, 17, 18 y 19

Estructuras de salto

Python no incorpora ninguna sentencia de ruptura de secuencia parecida a **goto** de otros lenguajes, esta sentencia no hace más que introducir problemas y está completamente desaconsejada. En aquellos lenguajes que aparece, permite saltar de forma arbitraria a otro punto de la ejecución.

Break y continue

Las sentencias **break** y **continue** sirven para romper la secuencia actual de un bucle. En caso de la sentencia **break** termina completamente el bucle y con la sentencia **continue** salta al comienzo del bucle haciendo que las condiciones se reevalúen.

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break # termina el bucle de la x
        else:
            # loop fell through without finding a factor
            print(n, 'is a prime number')
```

```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue # salta al comienzo del bucle
    print("Found an odd number", num)
```

Hacer los Ejercicios: 22 al 26

Prueba y depuración de programas

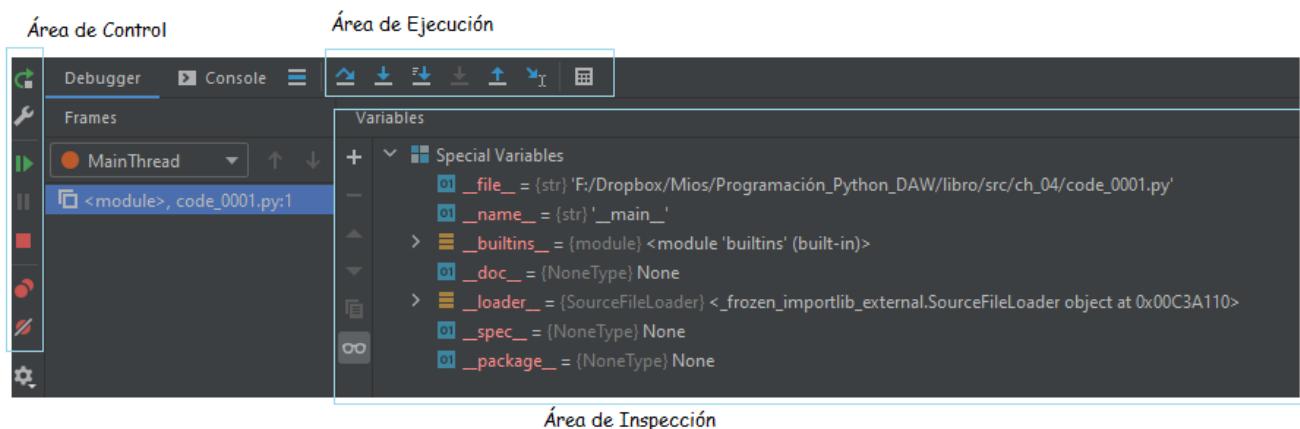
Todo programa deberá ser testado mediante un conjunto de pruebas que garanticen que sea operativo, cumpla con los requisitos estipulados e incluya comentarios aclaratorios.

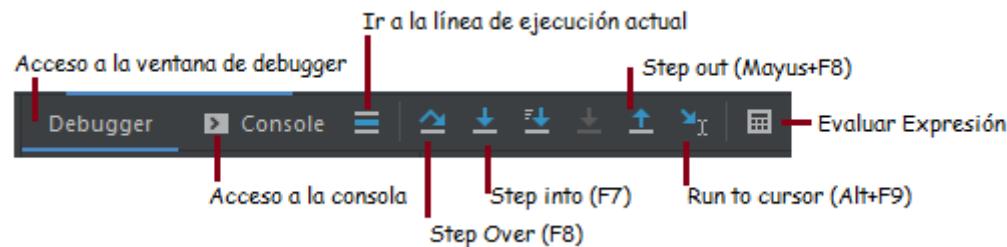
Al proceso mediante el cual se corrigen estos errores del programa se le conoce como depuración e implica una serie de tareas:

- ∞ La inspección del código.
- ∞ La utilización de baterías de prueba.
- ∞ La prueba de todos y cada uno de los módulos (pruebas unitarias).
- ∞ La corrección de los errores detectados, teniendo en cuenta que ante cada modificación del código pueden surgir nuevos errores.

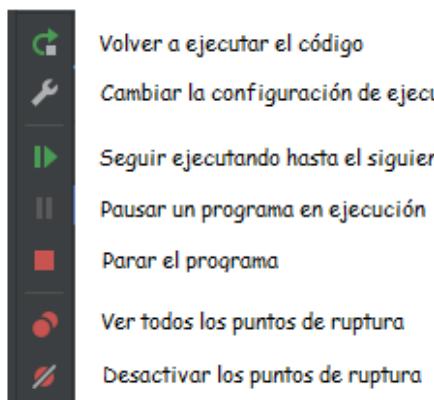
Entorno de depuración

Nuestro entorno es PyCharm, pero cualquier IDE tiene las mismas herramientas, incluso probablemente utilizará los mismos iconos para los mismos conceptos. La Ventana de depuración es la siguiente.



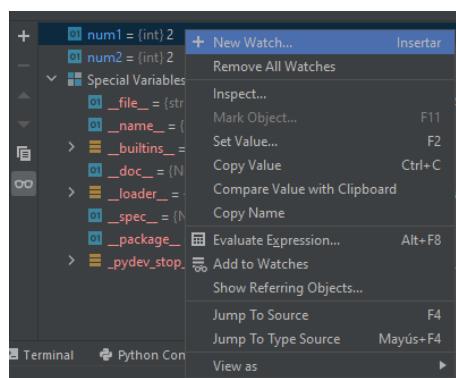
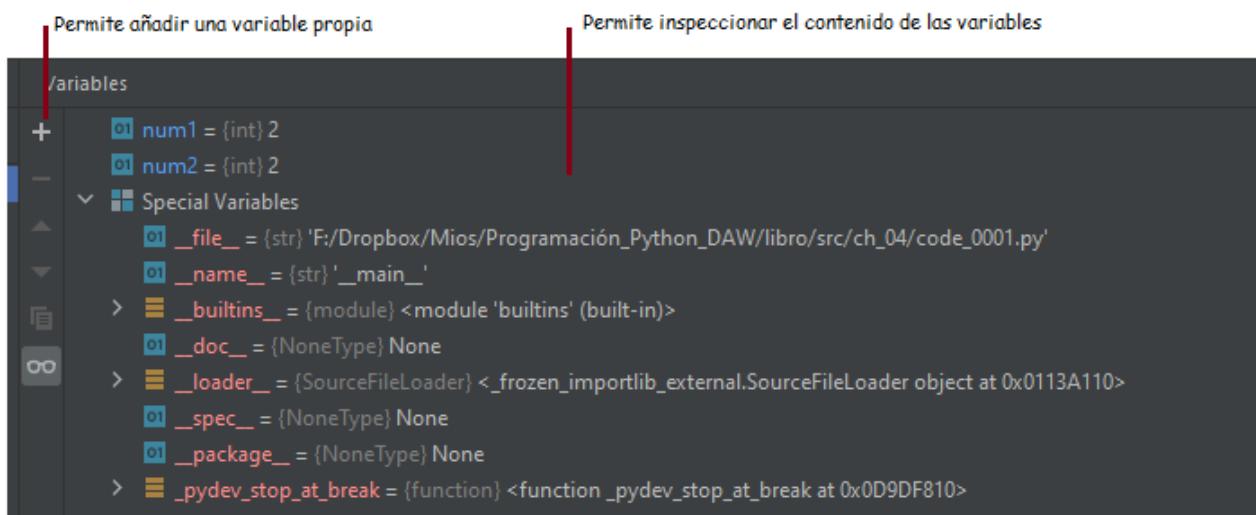


- ∞ **Debugger.** Permite acceder a la ventana de gestión.
- ∞ **Console.** Permite ver la salida del programa, se debe configurar la opción **Emulate terminal in output console** en las propiedades de ejecución del programa.



- ∞ **Step Over (F8).** Ejecuta la siguiente línea de código, si es una llamada a una función o método no entra y se ejecuta de una vez.
- ∞ **Step into (F7).** Ejecuta la siguiente línea de código, si es una función entra y se para en la primera línea de la función o método.
- ∞ **Step out (Mayus+F8).** Sale de la ejecución de la función actual.
- ∞ **Run tu cursor (Alt+F9).** Ejecuta hasta la posición actual del cursor de texto.
- ∞ **Evaluar expresión.** Permite introducir una expresión y ver su resultado.

Desde el área de inspección podemos acceder a las variables activas en ese momento y ver su contenido, desplegando cada miembro de un objeto.

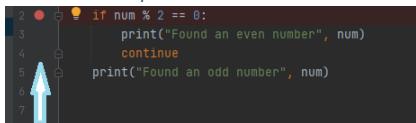


La ventana de Inspección permite algunas otras funciones interesantes al pulsar con el botón derecho del ratón, entre las que están cambiar el valor de una variable (**Set Value**), añadir a la lista de variables inspeccionadas (**Add to Watches**) y establecer cómo ver una variable (**View As**).

Ejemplo de depuración

Vamos a utilizar código Python para realizar ejercicios de depuración y entender todas las herramientas y facilidades que nos reporta el entorno. Python tiene incluido el depurador con lo que desde la consola Python también podríamos realizar labores de depuración, pero es altamente recomendable tener un entorno gráfico que nos facilite la tarea y nos muestre de forma unificada todos los datos del programa.

Punto de ruptura



```

2 ●  if num % 2 == 0:
3     print("Found an even number", num)
4     continue
5
6 ●  print("Found an odd number", num)
7

```

Un punto de ruptura es un marcador en línea de código ejecutable (no pueden ser comentarios) en la que parará el depurador al llegar a ella. Se establece pulsando en el área marcada por la flecha o situando el cursor en la línea correspondiente y pulsando **Ctrl+F8**.

Inspección



El entorno nos permite evaluar en cualquier momento que esté detenido cualquier expresión

usando el botón derecho de ratón y pulsando en **Evaluate expresión**. También podemos añadir una expresión o variable a la lista de inspecciones (variables a vigilar) que se mostrará su contenido en el área de Inspección.

Prácticas de depuración

Ejemplo: Copia el código del fichero

```

edad = int(input("Introduce tu edad:"))
fin = False
while not fin:
    if edad > 18:    # error >=
        fecha_nacimiento = int(input(
            "Introduce la fecha de nacimiento (YYYYMM):"))
        anio_actual = int(input(
            "Introduce el año de cálculo (YYYY):"))
        anio_nacimiento = fecha_nacimiento // 10000
        meses = fecha_nacimiento % 100
        edad_calculada_meses = edad * 12 + meses
        if edad_calculada > 0:
            print("Tu edad es:", edad, "en años")
            print("Tu edad es:", edad_calculada_meses,
                  "en meses")
        else:
            print("Datos erróneos")
    fin = input("Fin(Si)?") == 'si'

```

code_0001_a.py

- ∞ Cargar el código `code_0001_a.py`.
 - Establecer puntos de ruptura. Es importante determinar dónde poner los puntos de ruptura, generalmente al comienzo de un bloque o justo antes de la instrucción que no esté dando problemas.
 - Lanzar el programa. El programa se puede ejecutar en modo depuración igual que en modo normal, con el botón derecho del ratón, pero en vez de utilizar el comando Run, pulsaremos sobre el comando Debug. Si todo va bien el programa se ejecutará y entrará en modo depuración en el punto de ruptura.



La línea en la que está parado el programa se resalta en otro color y la ventana de depuración aparece, habilitándose los botones superiores de ejecución.

- c. A partir de este momento podemos ejecutar línea a línea, entrando en las funciones, inspeccionar o cambiar el valor de cualquier variable, o seguir la ejecución hasta la siguiente parada o finalización.
- d. Repetiremos el proceso hasta que el programa esté libre de errores.
- e. Encontrar los errores del programa. Las especificaciones son las siguientes: Para aquellas personas mayores de edad pedirá su fecha de nacimiento en formato YYYYMM (año cuatro dígitos_mes dos dígitos) encontrando la edad que tiene en años y en meses desde un año pedido. El programa saldrá cuando se teclee la palabra Fin.
 - i. Introducir en edad: 17, 18 y 19, ver qué ocurre, depurar con los tres valores.
 - ii. Con edad >18, introducir un valor de año y comprobar los resultados.
 - iii. Con edad>18, introducir una fecha posterior al año de cálculo.
 - iv. Comprobar la salida del programa, debe pedirse siempre y ser coherente.
 - v. Probar el programa con varias fechas y años.

Creación de casos de prueba

Todos los lenguajes tienen herramientas que permite la automatización de la ejecución de pruebas sobre nuestro código, Python no podía ser menos y tiene múltiples facilidades para la creación de casos de prueba. Este punto es complejo y no se abordará hasta más adelante una vez estudiado el paradigma orientado a objetos.

Documentación del código del programa

Introducción

La idea de tener que documentar un programa genera en el programador un rechazo inmediato. Son de esas cosas que sabemos que son buenas y debemos procurar, pero definitivamente no nos fascinan. Pero, la verdad es ésta: no importa lo fabuloso que sea un programa, nadie lo usará si no queda claro la manera de utilizarlo. En este sentido la documentación juega un rol primordial. Y aunque yo sea el único usuario de mi programa, resulta más sencillo entender y reutilizar lo que escribí hace seis meses si cuento con una documentación adecuada. Python proporciona un mecanismo bastante sencillo y conveniente para elaborar la documentación técnica de un programa.

Cadenas de documentación

En Python, un docstring o cadena de documentación es una literal de cadena de caracteres que se coloca como primer enunciado de un módulo, clase, método o función, y cuyo propósito es explicar su intención.

```
def promedio(a, b):
    'Calcula el promedio de dos números.'
    return (a + b) / 2
```

Un ejemplo más completo:

```
def formula_cuadratica(a, b, c):
    """Resuelve una ecuación cuadrática.

    Devuelve en una tupla las dos raíces que resuelven la
    ecuación cuadrática:

    ax^2 + bx + c = 0.

    Utiliza la fórmula general.

    Parámetros:
    a -- coeficiente cuadrático (debe ser distinto de 0)
    b -- coeficiente lineal
```

```

c -- término independiente

Excepciones:
ValueError -- Si (a == 0)

"""

if a == 0:
    raise ValueError(
        'Coeficiente cuadrático no debe ser 0.')
from cmath import sqrt
discriminante = b ** 2 - 4 * a * c
x1 = (-b + sqrt(discriminante)) / (2 * a)
x2 = (-b - sqrt(discriminante)) / (2 * a)
return (x1, x2)

```

La cadena de documentación en el segundo ejemplo es una cadena multi-líneas, la cual comienza y termina con triples comillas (""""). Aquí se pueden observar el uso de las convenciones establecidas en el [PEP 257](#) (Python Enhancement Proposals):

- ∞ La primera línea de la cadena de documentación debe ser una línea de resumen terminada con un punto. Debe ser una breve descripción de la función que indica los efectos de ésta como comando. La línea de resumen puede ser utilizada por herramientas automáticas de indexación; es importante que quepa en una sola línea y que esté separada del resto del docstring por una línea en blanco.
- ∞ El resto de la cadena de documentación debe describir el comportamiento de la función, los valores que devuelve, las excepciones que arroja y cualquier otro detalle que consideremos relevante.
- ∞ Se recomienda dejar una línea en blanco antes de las triples comillas que cierran la cadena de documentación.

Todos los objetos documentables (módulos, clases, métodos y funciones) cuentan con un atributo `__doc__` el cual contiene su respectivo comentario de documentación.

A partir de los ejemplos anteriores podemos inspeccionar la documentación de las funciones `promedio` y `formula_cuadratica` desde el **shell** de Python:

```

>>> promedio.__doc__
'Calcula el promedio de dos números.'
Sin embargo, si se está usando el shell de Python es mejor usar
la función help(), dado que la salida producida queda formateada
de manera más clara y conveniente:
>>> help(promedio)
Help on function promedio in module __main__:

promedio(a, b)
    Calcula el promedio de dos números.

```

Ciertas herramientas, por ejemplo, shell o editores de código, pueden ayudar a visualizar de manera automática la información contenida en los comentarios de documentación. La siguiente imagen muestra como el shell del ambiente de desarrollo integrado IDLE muestra la línea de resumen como descripción emergente (tool tip) al momento en que un usuario teclea el nombre de la función:

```

*Python Shell*
File Edit Debug Options Windows Help
Python 3.2.3 (default, Sep 25 2013, 18:22:43)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
>>> formula_cuadratica(
        (a, b, c)
    Resuelve una ecuación cuadrática.

Ln: 6 Col: 23

```

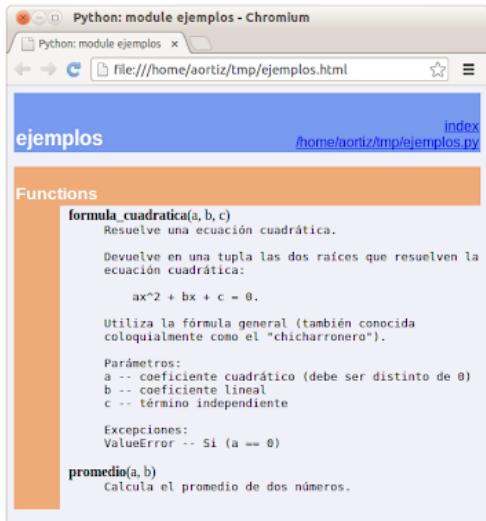
De esta manera un usuario puede tener a su alcance de manera sencilla toda la información que necesita para poder usar nuestras funciones.

Generando documentación en páginas de HTML

Los docstrings se pueden usar también para producir documentación en páginas de HTML que pueden ser consultadas usando un navegador de web. Para ello se usa el comando `pydoc` desde una terminal. Por ejemplo, si las dos funciones anteriores (`promedio` y `formula_cuadratica`) se encuentran en un archivo fuente llamado `ejemplos.py`, podemos ejecutar el siguiente comando en una terminal dentro del mismo directorio donde está el archivo fuente:

```
pydoc -w ejemplos
```

La salida queda en el archivo `ejemplos.html`, y así se visualiza desde un navegador.



La documentación de `pydoc` explica el resto de funciones que tiene esta librería.

<https://docs.python.org/3/library/pydoc.html>

Docstrings vs. comentarios

Un comentario en Python se inicia con el símbolo de número (#) y se extiende hasta el final de la línea. En principio los docstrings pudieran parecer similares a los comentarios, pero hay una diferencia pragmática importante: los comentarios son ignorados por el ambiente de ejecución de Python y por herramientas como `pydoc`; esto no es así en el caso de los docstrings.

A un nivel más fundamental hay otra diferencia aún más grande entre los docstrings y los comentarios, y ésta tiene que ver con la intención:

- ∞ Los docstrings son documentación, y sirven para entender qué hace el código.
- ∞ Los comentarios sirven para explicar *cómo* lo hace.

La documentación es para la gente que usa el código. Los comentarios son para la gente que necesita entender cómo funciona el código, posiblemente para extenderlo o darle mantenimiento.

Conclusiones

El uso de docstrings en Python facilita la escritura de la documentación técnica de un programa. Escribir una buena documentación requiere de disciplina y tiempo, pero sus beneficios se cosechan cuando alguien (quizás mi futuro yo dentro de seis meses) necesita entender qué hace nuestro software. Los docstrings no sustituyen otras buenas prácticas de programación, como son el uso apropiado de comentarios o el empleo de nombres descriptivos para variables y funciones, por lo que resulta importante utilizar todas estas prácticas de manera conjunta.

Cuestiones de Estilo

Una guía de estilo es un sencillo documento que recoge la forma en que los desarrolladores tienen que escribir su código. Si todos los desarrolladores siguen las recomendaciones de la guía, conseguiremos que nuestro proyecto tenga un estilo consistente. Normalmente, una buena guía de estilo debería:

- ∞ Promover que el código sea inteligible.
- ∞ Promover que el código esté organizado en unidades de trabajo comprensibles (funciones no más extensas de un cierto número de líneas, organización en párrafos, etc.).
- ∞ Definir las convenciones de nombres de variables, estilo de espaciados, etc.
- ∞ Indicar cómo comentar el código.

Las guías de estilo nos proporcionarán múltiples ventajas, entre las que se encuentra entenderlo mejor a nosotros y a cualquier programador; permitir un mejor mantenimiento y modificación; generar código legible y muchas otras.

Python, como todos los lenguajes, tiene una guía de estilo, en este caso se llama PEP-8 que la podemos encontrar (en español: <http://www.recursospthon.com/pep8es.pdf>) en:

<https://www.python.org/dev/peps/pep-0008/>

Entre las recomendaciones que implica esta guía podemos encontrar las siguientes:

- ∞ Usar cuatro espacios en vez de tabuladores.
- ∞ Escribir líneas de 79 caracteres máximo.
- ∞ Despues de la definición de una función añadir una linea en blanco.
- ∞ Usar los comentarios de una línea.
- ∞ Utilizar espacios alrededor de los operadores.
- ∞ El nombrado de clases debería ajustarse a: NombreClase
- ∞ El nombrado de funciones y variables debería ajustarse a: nombre_funcion

Ejercicio: Leer la guía de estilo y hacer un resumen de los puntos que nos parezcan más importantes, ponerla común en la clase y formar nuestra propia guía de estilo a seguir durante todo el curso.

Programación modular

Tiene por finalidad obtener algoritmos que se basan en la descomposición reiterada del problema inicial en subproblemas más simples. En esta descomposición una acción se dividirá en otras. El proceso se repite hasta que todas las acciones sean inmediatamente identificables.

Ejemplo

Fichar todos los libros de una biblioteca

1. Primer nivel:

Mientras quedan libros sin fichar Hacer

 Fichar un libro

 Fin Mientras

2. Segundo nivel:

 Fichar un libro:

 Coger un libro de los no fichados

 Darle un número de referencia

 Coger ficha

 Escribir en ella el nombre del libro

 Escribir el nombre del autor

 Escribir la editorial

 Escribir el número de referencia

 Duplicar la ficha

 Poner la ficha en el libro

 Poner la otra ficha en el fichero

3. Tercer nivel:

 Poner la otra ficha en el fichero:

 Mirar la primera ficha del fichero

 Mientras la ficha del fichero se anterior Hacer

 Pasar una ficha del fichero

 Fin Mientras

 Colocar la ficha en el fichero

Funciones en la programación modular

Python permite diversos paradigmas de programación y se pueden mezclar en un mismo fichero. En concreto el paradigma de programación modular determina crear códigos independientes que realizan una función y pueden o no retornar un valor: funciones y procedimientos. Para llamar a una función se utilizará su nombre y entre paréntesis se le pasarán los parámetros correspondientes.

Diferencia entre procedimientos y funciones

Formalmente una función y un procedimiento no es lo mismo. Una función es una sección de un programa que calcula un valor de manera independiente al resto del programa que tiene tres partes: parámetros de entrada, proceso o código y resultado. Por otro lado, un procedimiento solo tiene dos partes: entrada y proceso, no devuelve nada.

A nivel de código la diferencia principal es que en los procedimientos no encontraremos una sentencia **return**, mientras que en las funciones sí.

Bajo Python, no existen los procedimientos, incluso si omitimos la sentencia **return** en nuestra definición, el intérprete devolverá de forma predefinida **None** como valor de nuestra función.

Para Python como para el resto de lenguajes, una función es un bloque de código con autonomía propia de ejecución que podremos reutilizar. La mayor diferencia con los lenguajes tradicionales es que es un objeto con sus propiedades y métodos. Una función se puede definir en cualquier parte del código, aunque es recomendable estructurar el código en módulos.

Para que una función esté bien definida según la guía de estilo, ésta no debe ocupar más de veinticinco a treinta líneas de código.

Definición

```
def fib(n):      # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
    return False
```

La palabra reservada **def** se usa para definir funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente y deben estar con sangría. La sentencia **return** final es optativa y representa el valor devuelto por la función cuando es utilizada como parte derecha de una asignación.

Esta definición se encontrará dentro de una definición de clase si es un método o fuera de ella si es una función, el resto del apartado trataremos como si se encontrase dentro de una clase, pero omitiremos el primer parámetro.

La primera sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o docstrings para documentación.

Se pueden devolver varios valores separándolos con comas

```
def suma_multiplicacion(num_1, num_2):
    return num1 + num2, num_1 * num_2
```

Ámbito de las variables

El ámbito es la zona de código que tiene acceso una variable. Python, como todos los lenguajes, tiene varios ámbitos para definir las variables uno **local**, uno **nonlocal** y otro **global**. Por defecto estos ámbitos no se mezclan y no podemos hacer uso de uno dentro del otro. El ámbito local se corresponde con el nivel de función, el ámbito global con el de programa, el **nonlocal** con una función interna.

Para modificar una variable definida en el programa deberemos anteponer la palabra clave **global** y el nombre de la variable para poder usarla. Nunca podremos cambiar una variable local fuera de la función que la defina, aunque sí podremos acceder a ella sin especificar **global** si no la modificamos.

```
mi_var_global = 3

def mi_fun():
    global mi_var_global
    mi_var_global = 4

mi_fun()
print(mi_var_global)
```

En una clase no se debería acceder nunca a una variable **global**.

Parámetros

Un parámetro es un valor que se le pasa al método o función, un valor que espera para realizar su trabajo. Como Python define el tipo de una variable (parámetro en este caso) en tiempo de ejecución, no es necesario determinarlo en el parámetro, y no solo eso, el mismo parámetro aceptará cualquier tipo de datos para su ejecución, por lo que tendremos que realizar una conversión explícita si fuera necesario.

```

def suma(numero_1, numero_2):
    print(numero_1 + numero_2)

suma(2,3)
suma(2) # Error faltan parámetros

```

Parámetros de Entrada – Salida

El comportamiento común de un parámetro es como entrada, esto quiere decir que recibirá un dato almacenado en el parámetro enviado desde el contexto en el que se llama a la función, se utilizará dentro de la función y no se modificará.

Los parámetros de salida permiten modificar el valor de una variable externa a la función relacionada con el parámetro. De esta manera, si la función modifica el valor del parámetro, la variable externa relacionada a este parámetro se modificará también, inclusive después de que la función haya terminado de ejecutarse.

En Python todos los parámetros son de entrada.

Juego del ahorcado

Primer nivel

```

Acción Juego del Ahorcado
    Pedir la palabra al primer jugador
    Jugar con el segundo
Fin Acción

```

Segundo nivel

```

Acción Pedir la palabra al primer jugador
    Imprimir "Palabra?"
    Leer Palabra
    Borrar la pantalla
Fin Acción

```

```

Acción Jugar con el segundo
    numero_fallos = 0
    letras_faltan = palabra
    letras_adivinadas= "-" * longitud(palabra)
    Mientras no_acabe_el_juego Hacer
        Pedir letra al segundo jugador
        Evaluar jugada
    Fin mientras
Fin Acción

```

no_acabe_el_juego será cuando numero_fallos <= 10 y letras_adivinadas <> palabra

Tercer nivel

```

Acción Pedir letra al segundo jugador
    Imprimir letras_acertadas, "Letra?"
    Leer letra
Fin Acción

```

```

Acción Evaluar jugada
    numero_aciertos_letra = 0

```

```

Para índice = 0 hasta longitud(letras_faltan) Hacer
    Si letras_faltan[índice] = letra Entonces
        letras_acertadas[índice] = letra
        letras_faltan[índice] = "-"
        numero_aciertos_letra += 1
    Fin si
Fin para
Si numero_aciertos_letra = 0 Entonces
    Numero_fallos += 1
Fin Si
Fin Acción

```

Ejercicio: Implementarlo en Python: code_0001_b.py

Ámbito de las variables

El ámbito es la zona de código que tiene acceso una variable. Python, como todos los lenguajes, tiene varios ámbitos para definir las variables **uno local**, **uno nonlocal** y **otro global**. Por defecto estos ámbitos no se mezclan y no podemos hacer uso de uno dentro del otro. El ámbito local se corresponde con el nivel de función, el ámbito global con el de programa, el **nonlocal** con una función interna.

Para modificar una variable definida en el programa deberemos anteponer la palabra clave **global** y el nombre de la variable para poder usarla. Nunca podremos cambiar una variable local fuera de la función que la defina, aunque sí podremos acceder a ella sin especificar **global** si no la modificamos.

```

mi_var_global = 3

def mi_fun():
    global mi_var_global
    mi_var_global = 3
    print(mi_var_global) """Print a Fibonacci series up to n."""

```

Del mismo modo, si dentro de una función interna a otra, queremos acceder al bloque de la función padre deberemos usar la palabra clave **nonlocal**, que nos da acceso al bloque inmediatamente superior, como en el ejemplo siguiente.

```

g = 5
def miFunc2():
    global g
    g = 6
    b = 6
    def miInterna():
        nonlocal b # accedemos a la b externa
        global g
        b = 7
        g = 8
    miInterna()
    print(b, g)

miFunc2() # 7,8
print(g) # 8

```

Recomendaciones

- Crear los algoritmos en papel
- Buscar código repetido y crear funciones o procedimientos con dicho código
- Se deben devolver uno o dos elementos en la función
- En caso de devolver más crear un objeto para contenerlo (listas, tuplas, clase...)
- Deben tener un tamaño corto
- Los parámetros siempre serán de entrada, no utilizarlos para salida
- Tendrán un único punto de entrada y un único punto de salida (return), la última línea

Ejercicios

1. Escribe un programa que pide la edad por teclado y nos muestra el mensaje de "Eres mayor de edad" solo si lo somos.
2. Escribe un programa que pide la edad por teclado y nos muestra el mensaje de "eres mayor de edad" o el mensaje de "eres menor de edad".
3. Escribe un programa que lee dos números, calcula y muestra el valor de su suma, resta, producto y división. (Ten en cuenta la división por cero).
4. Escribe un programa que lee 2 números y muestra el mayor.
5. Escribe un programa que lee un número y me dice si es positivo o negativo, consideraremos el cero como positivo.
6. Escribe un programa que lee dos números y los visualiza en orden ascendente.
7. Escribe un programa que lee dos números y nos dice cuál es el mayor o si son iguales.
8. Escribe un programa que lea tres números distintos y nos diga cuál es el mayor.
9. Escribe un programa que lea una calificación numérica entre 0 y 10 y la transforma en calificación alfabética, escribiendo el resultado.
 - de 0 a <3 Muy Deficiente.
 - de 3 a <5 Insuficiente.
 - de 5 a <6 Bien.
 - de 6 a <9 Notable
 - de 9 a 10 Sobresaliente
10. Escribe un programa que recibe como datos de entrada una hora expresada en horas, minutos y segundos que nos calcula y escribe la hora, minutos y segundos que serán, transcurrido un segundo.
11. Escribe un programa que calcula el salario neto semanal de un trabajador en función del número de horas trabajadas y la tasa de impuestos de acuerdo a las siguientes hipótesis:
 - Las primeras 35 horas se pagan a tarifa normal.
 - Las horas que pasen de 35 se pagan a 1,5 veces la tarifa normal.
 - Las tasas de impuestos son:
 - Los primeros 500 euros son libres de impuestos.
 - Los siguientes 400 tienen un 25% de impuestos.
 - Los restantes un 45% de impuestos.
12. Implementa un programa que pida al usuario por teclado sus datos personales en orden apellido1-apellido2-nombre y los muestre por pantalla en orden nombre-apellido1-apellido2.
13. Desarrolla un programa que, una vez ejecutado, lea información desde teclado hasta que reciba la cadena de entrada "FIN".
14. Implementa un programa que reciba como parámetro un dígito correspondiente a un año y calcula si es o no bisiesto.
15. Escribe un programa calculadora que reciba como parámetros dos enteros (num1 y num2) desde su llamada y que los muestre por pantalla. A continuación, se deberá solicitar al usuario que seleccione la operación matemática a realizar entre los dos números. Las operaciones que implementará serán suma, resta, multiplicación, división y potencia.
16. Desarrolla un programa que pregunte por pantalla un número N y que, una vez leído, imprima una pirámide N filas.

```
Dime un número para realizar su pirámide: 6
*
***
*****
*****
*****
*****
*****
```

17. Utilizando la sentencia de control **if**, implementa un programa que pida un número por teclado y que indique por pantalla a qué mes se corresponde.
18. Realiza un programa que muestre por pantalla los 20 primeros números naturales (1, 2, 3... 20).
19. Realiza un programa que muestre los números pares comprendidos entre el 1 y el 200. Para ello utiliza un contador y suma de 2 en 2.
20. Realiza un programa que muestre los números pares comprendidos entre el 1 y el 200. Esta vez utiliza un contador sumando de 1 en 1.
21. Realiza un programa que muestre los números desde el 1 hasta un número N que se introducirá por teclado.
22. Realiza un programa que lea un número positivo N y calcule y visualice su factorial N!. Siendo el factorial:
$$0! = 1$$
$$1! = 1$$
$$2! = 2 * 1$$
$$3! = 3 * 2 * 1$$
$$N! = N * (N-1) * (N-2) \dots * 3 * 2 * 1$$
23. Realiza un programa que lea 10 números no nulos y luego muestre un mensaje de si ha leído algún número negativo o no.
24. Realiza un programa que lea 10 números no nulos y luego muestre un mensaje indicando cuántos son positivos y cuantos negativos.
25. Realiza un programa que lea una secuencia de números no nulos hasta que se introduzca un 0, y luego muestre si ha leído algún número negativo, cuantos positivos y cuantos negativos.
26. Realiza un programa que calcule y escriba la suma y el producto de los 10 primeros números naturales.
27. Realiza un programa que lea una secuencia de notas (con valores que van de 0 a 10) que termina con el valor -1 y nos dice si hubo o no alguna nota con valor 10.
28. Realiza un programa que sume independientemente los pares y los impares de los números comprendidos entre 100 y 200, y luego muestra por pantalla ambas sumas.
29. Realiza un programa que calcule el valor A elevado a B (A^B) sin hacer uso del operador de potencia (^), siendo A y B valores introducidos por teclado, y luego muestre el resultado por pantalla.
30. Realiza un programa donde el usuario "piensa" un número del 1 al 100 y el ordenador intenta adivinarlo. Es decir, el ordenador irá proponiendo números una y otra vez hasta adivinarlo (el usuario deberá indicarle al ordenador si es mayor, menor o igual al número que ha pensado).
31. Realiza un programa que dada una cantidad de euros que el usuario introduce por teclado (múltiplo de 5 €) mostrará los billetes de cada tipo que serán necesarios para alcanzar dicha cantidad (utilizando billetes de 500, 200, 100, 50, 20, 10 y 5). Hay que indicar el mínimo de billetes posible. Por ejemplo, si el usuario introduce 145 el programa indicará que será necesario 1 billete de 100 €, 2 billetes de 20 € y 1 billete de 5 € (no será válido por ejemplo 29 billetes de 5 que, aunque sume 145 € no es el mínimo número de billetes posible).
32. Realiza un programa que cuente los múltiplos de 3 desde el 1 hasta un número que introducimos por teclado.

```
run:  
Dime un número: 13  
Cantidad de multiplos de 3: 4
```

33. Realiza un programa en que pida un número entero positivo y nos diga si es primo o no.

```
run:  
Dime un número: 13  
Es primo.
```

34. Realiza un programa que lea y acepte únicamente aquellos que sean mayores que el último dado. La introducción de números finaliza con la introducción de un 0. Al final se mostrará:

El total de números introducidos, excluido el 0.

El total de números fallados.

```
Dime un número inicial: 20  
Dime un número: 21  
Dime un número: 8  
Fallo es menor.  
Dime un número: 15  
Dime un número: 10  
Fallo es menor.  
Dime un número: 30  
Dime un número: 0  
Total de números introducidos: 6  
Números fallados: 2
```

35. Realiza un programa para calcular la suma de los cuadrados de los 5 primeros números naturales.

36. Realiza un programa que lea un número y a continuación escriba el carácter "*" tantas veces igual al valor numérico leído. En aquellos casos en que el valor leído no sea positivo se deberá escribir un único asterisco.

```
run:  
Dime un número: 8  
*****
```

37. Realiza un programa que pida un número entero N entre 0 y 20 y luego muestre por pantalla los números desde 1 hasta N, uno en cada línea, repitiendo cada número tantas veces como su valor.

```
run:  
Dime un número: 5  
1  
22  
333  
4444  
55555
```

38. Realiza un programa que pida dos números enteros A y B, siendo B mayor que A. Luego visualiza los números desde A hasta B e indicar cuantos hay que sean pares. Ejemplo:

```
run:  
Dime un número: 5  
Dime otro número mayor al anterior: 11  
5 6 7 8 9 10 11  
La cantidad de pares son: 3
```

39. Crea un programa que pida al usuario un número entero y muestre su cuadrado. Se repetirá mientras el usuario introduzca un número distinto de cero.

40. Crea un programa que muestre la "tabla de multiplicar del 5", usando "while" y después "for"

41. Crea un programa que muestre los números del 1 al 20, excepto el 15, usando "for" y "continue".

42. Crea un programa que pida al usuario su login y su contraseña. Se repetirá hasta que el usuario introduzca

- como login "1809" y como contraseña "1234". Hazlo con while.
43. Crea un programa que pida un número de tipo byte al usuario y escriba en pantalla un cuadrado formado por asteriscos, que tendrá como alto y ancho la cantidad introducida por el usuario. Hazlo de dos formas, usando "while" y "for".
44. Contar el número de veces que aparece una letra en una frase, pedidas ambas al usuario.
45. Similar al anterior, pero a partir de una posición inicial pedida al usuario.
46. Encontrar si una subcadena está en una cadena, pedidas al usuario. Devolverá la posición de la subcadena en caso de estar o Fase si no lo está, sin usar slices [::].
47. Crear la cadena inversa a dos cadenas concatenadas que se pedirán al usuario sin usar [::-1].
48. A partir de una cadena que tiene separadores dividirla y mostrarla en sus partes. Se pedirá al usuario tanto el separador como la cadena. Ejemplo: separador: - - cadena: la-casa-alta → mostrará: la casa alta
49. Crear un menú que termine cuando se pulse salir. Tendrá tres opciones más que mostrará solo el texto de la opción seleccionada.
50. A partir de la velocidad inicial de un móvil pedida al usuario, se calculará la distancia en los n primeros segundos (pedidos al usuario) mostrando la distancia segundo a segundo.
51. Dibujar con letras N y B un tablero de ajedrez
52. Hacer una función que intercambie dos variables globales
53. Hacer una función que intercambie dos variables que se les pasa
54. Hacer una función que valide un argumento, de tal manera que devolverá verdadero si está entre 5 y 50, en otro caso devolverá falso.
55. Hacer una función que devuelva el valor absoluto de un número.
56. Hacer una función que convierta coordenadas polares a (radio y ángulo) en rectangulares (x,y). $x = \text{radio} * \cos(a)$, $y = \text{radio} * \sin(a)$.
57. Crear una función que nos valide una fecha.
58. Hacer el juego de MasterMind para cuatro columnas y seis fichas (ABCDEF). Se eligen al azar 4 de las fichas con repetición y se colocan en orden. El jugador colocará como desee cuatro fichas y se darán como pista las que están bien colocadas con un + y las que están, pero mal colocadas con un -. El juego termina cuando se colocan todas bien colocadas.

Elección: ACBC

El usuario pone	Bien	Mal	No están	Pistas
BBBB	--B-	----	BB-B	+
ABCD	A---	-BC-	---D	++-
CCEE	-C--	C---	--EE	+-
EECC	---C	--C-	EE--	+-
ACCB	AC--	--CB	----	++-

Ampliación

59. Diseña un programa que calcule el máximo de 5 números enteros. Intenta resolverlo con un ((candidato a valor máximo)) que se va actualizando al compararse con cada número.
60. Diseña un programa que, dados cinco números enteros, determine cuál de los cuatro últimos números es más cercano al primero. (Por ejemplo, si el usuario introduce los números 2, 6, 4, 1 y 10, el programa responder 'a que el número más cercano al 2 es el 1.).
61. Un vector en un espacio tridimensional es una tripleta de valores reales (x, y, z). Deseamos confeccionar un programa que permita operar con dos vectores. El usuario verá en pantalla un menú con las siguientes opciones:
- 1) Introducir el primer vector
 - 2) Introducir el segundo vector
 - 3) Calcular la suma
 - 4) Calcular la diferencia
 - 5) Calcular el producto escalar
 - 6) Calcular el producto vectorial

- 7) Calcular el 'ángulo (en grados) entre ellos
- 8) Calcular la longitud
- 9) Finalizar

Operación	Cálculo
Suma: $(x_1, y_1, z_1) + (x_2, y_2, z_2)$	$(x_1 + x_2, y_1 + y_2, z_1 + z_2)$
Diferencia: $(x_1, y_1, z_1) - (x_2, y_2, z_2)$	$(x_1 - x_2, y_1 - y_2, z_1 - z_2)$
Producto escalar: $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2)$	$x_1x_2 + y_1y_2 + z_1z_2$
Producto vectorial: $(x_1, y_1, z_1) \times (x_2, y_2, z_2)$	$(y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$
Ángulo entre (x_1, y_1, z_1) y (x_2, y_2, z_2)	$\frac{180}{\pi} \cdot \arccos \left(\frac{x_1x_2 + y_1y_2 + z_1z_2}{\sqrt{x_1^2 + y_1^2 + z_1^2} \sqrt{x_2^2 + y_2^2 + z_2^2}} \right)$
Longitud de (x, y, z)	$\sqrt{x^2 + y^2 + z^2}$

62. Implementa un programa que lea de teclado una cadena que representa un número binario. Si algún carácter de la cadena es distinto de '0' o '1', el programa advertirá al usuario de que la cadena introducida no representa un número binario y pedirá de nuevo la lectura de la cadena. En caso de ser correcta mostrará el valor binario.
63. Hay un tipo de pasatiempos que propone descifrar un texto del que se han suprimido las vocales. Por ejemplo, el texto ((.n .jmpl. d. p.s.t..mp.s)), se descifra sustituyendo cada punto con una vocal del texto. La solución es ((un ejemplo de pasatiempos)). Diseña un programa que ayude al creador de pasatiempos. El programa recibirá una cadena y mostrará otra en la que cada vocal ha sido reemplazada por un punto. If letra in "aeiou" remplazar por .
64. Una de las técnicas de criptografía más rudimentarias consiste en sustituir cada uno de los caracteres por otro situado n posiciones más a la derecha. Si n = 2, por ejemplo, sustituiremos la ((a)) por la ((c)), la ((b)) por la ((e)), y así sucesivamente. El problema que aparece en las últimas n letras del alfabeto tiene fácil solución: en el ejemplo, la letra ((y)) se sustituirá por la ((a)) y la letra ((z)) por la ((b)). La sustitución debe aplicarse a las letras minúsculas y mayúsculas y a los dígitos (el ((0)) se sustituye por el ((2)), el ((1)) por el ((3)) y así hasta llegar al ((9)), que se sustituye por el ((1))). Diseña un programa que lea un texto y el valor de n y muestre su versión criptografía.
65. Hacer un programa que calcule la corona circular pidiendo los dos radios, pero usando una función que calcule el área de un círculo.
66. Hacer un programa que pida dos listas y calcule la suma vectorial y el producto definido según el siguiente ejemplo usando una función para cada operación

$$(a,b,c) + (d,e,f) \rightarrow (a+d, b+e, c+f)$$

$$(a,b,c) * (d,e,f) \rightarrow a*d+b*e+c*f$$
67. Hacer un programa que calcule el máximo y el mínimo de 5 elementos guardados en una lista
 - > con una función para max y otra para min
 - > con una función solo y un parámetro que indique si es máx o min
68. Hacer un programa que pida dos conjuntos de 5 elementos (los conjuntos no tienen elementos repetidos) y nos haga la unión y la intersección con funciones
69. Encontrar todos los números que son amigos hasta un número pedido. Dos números son amigos si la suma de sus divisores es igual al otro número y viceversa:

$$284 \rightarrow 1+2+4+71+142 = 220$$

$$220 \rightarrow 1+2+4+5+10+11+20+22+44+55+110 = 284$$
70. Diseña un programa que permita jugar a dos personas al tres en raya.

Opcional: Hacer los ejercicios de ampliación

U. T. 4 P.O.O.



Introducción

La programación Orientada a objetos se define como un paradigma de la programación, una manera de programar específica, donde se organiza el código en unidades denominadas clases, de las cuales se crean objetos que se relacionan entre sí para conseguir los objetivos de las aplicaciones. La programación Orientada a objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación.

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser reutilizados se creó la posibilidad de utilizar módulos. El primer módulo existente fue la función, que somos capaces de escribir una vez e invocar cualquier número de veces.

Sin embargo, la función se centra mucho en aportar una funcionalidad dada, pero no tiene tanto interés con los datos. Es cierto que la función puede recibir datos como parámetros, pero los trata de una forma muy volátil. Simplemente hace su trabajo, procesando los parámetros recibidos y devuelve una respuesta.

Con la Programación Orientada a Objetos se buscaba unificar los datos y su gestión en una única estructura, creando unas mejores condiciones para poder desarrollar aplicaciones cada vez más complejas, sin que el código se volviera un caos. Además, se pretendía dar unas pautas para realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, de manera que consigamos que el código se pueda reutilizar.

la Programación Orientada a Objetos, un paradigma de programación que permite desarrollar aplicaciones complejas manteniendo un código más claro y manejable que otros paradigmas anteriores

Cómo se piensa en clases y objetos

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo, vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además, tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

Pues en un esquema POO "el coche" sería lo que se conoce como "Clase". Sus características, como el color o el modelo, serían propiedades y las funcionalidades asociadas, como ponerse en marcha o parar, serían métodos.

La clase es como un libro, que describe como son todos los objetos de un mismo tipo. La clase coche describe cómo son todos sus coches, qué propiedades tienen y qué funcionalidades deben poder realizar. A partir de una clase podemos crear cualquier número de objetos de esa clase. Un coche rojo que es de la marca Ford y modelo Fiesta, otro verde que es de la marca Seat y modelo Ibiza.

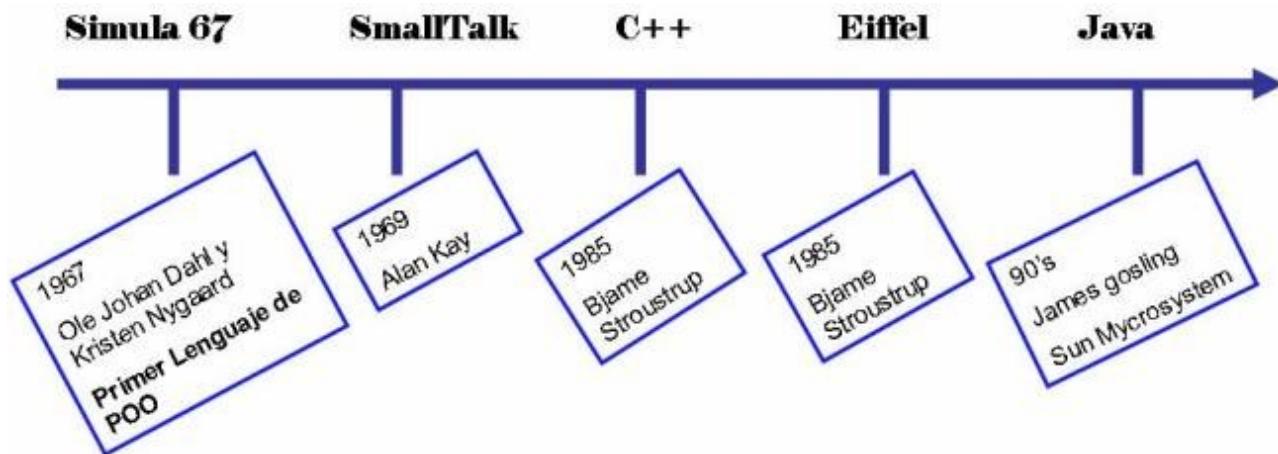
Identificar las clases es una de las tareas más importantes en la programación orientada a objetos y de esta tarea dependerá la calidad del producto final.

Orígenes

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard, del Centro de Cómputo Noruego en Oslo. En este centro se trabajaba en simulaciones de naves y la idea surgió al agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus propios datos y comportamientos. Fueron refinados más tarde en Smalltalk, desarrollado en Simula en Xerox PARC (cuya primera versión fue escrita sobre Basic) pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "sobre la marcha" (en tiempo de ejecución) en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos se fue convirtiendo en el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominio fue consolidado gracias al auge de las Interfaces gráficas de usuario, para las cuales la programación orientada a objetos está particularmente bien adaptada. En este caso, se habla también de programación dirigida por eventos. Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp y Pascal, entre otros.

La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y en la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros", por su parte, carecían de las características que muchos programadores se habían acostumbrado a utilizar. Para evitar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características imperativas de maneras "seguras". El Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos, pero ha sido reemplazado por Java, en gran parte debido a la aparición de Internet y a la implementación de la máquina virtual de Java en la mayoría de navegadores. PHP en su versión 5 se modificó, soportando una orientación completa a objetos, cumpliendo todas las características propias de la orientación a objetos.



Características de la POO

La POO posee muchas e importantes características que aporta con respecto a otros paradigmas para el desarrollo de aplicaciones, entre las que podemos destacar las siguientes:

- ∞ Permite enfocar los problemas de una forma mucho más intuitiva de lo que lo hacen otros paradigmas.
- ∞ Mientras que, en la programación estructurada datos y código están totalmente separados, en la POO se encuentran unidos en una única entidad.
- ∞ Se persigue trabajar con un modelo más cercano al mundo real en el que los diferentes objetos interaccionan entre sí.
- ∞ Permite generar programas modulares y muy escalables.
- ∞ La mejor opción para conseguir una buena experiencia de usuario.
- ∞ Su capacidad para permitir reutilizar código sin necesidad de reescribirlo nuevamente facilitó mucho la tarea de los programadores, que vieron en esta tendencia una gran oportunidad de rentabilizar el esfuerzo realizado.

Las propiedades de la POO son:

- ∞ **Abstracción.** Es un proceso mental por el que se ignoran las características de algo, quedándonos con lo que realmente nos importa. La abstracción es algo que hacemos constantemente los humanos en nuestro día a día. Si no lo realizáramos nuestro cerebro se colapsaría con toda la información que nos rodea. Una forma de manejar la complejidad del software es conseguir escribir código de tal manera que permita la abstracción.

- ∞ **Encapsulamiento.** Es el proceso por el cual se integran en una única estructura los datos y los mecanismos de gestión sobre esos datos, creando un todo. Una clase se compone tanto de variables (atributos) como de funciones y procedimientos (métodos).
- ∞ **Ocultación.** Hay una zona oculta al definir las clases (zona privada) que sólo es utilizada por esa clase y por alguna clase relacionada. Hay una zona pública (llamada también interfaz de la clase) que puede ser utilizada por cualquier parte del código. Una clase define la visibilidad de sus datos y métodos para que otras clases tengan acceso a ellos y tener un control total sobre la gestión de los mismos.
- ∞ **Polimorfismo.** Cada método de una clase puede tener varias definiciones distintas.
- ∞ **Herencia.** Una clase puede heredar propiedades (atributos y métodos) de otra creando de esta manera una jerarquía de clases. La herencia se puede hacer desde un único padre (herencia simple) o desde varios padres (herencia múltiple).

Ventajas y desventajas de la POO

Ventajas	Desventajas
Reusabilidad	Cambio en la forma de pensar, más difícil
Facilidad de desarrollo y mantenimiento	Curva de aprendizaje más larga
Facilidad de modificación	Ejecución más lenta
Aislamiento de los errores	No todos los problemas se pueden resolver con este enfoque
Reducción de código redundante	No es productivo para tareas simples
Fácil entendimiento de la lógica	
Cada componente tiene independencia de los demás	

Conceptos Fundamentales POO

Para hacer un desarrollo en POO debemos saber los conceptos fundamentales de la misma y los principios de programación. Según Robert C. Martin existen cinco principios básicos que constituyen la programación orientada a objetos son:

- ∞ **Principio de responsabilidad única.** Cada clase debe tener una única responsabilidad, y esta responsabilidad debe estar contenida únicamente en esa clase. Cada responsabilidad es el eje y la razón de cambio. Para contener la propagación del cambio, se deben separar las responsabilidades.
- ∞ **Principio de abierto-cerrado.** Una entidad (clase, módulo, función, etc.) debe quedarse abierta para su extensión, pero cerrada para su modificación. Si un cambio impacta a varios módulos, entonces la aplicación no está bien diseñada. Se deben diseñar módulos que procuren no cambiar y así, reutilizar el código más adelante(extensión).
- ∞ **Principio de sustitución.** Si en alguna parte de un programa se utiliza una clase, y esta clase es extendida, se puede utilizar cualquiera de las clases hijas sin que existan modificaciones en el código. Cada clase que hereda de otra puede usarse como sus padres sin necesidad de conocer las diferencias entre ellas.
- ∞ **Principio de segregación de interfaz.** Hace referencia a que muchas interfaces cliente específicas son mejores que una interfaz de propósito general. Se aplica a una interfaz amplia y compleja para dividirla en otras más pequeñas y específicas, de tal forma que cada cliente use solo aquella que necesite pudiendo así ignorar al resto.
- ∞ **Principio de inversión de dependencias.** Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.



La herramienta **Dia** es muy útil para hacer el diseño bajo POO

Pasaremos a ver los conceptos fundamentales de la POO en los siguientes apartados.

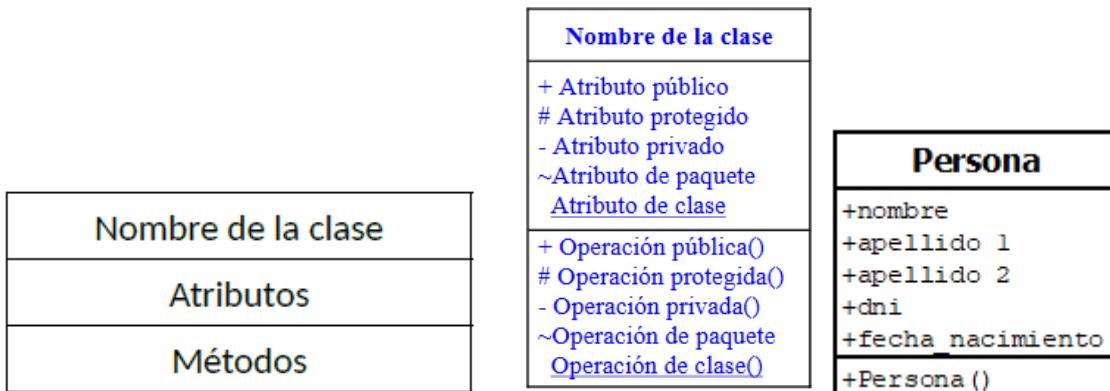
Clases

Una clase describe un grupo de objetos que contienen una información similar (atributos) y un comportamiento común (métodos), este conjunto de objetos se denomina **interfaz**. El interfaz público es el conjunto de atributos y métodos accesibles a los demás objetos, es muy importante definirlo correctamente ya que si cambiamos la firma (parámetros y tipos de parámetros) dejaremos a los objetos que los utilicen inservibles.

Las definiciones comunes (nombre de la clase, los nombres de los atributos, y los métodos) se almacenan una única vez en cada clase, independientemente de cuántos objetos de esa clase estén presentes en el sistema. Una clase es como un molde a partir de cual se pueden crear objetos.

Antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

- ∞ **Propiedades.** Las variables miembros de la clase.
- ∞ **Métodos.** Las funciones miembros de la clase.
- ∞ **Visibilidad.** Tanto atributos como métodos tienen una visibilidad con respecto a ella misma y a las demás clases del programa.



Propiedades o atributos

Una propiedad es una característica única del objeto, se asemejan a las variables de la programación estructurada, recoge información de la clase. La propiedad puede contener cualquier valor, incluido otro objeto de una clase. Las propiedades pueden ser reales o generadas. Las propiedades reales almacenan valores en memoria, las propiedades generadas simulan una propiedad a través de un método. El valor de estas últimas propiedades es la capacidad de control que generan y la posibilidad de crear valores calculados complejos, no solo los almacenados. En el ejemplo anterior las propiedades serían: *dni, nombre, apellido1, apellido2 y fecha_nacimiento*.

Métodos

Una clase es una estructura de datos que incluye las operaciones necesarias para gestionarla. Esas operaciones son los llamados métodos. Un método no es más que una función o procedimiento de gestión de la clase que puede ser utilizado por esta o por otras. En el ejemplo anterior, el método sería *Persona()*.

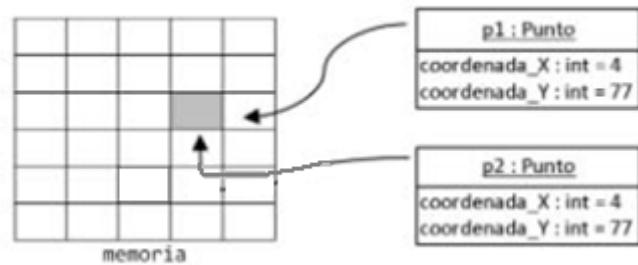
Todas las clases implementan métodos, pero hay dos que se implementarán siempre independientemente de que nosotros los creemos o no. Estos métodos son el constructor de la clase y el destructor. El constructor se usa en el momento que creamos un objeto y será llamado por el sistema una vez la estructura se haya creado en memoria para inicializarlo. Del mismo modo, el destructor es el último método que se ejecuta cuando hayamos solicitado la eliminación de un objeto para liberar recursos.

Ejemplo: Define gráficamente una clase Bombero considerando las siguientes propiedades de clase: nombre (Cadena), apellidos (Cadena), edad (Entero), casado (Booleano), especialista (Booleano).

Define un constructor (método con el mismo nombre que la clase) que reciba los parámetros necesarios para la inicialización y los métodos para poder establecer y obtener los valores de los atributos.

Objetos

Las clases son el molde que determina la estructura de “algo” que podemos utilizar. El objeto es la instanciación de una clase, es la creación de ese “algo”. Un objeto se crea a partir de la clase con los métodos y propiedades que ésta determina. Un objeto de una clase es independiente de otro objeto de la clase y el estado de un objeto será diferente al de otro objeto de la misma (se conoce como estado de un objeto al valor de todas sus propiedades).

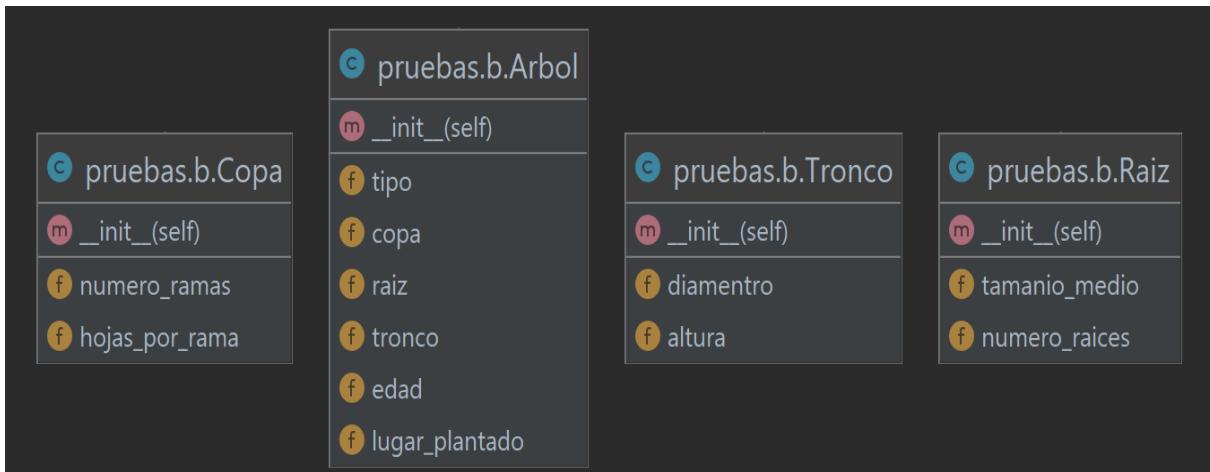


Dos objetos no serán nunca el mismo, incluso aunque su estado sea el mismo, a menos que apunten a la misma área de memoria (es lo que se denomina clonado del objeto).

Los objetos se crean y se destruyen de forma dinámica a través de las herramientas del lenguaje elegido, los utilizamos durante todo el desarrollo de la aplicación para guardar información, realizar acciones sobre ellos, etc.

Las clases se utilizarán cuando queramos acceder a datos o métodos compartidos entre todos los objetos de la clase.

Ejemplo: Crea la siguiente estructura bajo Python. Y haz uso de ella.



Encapsulación y Ocultación

La clase forma un todo, aúna los datos y las operaciones en una única estructura de datos, permite la abstracción, la ocultación, el encapsulamiento, la herencia y el polimorfismo.

La encapsulación es el mecanismo por el que una clase agrupa todos los datos y operaciones dentro de una única estructura. Con la encapsulación un objeto puede ocultar la información que contiene al mundo

exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada, es lo que se denomina visibilidad.

La visibilidad permite que los métodos y propiedades de una clase sean utilizados de forma correcta por otros elementos del programa. La visibilidad puede ser pública (el signo + se utiliza para representar la visibilidad en una definición de clase), privada (-) o protegida (#). En caso de visibilidad pública cualquier parte del programa tendrá acceso, tanto la propia definición de clase como partes externas a la misma. Si la visibilidad es privada, solo la clase tiene acceso a dichos datos u operaciones, ni partes externas ni otras clases podrán utilizarla. Por último, en caso que sea protegida, tanto la propia clase, como cualquiera que herede de ella podrán utilizar el método o propiedad, estando prohibido el uso a otras clases o partes externas.

Visibilidad	Public	Private	Protected
Desde la misma clase	✓	✓	✓
Desde una subclase	✓	✗	✓
Desde otra clase (no subclase)	✓	✗	✗

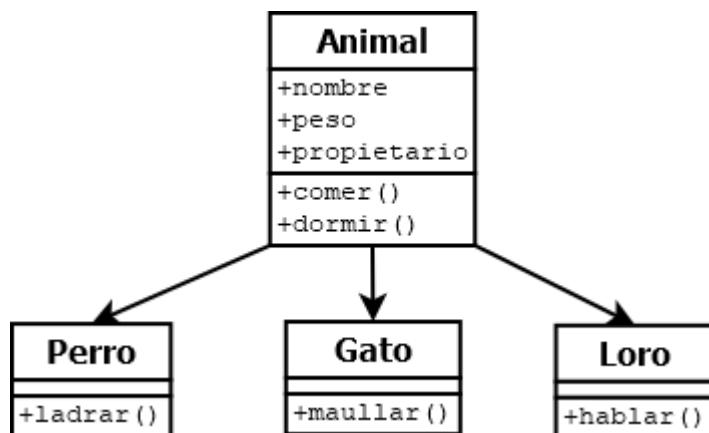
Abstracción

La abstracción es el mecanismo por el que podemos reducir la complejidad de la información, recogiendo solo aquella información que es relevante. Consiste en la generalización conceptual de los atributos y comportamiento de un determinado conjunto de objetos.

Pongamos un ejemplo para comprenderlo mejor. Queremos modelizar una clínica veterinaria en la que vamos a tratar varios tipos de animales: perros, gatos, loros, por simplicidad. Si hacemos un estudio llegamos a la conclusión que necesitamos tres clases diferentes para cada animal.

Perro	Gato	Loro
+nombre +peso +propietario +comer() +dormir() +ladrar()	+nombre +peso +propietario +comer() +dormir() +maullar()	+nombre +peso +propietario +comer() +dormir() +hablar()

Como podemos observar casi todos los métodos y propiedades son similares, por lo que podemos **abstraer** lo importante de las tres clases en otra que llamaremos animal, dejando lo particular en cada una de ellas.



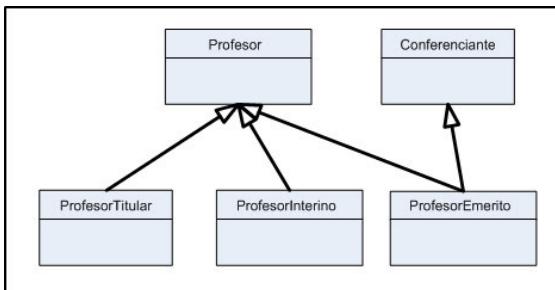
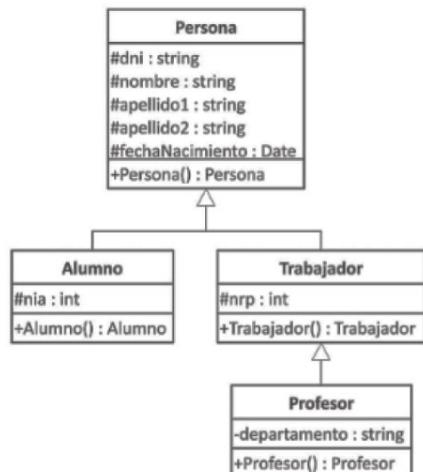
Herencia. Tipos de Herencia

La herencia es la capacidad que tienen las clases de usar las características ya declaradas en otras clases para su propia declaración. La herencia genera una relación de padres e hijos en las clases, una jerarquía, en la que las clases que heredan se sitúan en la parte inferior de la jerarquía, y las clases de las que se heredan en la parte superior.

Como vemos en el ejemplo anterior, las clases perro, gato y loro sería las clases hijas, que heredan las propiedades y métodos de la clase padre animal. Esta jerarquía se puede extender todo lo que se necesite y crear todo lo complejo que el programa requiera.

La herencia se puede presentar en dos vertientes en función de cuántos padres se herede:

- ∞ **Herencia simple.** Cada clase hija solo tendrá un único ancestro o padre. Java tiene este tipo de jerarquía.
- ∞ **Herencia múltiple.** Cada clase hija podrá heredar de tantos ancestros como requiera. Python tiene este tipo de jerarquía.



En el momento que creamos una relación de herencia estamos creando una jerarquía de clases que podremos utilizar para documentar el desarrollo.

Ejemplo: Implementa las estructuras anteriores bajo Python

Ejemplo Definir los siguientes términos:

- | | |
|------------------------------|--------------------------|
| • Abstracción. | • Atributo. |
| • Ciclo de vida. | • Encapsulamiento. |
| • Encuesta. | • Entrevista. |
| • Herencia. | • Método. |
| • Metodología de desarrollo. | • Objeto. |
| • Observación. | • Orientación a Objetos. |
| • Polimorfismo. | • Sistema. |
| • Visibilidad. | |

Ejemplo: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.

José de 30, María de 25 y Pedro de 19 años de edad, todos expertos en alguna rama de la ingeniería, pertenecientes al grupo “IPL”, postulan un proyecto “Inversión de Polos electromagnéticos por medio del plástico”, ya habían postulado otro proyecto el 2001 y otro más el 2014. En este proyecto el grupo se adjudicó 12 millones de pesos para el desarrollo, el monto fue definido por el concurso “Grandes ideas de chilenos para chilenos”, cuyo periodo de postulación duró 6 meses, e inició el 10 de enero del 2016, los otros proyectos fueron presentados en otros fondos y concursos.

Ejemplo: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.

La maratón de Nueva York, es una de las maratones más famosas de la historia, pertenece al selecto grupo de los World Marathon Majors, al cual pertenecen otras importantes maratones mundiales. Inició en 1970, y se ha corrido todos los años, excepto el 2012, cuando fue cancelada por el huracán Sandy. El año 2015 convocó a más de 50.000 atletas. Cubre unos 42.195 kilómetros por toda la ciudad. La carrera inicia en Staten Island, pasa por Brooklyn y Queens, entra a la isla Manhattan y luego avanza por la Primera Avenida, el Bronx, vuelve a Manhattan por la Quinta Avenida, y finaliza en Central Park. El campeón del año 2015 fue Stanley Biwott, de Kenia, con un tiempo de 2:10:34. La campeona fue Mary Keitany, también de Kenia, con un tiempo de 2:24:25.

Ejemplo: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.

El estudio Marvel, presento su Universo Cinemático, con grandes propuestas, para los amantes de los comics, y de las películas de acción, fantasía y ciencia ficción. Esta inició el 2008 con la presentación de Iron Man, esta película fue protagonizada por Robert Downey Jr. Actor norteamericano, en el papel de Tony Stark, la película fue dirigida por Jon Favreau, este año se ha estrenado Captain America: Civil War, protagonizada por Chris Evans, en el papel del Capitán América, y un gran reparto de otros actores y personajes ya presentados en otras películas, esta fue dirigida por los hermanos Russo. Las películas de Marvel se agrupan en fases de entrega, por ejemplo, esta última corresponden a la fase 3, la primera fase inició en el 2008, y se espera que la serie de películas finalice en el 2019 ó 2020.

*Soluciones*Actividad N°1: Definir los siguientes términos:

- **Abstracción:** es la acción de aislar a un elemento desde su contexto para realizar un análisis dirigido y delimitado. Interesando mucho más el ¿qué hace? en vez de ¿Cómo lo hace?
- **Atributo:** cualidad, adjetivo que define o declara una característica de un objeto, por ejemplo, para una clase alumno, el nombre y la edad, serían atributos.
- **Ciclo de vida:** se les llama así a las metodologías para desarrollo de software, que inician con la toma de requerimientos para el desarrollo, luego diseñan y construyen, para luego probar y realizar procesos de mantenimiento para preservar un sistema informático a través del tiempo, hasta el momento en que será reemplazado.
- **Encapsulamiento:** es la técnica utilizada para que las clases y objetos sean utilizados a través de los métodos que ellas estimen, evitando que el usuario conozca la estructura y la forma de procesar que tiene, esto asegura que la clase quede protegida por modificaciones externas.
- **Encuesta:** técnica de captura de requerimientos, que se basa en un conjunto de preguntas de rápida resolución, mayormente cerradas o de respuestas pre-escritas. Idealmente utilizada cuando el volumen de las personas a evaluar es alto, o se pretende capturar impresiones, pero no sugerencias y nuevas ideas. La mayoría de las encuestas apuntan a la mejora de la percepción del producto, interfaces, pantallas, documentos y dispositivos.
- **Entrevista:** técnica de captura de requerimientos, que se basa en un conjunto de preguntas de conversación y respuesta amplia, idealmente se utilizan para cuando el volumen de entrevistados es pequeño, o es un conjunto selecto como usuarios finales, focus group, o stakeholders, donde sus impresiones y sugerencias son significativas para determinar los alcances del sistema.
- **Herencia:** capacidad de una clase para adaptar los atributos y métodos de clases superiores, en el caso de Java de una clase superior (no permite herencia múltiple), la herencia presenta las clases en una estructura jerárquica.
- **Método:** es el comportamiento programado de una clase, los atributos de una clase son tratados a través de sus métodos. Los métodos son las acciones que pueden realizar las clases.
- **Metodología de desarrollo:** es el conjunto ordenado de las fases para desarrollar una solución de software. Incluyen el Análisis, el Diseño, la Construcción, las Pruebas y el Mantenimiento.
- **Objeto:** cualquier elemento del mundo real, que nos entregue información a través de sus características propias (atributos) o de lo que hace (métodos), son capaces de comunicarse e interactuar con otros objetos, logrando una simulación de la vida.
- **Observación:** técnica para captar requerimientos en un ambiente laboral, que consiste en ser parte del trabajo para conocer a fondo el actuar de las personas que laboran, para determinar los aspectos automatizables y que, a la vez, provoquen menor impacto en la organización.
- **Orientación a Objetos:** la Programación Orientada a Objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación. Con la POO tenemos que aprender a pensar las cosas de una manera distinta, para escribir nuestros programas en términos de objetos, propiedades, métodos y otras cosas que veremos rápidamente para aclarar conceptos y dar una pequeña base que permita soltarnos un poco con este tipo de programación.
- **Polimorfismo:** es la capacidad de los objetos de implementar sus métodos y que estos reaccionen de manera distinta según la naturaleza de los objetos creados o de los estímulos que esta reciba.
- **Sistema:** un conjunto de elementos que interactúan entre sí para lograr un objetivo común.
- **Visibilidad:** la capacidad de las clases, atributos o métodos, para ser invocadas y utilizadas por otros.

Actividad N°2: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.

Objetos:

- José, María, Pedro.
- IPL
- “Inversión de Polos electromagnéticos por medio del plástico”
- “Grandes ideas de chilenos para chilenos”

Clases y Atributos:

- Postulante (clase)
 - nombre Postulante
 - edad Postulante
 - especialidad Postulante
- Grupo (clase)
 - nombreGrupo
- Proyectos (clase)
 - nombreProyecto
 - añoPresentacionProyecto
 - situacionProyecto
- Concurso (clase)
 - nombreConcurso
 - fondoConcurso
 - duracionPostulacionConcurso
 - fechalinicioConcurso

Actividad N°3: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.

Objetos:

- Maratón de Nueva York.
- World Marathon Majors.
- Huracán Sandy.
- Staten Island, Brooklyn, Queens, Isla Manhattan, Primera Avenida, Quinta Avenida, Central Park, El Bronx.
- Stanley Biwott, Mary Keitany.
- Kenia.

Clases y atributos

- Maratón (clase)
 - nombreMaraton
 - distanciaMaraton
 - cantidadParticipantesMaraton
 - añoInicioMaraton
 - inicioMaraton
 - terminoMaraton

- Agrupación (clase)
 - nombreAgrupacion
- Evento (clase)
 - nombreEvento
 - tipoEvento
 - añoEvento
- Localidad (clase)
 - nombreLocalidad
 - tipoLocalidad
- Campeón.
 - nombreCampeon
 - sexoCampeon
 - tiempoCampeon
 - nacionalidadCampeon
- País (clase)
 - nombrePais
 - banderaPais

Actividad N°4: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.

Objetos:

- Iron Man, Captain America: Civil War.
- Robert Downey Jr., Chris Evans
- Jon Favreau, Hermanos Russo
- Fase 3

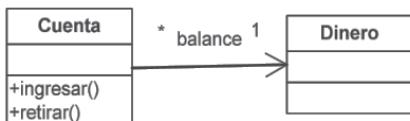
Clases y Atributos:

- Película (clase)
 - nombrePelicula
 - añoPelicula
- Actor (clase)
 - nombreActor
 - nacionalidadActor
 - papelActor
- Director (clase)
 - nombreDirector
 - nacionalidadDirector
- Fase 3 (clase)
 - nombreFase3
 - añoFase3

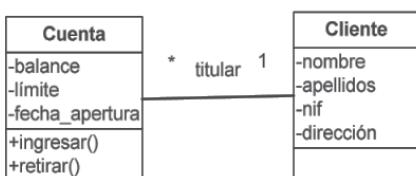
Relaciones entre las clases

Las clases, además de las jerarquías crean otro tipo de relaciones entre ellas en función del problema. Estas relaciones se reflejarán como propiedades o métodos dentro de las clases para poder realizar la relación. Las relaciones se describen adecuadamente a través UML y su diagrama de clases.

- ∞ **Asociación.** Diremos que dos (o más) clases tienen una relación de asociación cuando una de ellas tenga que requerir o utilizar alguno de los servicios (es decir, acceder a alguna de las propiedades o métodos) de las otras.



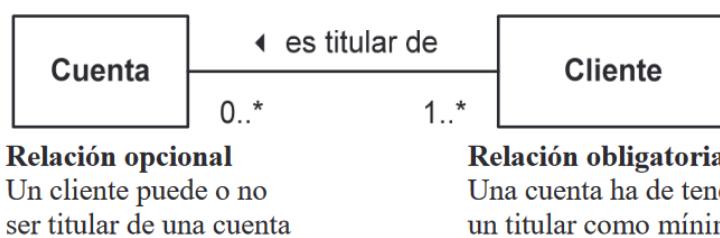
Asociación unidireccional



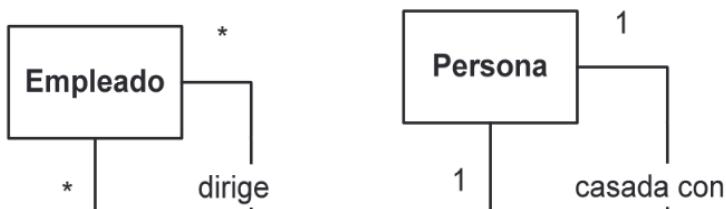
Asociación bidireccional

Multiplicidad	Significado
1	Uno y sólo uno
0 .. 1	Cero o uno
N .. M	Desde N hasta M
*	Cero o varios
0 .. *	Cero o varios
1 .. *	Uno o varios (al menos uno)

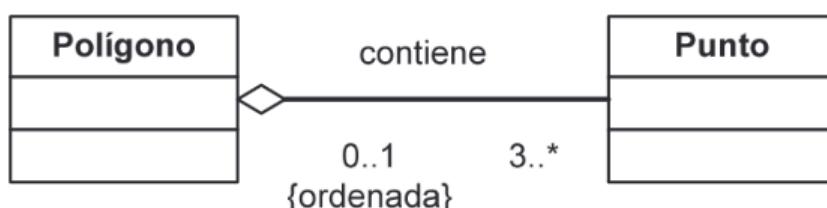
La asociación presenta además una multiplicidad en la que indica cómo se relacionan las clases de la asociación. Podemos ver en la tabla anterior el significado de las cardinalidades.



Se pueden dar relaciones de asociación entre la misma clase como se ve a continuación.

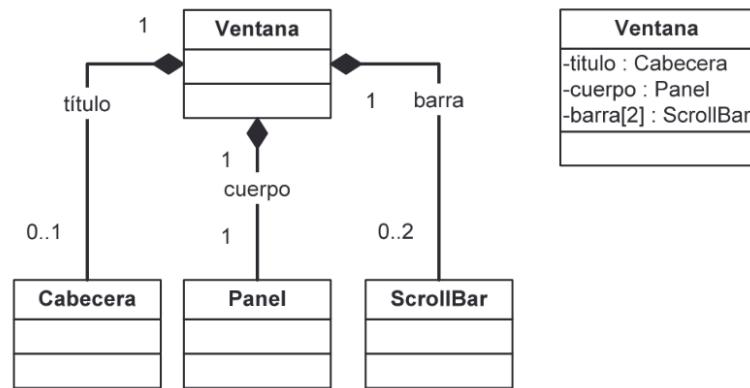


- ∞ **Agregación.** La agregación es una relación entre clases donde una clase forma parte de otra, pero no de forma exclusiva, pudiendo formar parte de otras relaciones de agregación.

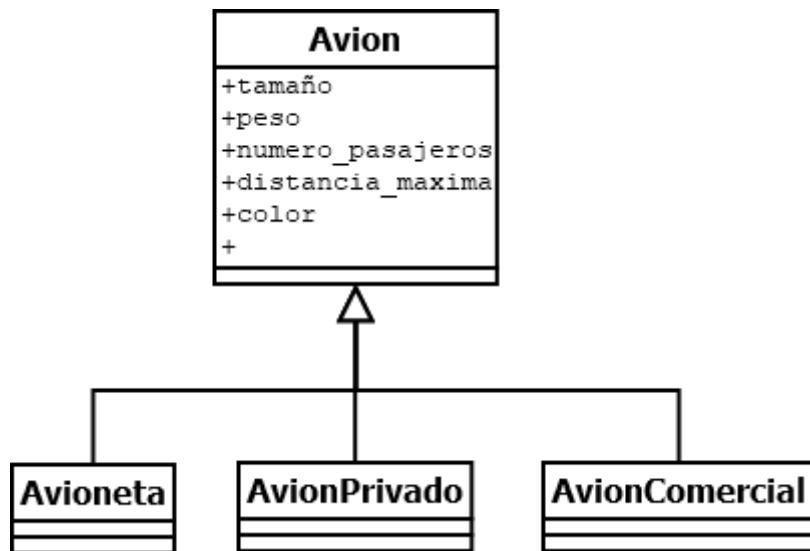


En este ejemplo punto forma parte de polígono, pero nada le impide formar parte también de otra clase, por ejemplo, círculo.

- ∞ **Composición.** La composición es una relación de agregación en la que las partes constituyentes forman parte de forma exclusiva de la relación, no pudiendo formar parte de otras relaciones de agregación o composición. Además, cada parte no tiene nada en común con las otras clases constituyentes, son disjuntas y solo existen asociadas a la relación.

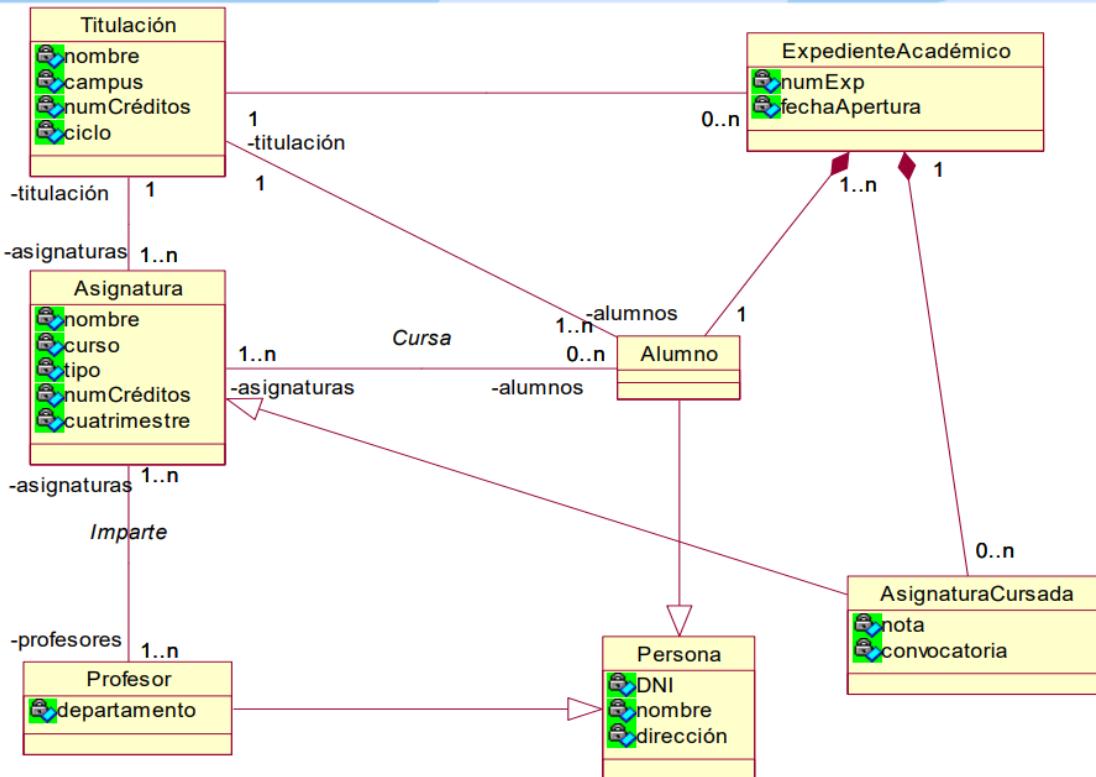


Ejemplo: Considera que queremos representar mediante un programa los aviones que operan en un aeropuerto. Crea un esquema análogo para aviones. Define cuáles podrían ser las clases y cuáles podrían ser algunos objetos de una clase.



Ejemplo: Se desea desarrollar una aplicación para la gestión académica de una universidad. Especificaciones: El sistema debe ser capaz de gestionar todos los expedientes académicos de los alumnos dando la posibilidad de realizar las operaciones típicas de altas, bajas, modificaciones y consultas de los datos del mismo. Debemos dar un número de expediente único en el sistema y una fecha de apertura. La información mínima que se debe guardar de un alumno son el DNI, nombre, dirección, la titulación en la que está matriculado, así como las asignaturas que está cursando actualmente. También debemos almacenar su historial académico, donde deben aparecer todas las asignaturas cursadas y sus respectivas notas y convocatoria.

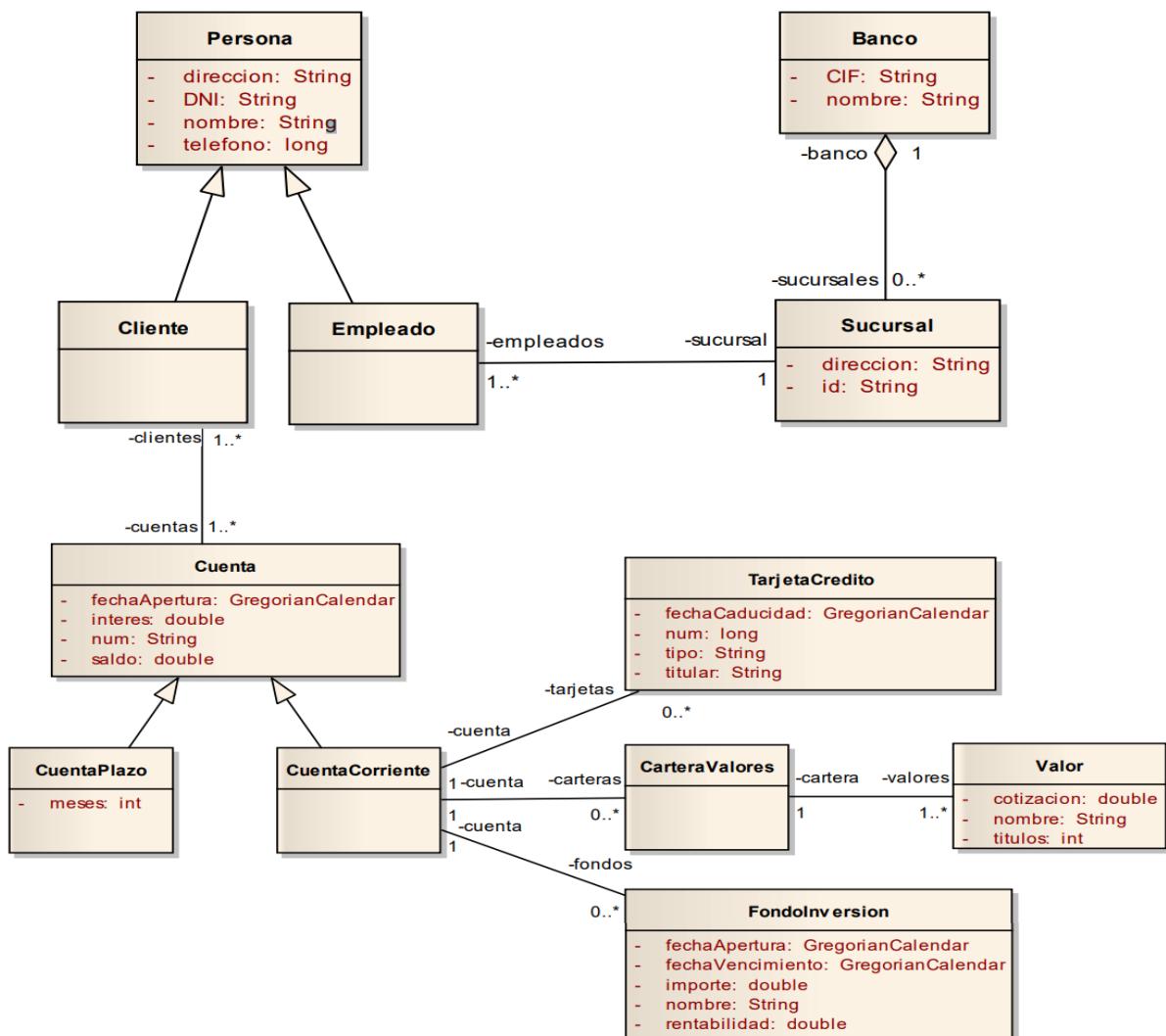
Se debe realizar la gestión de las distintas titulaciones que existen en la universidad teniendo en cuenta que una titulación sólo se da en un campus determinado y los datos que podemos consultar son el nombre, el número de créditos, si es de primer o segundo ciclo, etc. Se tienen que gestionar las asignaturas que se imparten en una titulación, teniendo en cuenta que una asignatura solo se puede dar en un único curso. Algunos de los datos que se pueden consultar de una asignatura son: el nombre, número de créditos, cuatrimestre en el que se imparte y su tipo (obligatoria, troncal, optativa). Se debe guardar la información de los profesores que imparten las distintas asignaturas de la titulación. Se debe almacenar como mínimo su DNI, nombre, dirección y departamento al que pertenece. También se podrá consultar las distintas asignaturas que imparte.



Ejercicio: Implementa las estructuras anterior y siguiente bajo Python

Ejemplo: Se desea desarrollar una aplicación de gestión bancaria. Especificaciones: El sistema debe ser capaz de gestionar una serie de productos asociados a los clientes del banco. Los productos que gestiona el banco son: cuentas bancarias, fondos de inversión y carteras de valores. Las cuentas deben tener: número de cuenta, fecha de apertura, saldo y tipo de interés y los datos de sus clientes. El banco tiene dos tipos de cuentas: corrientes y a plazo. Las cuentas corrientes pueden tener tarjetas de crédito asociadas. Solo éstas cuentas pueden tener el resto de productos asociados. Las cuentas a plazo deben tener el número de meses que estará abierta

De los clientes y los empleados se debe almacenar la siguiente información: DNI, nombre, dirección y teléfono. De los empleados necesitamos saber en qué sucursal trabajan. Cada sucursal tendrá un identificador y una dirección. Los fondos de inversión deben tener un nombre, importe, rentabilidad y la fecha de apertura y vencimiento. Las carteras de valores están compuestas por los valores asociados, almacenando el nombre del valor, el número de títulos y el precio de cotización. Las tarjetas de crédito deben almacenar el tipo (Visa, MasterCard, etc.), el número, el titular y la fecha de caducidad.



```
class Persona:  
    def __init__(self, dni, direccion=None,  
                 nombre=None, telefono=None):  
        self.dni = dni  
        self.direccion = direccion  
        self.nombre = nombre  
        self.telefono = telefono  
  
    def __str__(self):  
        return f"Dº {self.nombre} ({self.direccion})"  
  
class Cliente(Persona):  
    def __init__(self, **kwargs):  
        super().__init__(**kwargs)  
        # La relación original es N a N, pero no lo hemos visto  
        # todavía, un cliente N cuentas no al revés  
        self.cuentas = []  
  
    def add_cuenta(self, cuenta):  
        if isinstance(cuenta, Cuenta):  
            self.cuentas.append(cuenta)  
  
    def get_cuenta(self, identificador):  
        for index, cuenta in enumerate(self.cuentas):  
            if cuenta.num == identificador:  
                return self.cuentas[index]  
        return None  
  
    def print_cuentas(self):  
        str_ret = ""  
        for cuenta in self.cuentas:  
            str_ret += str(cuenta) + "\n"  
        return str_ret  
  
class Empleado(Persona):  
    def __init__(self, **kwargs):  
        super().__init__(**kwargs)  
  
class Sucursal:  
    def __init__(self, direccion, identificador):  
        self.direccion = direccion  
        self.id = identificador  
        self.empleados = []  
  
    def __str__(self):  
        return f"Sucursal: {self.direccion} ({self.id})"  
  
    def add_empleado(self, empleado):  
        if type(empleado) == Empleado:  
            self.empleados.append(empleado)  
  
    def get_empleado(self, identificador):  
        for index, empleado in enumerate(self.empleados):  
            if empleado.dni == identificador:
```

```
        return self.empleados[index]
    return None

    def print_empleados(self):
        str_ret = ""
        for empleado in self.empleados:
            str_ret += str(empleado) + "\n"
        return str_ret

class Banco:
    def __init__(self, cif=None, nombre=None):
        self.cif = cif
        self.nombre = nombre
        self.sucursales = [] # relación

    def __str__(self):
        str_ret = ""
        for sucursal in self.sucursales:
            str_ret += "|---" + str(sucursal) + "\n"

        return f"Banco:{self.nombre}\n" + str_ret

    def add_sucursal(self, sucursal):
        if type(sucursal) == Sucursal:
            self.sucursales.append(sucursal)

    def get_sucursal(self, identificador):
        for index, sucursal in enumerate(self.sucursales):
            if sucursal.id == identificador:
                return self.sucursales[index]
        return None

class Cuenta:
    def __init__(self, num=0):
        self.fecha_apertura = ""
        self.interes = 0
        self.num = num
        self.saldo = 0

    def __str__(self):
        return f"Cuenta: {self.num} ({self.saldo})"

    def add_saldo(self, saldo):
        if saldo > 0:
            self.saldo += saldo

    def del_saldo(self, saldo):
        if saldo > 0:
            self.saldo -= saldo

class CuentaPlazo(Cuenta):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.meses = 0
```

```
def __str__(self):
    return f"CuentaPlazo: {self.num} ({self.saldo})"

class CuentaCorriente(Cuenta):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.tarjetas = []
        self.fondos = []
        self.carteras = []

    def __str__(self):
        return f"CuentaCorriente: {self.num} ({self.saldo})"

    def add_tarjeta(self, tarjeta):
        if type(tarjeta) == TarjetaCredito:
            self.tarjetas.append(tarjeta)

    def get_tarjeta(self, identificador):
        for index, tarjeta in enumerate(self.tarjetas):
            if tarjeta.num == identificador:
                return self.tarjetas[index]
        return None

    def print_tarjetas(self):
        str_ret = ""
        for tarjeta in self.tarjetas:
            str_ret += str(tarjeta) + "\n"
        return str_ret

    def add_fondo(self, fondo):
        if type(fondo) == Fondo:
            self.fondos.append(fondo)

    def get_fondo(self, identificador):
        for index, fondo in enumerate(self.fondos):
            if fondo.nombre == identificador:
                return self.fondos[index]
        return None

    def print_fondos(self):
        str_ret = ""
        for fondo in self.fondos:
            str_ret += str(fondo) + "\n"
        return str_ret

    def add_cartera(self, cartera):
        if type(cartera) == CarteraValores:
            self.carteras.append(cartera)

    def get_cartera(self, identificador):
        for index, cartera in enumerate(self.carteras):
            if cartera.nombre == identificador:
                return self.carteras[index]
        return None

    def print_carteras(self):
```

```
str_ret = ""
for cartera in self.carteras:
    str_ret += str(cartera) + "\n"
return str_ret

class TarjetaCredito:
    def __init__(self, num):
        self.fecha_caducidad = None
        self.num = num
        self.tipo = ""
        self.titular = ""

    def __str__(self):
        return f"Tarjeta Número {self.num} ({self.titular})"

class Fondo:
    def __init__(self, nombre):
        self.nombre = nombre
        self.fecha_apertura = None
        self.fecha_vencimiento = None
        self.importe = 0
        self.rentabilidad = 0

    def __str__(self):
        return f"Fondo {self.nombre} ({self.importe})"

class Valor:
    def __init__(self, nombre):
        self.nombre = nombre
        self.cotizacion = 0
        self.titulos = 0

    def __str__(self):
        return f"Valor {self.nombre} ({self.titulos})"

class CarteraValores:
    def __init__(self, num):
        self.num = num
        self.valores = []

    def __str__(self):
        return f"Cartera de Valores: {self.num}"

    def add_valor(self, valor):
        if type(valor) == Valor:
            self.valores.append(valor)

    def get_valor(self, identificador):
        for index, valor in enumerate(self.valores):
            if valor.nombre == identificador:
                return self.valores[index]
        return None

    def print_valores(self):
```

```
str_ret = ""
for valor in self.valores:
    str_ret += str(valor) + "\n"
return str_ret

if __name__ == "__main__":
    cliente = Cliente(direccion="Hola", nombre="JP", dni=99)
    print(cliente)

    # Pruebas de banco y sucursales
    banco = Banco("B123456789", "Banco UNO")
    for value in range(5):
        # Creamos 5 sucursales, su id será el número
        sucursal_aux = Sucursal("Direccion " +
                                str(value), value)
        banco.add_sucursal(sucursal_aux)

    # Añadimos empleados a una sucursal
    print(banco)
    SUCURSAL_A_MODIFICAR = 3
    print(banco.get_sucursal(SUCURSAL_A_MODIFICAR))
    banco.get_sucursal(SUCURSAL_A_MODIFICAR).add_empleado(
        Empleado(nombre="Empleado 1", dni=1))
    banco.get_sucursal(SUCURSAL_A_MODIFICAR).add_empleado(
        Empleado(nombre="Empleado 2", dni=2))
    print(banco.get_sucursal(
        SUCURSAL_A_MODIFICAR).print_empleados())
    print(banco.get_sucursal(
        SUCURSAL_A_MODIFICAR).get_empleado(2))
    print()

    # Creamos cuentas en el cliente
    cuenta_aux = CuentaPlazo(num=1)
    cuenta_aux.add_saldo(100)
    cliente.add_cuenta(cuenta_aux)
    cuenta_aux = CuentaCorriente(num=2)
    cuenta_aux.add_saldo(1000)
    cliente.add_cuenta(cuenta_aux)
    print(cliente.print_cuentas())
    print(cuenta_aux)
    cuenta_aux.add_tarjeta(TarjetaCredito(0))
    cuenta_aux.add_tarjeta(TarjetaCredito(1))
    cuenta_aux.add_fondo(Fondo("Nombre 1"))
    cuenta_aux.add_fondo(Fondo("nombre 2"))
    print(cuenta_aux.print_tarjetas())
    print(cuenta_aux.print_fondos())
    cuenta_aux.add_cartera(CarteraValores("Una cartera"))
    cv = CarteraValores("Cartera dos")
    cuenta_aux.add_cartera(cv)
    print(cuenta_aux.print_carteras())
    cv.add_valor(Valor("val 1"))
    cv.add_valor(Valor("val 2"))
    print(cv.print_valores())
```

Sobrecarga

Una de las restricciones de los primeros lenguajes estructurados (fuertemente tipados) es que las funciones o procedimientos solo podían esperar el argumento de un tipo específico. De tal manera que, si una función esperaba como primer parámetro un entero, si se le pasaba cualquier otro tipo de parámetro generaba un error de ejecución.

Para solucionar este problema se definió en la POO la sobrecarga, o la creación de métodos que se nombran igual, pero difieren en la firma (número y tipos de parámetros), de tal manera que se determinará en tiempo de ejecución (enlace dinámico o unión tardía) al método a llamar en función de los tipos de los parámetros.

```

1 public class Calculos{
2
3     public int Suma(int a, int b){
4         return a + b;
5     }
6
7     public double Suma(double a, double b){
8         return a + b;
9     }
10
11    public long Suma(long a, long b){
12        return a + b;
13    }
14
15 }
```

Los lenguajes dinámicamente tipados adolecen de la sobrecarga al admitir sus variables y parámetros cualquier tipo.

Polimorfismo y Enlace Dinámico

El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, ...)

Dicho de otra manera, la posibilidad que un objeto de una clase pueda ser utilizado como si fuera un objeto de la clase padre de la misma. El sistema de POO hace una “conversión” entre el objeto hijo y el objeto padre y se pueden utilizar de manera indistinta.

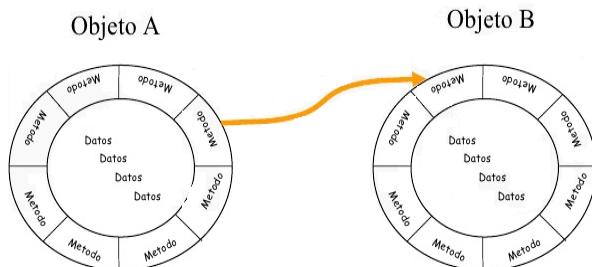


El único requerimiento para usar el polimorfismo es que hagamos uso de los métodos heredados exclusivamente para tratar los objetos hijos.

En este ejemplo, cualquier método que espere como parámetro una clase vehículo, podrá aceptar también cualquiera de sus hijas (coche, moto y bus) siempre que solo haga uso del interfaz (conjunto de métodos definidos en una clase) de la clase padre.

Mensajes

En la POO se denomina mensaje al uso que un objeto hace de un método de un segundo objeto facilitándole los parámetros esperados, y el primer objeto recoge los resultados. El símil con la programación estructurada es la llamada a una función o procedimiento.



El mecanismo de mensajes es la base de la comunicación de datos entre los objetos y de realización de tareas. Para que una aplicación funcione los objetos se tienen que intercambiar entre sí mensajes, haciendo que los objetos reaccionen a esos mensajes y efectúen las funciones para las que han sido programados.

Lenguajes

Un lenguaje es orientado a objetos si cumple entre otras características con lo siguiente:

- ∞ Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombre y un estado local oculto.
- ∞ Los objetos tienen un tipo asociado (la clase).
- ∞ Los tipos (clases) pueden heredar atributos de los supertipos (superclases)

La relación de lenguajes orientados a objetos es inmensa, podemos destacar C++, Java y Python como algunos exponentes de los mismos, pero en la siguiente dirección encontraremos una lista más amplia.

https://en.wikipedia.org/wiki/List_of_object-oriented_programming_languages

Ejercicios

1. Representa mediante un diagrama de clases la siguiente especificación relacionada con los alquileres de cámaras en una tienda de fotografía.
 - ∞ La tienda alquila cámaras fotográficas analógicas.
 - ∞ Las cámaras se caracterizan por su marca, modelo y soporte flash (si, no).
 - ∞ Cada cámara es compatible con uno o más tipos de películas.
 - ∞ Las películas se caracterizan por su marca, nombre, sensibilidad ISO (50, 100, 200, 400, 800, 1600) y formato (35 mm, 110 mm, 120 mm).
 - ∞ Para cada marca con la que trabaja la tienda se conoce la dirección del servicio de reparación más cercano.
 - ∞ La tienda dispone de varios ítems de cada modelo de cámara.
 - ∞ Cada ítem tiene una pegatina con una referencia, y puede estar en la tienda, alquilado, con retraso o en reparación.
 - ∞ Los clientes pueden tener un máximo de 1 cámara en alquiler.
2. Representa mediante un diagrama de clases la siguiente especificación sobre las personas que participan en una película
 - ∞ De cada película se almacena el título, la sinopsis, el año, el género al que pertenece (drama, comedia, acción, terror, romance, aventura, scifi) y el país.
 - ∞ Sobre las personas que participan en la película es necesario conocer el nombre, los apellidos, la fecha de nacimiento y la nacionalidad.
 - ∞ Una persona puede participar en una película como actor, director, productor o guionista.
 - ∞ Una película tiene al menos un director y un guionista.
 - ∞ Una persona se considera actor si ha actuado al menos en una película.
 - ∞ Una película puede tener asociados varios trailers que son editados por una o más personas. No puede existir el trailer de una película hasta que existe la película.
3. Representa mediante un diagrama de clases la siguiente especificación relacionada con un sistema para gestionar series:
 - ∞ Las series se caracterizan por su título, año de inicio, sinopsis y género al que pertenece (acción, aventura, animación, comedia, documental, drama, horror, musical, romance, ciencia ficción)
 - ∞ Las series se organizan en temporadas ordenadas que tienen una fecha de producción y una fecha de estreno de televisión a nivel mundial.
 - ∞ Cada temporada está a su vez formada por capítulos ordenados que tienen un título, una duración y una sinopsis.
 - ∞ Los usuarios se caracterizan por su nombre y apellidos, dirección de correo electrónico y fecha de nacimiento.
 - ∞ Si un usuario ha visto algún capítulo de una temporada el sistema la marca como empezada, si ha visto todos los capítulos de la temporada la marca como vista y un usuario en cualquier momento puede dar una temporada como cancelada, es decir, indicar que la deja de seguir.
 - ∞ Además, el sistema registra qué capítulos ha visto el usuario.
4. Representa mediante un diagrama de clases la siguiente especificación sobre una empresa:
 - ∞ Una aplicación necesita almacenar información sobre empresas, sus empleados y sus clientes.
 - ∞ Ambos se caracterizan por su nombre y edad.
 - ∞ Los empleados tienen un sueldo bruto, los empleados que son directivos tienen una categoría, así como un conjunto de empleados subordinados.
 - ∞ De los clientes además se necesita conocer su teléfono de contacto.
 - ∞ La aplicación necesita mostrar los datos de empleados y clientes.
5. Construir un diagrama de clases para un hospital con un conjunto de pacientes y un conjunto de empleados sanitarios (médicos y enfermeros) siguiendo las siguientes especificaciones:
 - ∞ Un paciente es atendido por uno o más médicos y es asistido por un grupo de enfermeros
 - ∞ Cada paciente se describe por su número de historia clínica, su nombre y dirección

- ∞ En la base de datos se mantiene información del personal sanitario referente a su número de empleado, nombre y tipo. Para los médicos hay que indicar además su especialidad
- ∞ Para cada paciente se mantiene un registro de los análisis realizados identificados por un número de referencia, además se indica el tipo de análisis, la fecha en la que se realizó, el médico que solicitó el análisis, él o los enfermeros que realizaron la prueba y los resultados que se obtuvieron en la misma.

Ejercicio (Repasso)

Introducción

Implemente la funcionalidad básica de la tienda: gestionar el almacén, gestionar los clientes y realizar ventas.

Descripción del proyecto

La práctica a desarrollar consistirá en diseñar e implementar completamente el sistema de gestión de una tienda virtual. Se deberá realizar un diagrama de clases adaptado a la funcionalidad requerida en esta práctica e implementar la funcionalidad posteriormente. Se valorará un diseño que aplique adecuadamente los principios de orientación a objetos (jerarquías de clase, modularidad, etc.)

El almacenamiento se realizará en Arrays.

El sistema a implementar consistirá en una aplicación que simulará el comportamiento de una tienda online que venderá distintos tipos de artículos (en el ejemplo a implementar libros, discos y películas)

Existirán dos posibles perfiles de usuario:

- Un perfil administrador que se encargará de gestionar el almacén.
- Y un perfil cliente que será el que realice las compras en la tienda.

Las posibles acciones para el administrador serán, al menos, las siguientes:

- Mostrar el inventario completo de la tienda/almacén, ordenados por id de artículo
- Mostrar el inventario de un tipo de artículo concreto ordenado por descripción de artículo
- Realizar búsquedas de artículos
- Aumentar el stock de un artículo
- Mostrar el listado de clientes de la tienda

Por su parte, las acciones para los clientes serán, al menos:

- Registrarse como usuario en la tienda.
- Aumentar su saldo para compras.
- Realizar búsquedas de artículos concretos (queda a disposición del alumno el definir los mecanismos de búsqueda que proporcionará la aplicación).
- Añadir y eliminar artículos de su carrito de la compra.
- Comprar los artículos que ha añadido al carrito (si no hay saldo suficiente a la hora de realizar la compra ésta no será posible)
- Visualizar su histórico de compras entre dos fechas dadas

En la tienda de prueba que se implementará se gestionarán al menos tres posibles tipos de artículos: libros, discos y películas. Cada uno de ellos tendrá los siguientes atributos:

- Libro: id interno, título, autor, editorial.
- Disco: id interno, título, intérprete, año de publicación.
- Película: id interno, título, género, director.

U. T. 5 POO (I).



Clases

Una clase describe un grupo de objetos que contienen una información similar (atributos) y un comportamiento común (métodos).

Las definiciones comunes (nombre de la clase, los nombres de los atributos, y los métodos) se almacenan una única vez en cada clase, independientemente de cuántos objetos de esa clase estén presentes en el sistema.

Una clase es como un molde, a partir de ella se pueden crear objetos. Es decir, antes de poder utilizar un objeto se debe definir la clase a la que pertenece.

En notación UML la estructura de una clase se define así:

Nombre de la clase
Atributos
Métodos

Ejemplo: Definir una clase persona

La clase Persona contiene dos atributos (variables) para almacenar datos sobre una persona (nombre y edad) además de varios métodos (es mayor de edad, imprimir el nombre) que hacen cosas con dichos datos

Persona
+ nombre
+ edad

+ es_mayor_de_edad
+ imprimir_nombre

Los atributos y métodos de una clase se llaman **miembros de una clase**. Los atributos de una clase se llaman **variables de instancia** porque cada instancia de la clase (es decir, cada objeto de la clase), contiene sus propias variables atributo. Por lo tanto, los datos de cada objeto son individuales e independiente de los demás objetos, las variables que se comparten entre todos los objetos se llaman **variables de clase**.

Declaración de una clase

Python utiliza una programación muy fácil, limitando el número de estructuras y características posibles en el lenguaje, lo que hace que el inicio y las características de la POO también sean sencillas.

Python solo implementa una visibilidad pública, todos los métodos y atributos son públicos

Para empezar, vamos a ver cómo se definiría y usaría una clase muy simple, la clase Persona. Empezaremos definiendo la clase y a continuación un objeto de la misma, haremos uso del constructor (`__init__`) con dos subrayados delante y detrás, para inicializarla, y del método imprimir.

Definamos La clase persona:

```

class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre # definimos las propiedades
        self.edad = edad

    #definimos los métodos de la clase
    def es_mayor_de_edad(self):
        return self.edad >= 18

    def imprimir(self):
        return self.nombre

# creación y uso de objetos
p = Persona("Juan", 23)
print(p.imprimir())
print(f"{p.nombre} tiene {p.edad} años")
print(p)
q = Persona("Elena", 63)
print(f"{q.nombre} tiene {q.edad} años")
print(q)

```

La definición de una clase siempre comienza con la palabra clave **class** seguida del nombre de la clase y terminada con los dos puntos(:), en caso que se dé la herencia la sintaxis cambia un poco, pero se verá en el siguiente capítulo. Los métodos se definen con la palabra clave **def** seguido del nombre y los parámetros entre paréntesis. Hay que destacar que el primer parámetro de la definición será **self** en el que el sistema nos pasará una referencia al objeto actual (este mecanismo difiere mucho de otros lenguajes, que tienen una o varias variables internas predefinidas que apuntan a ese objeto: this, that, etc.). Para definir propiedades tenemos que inicializarlas en cualquier método, pero el procedimiento que vamos a seguir es usar el método “constructor o inicializador” (**__init__**) para tal fin. Como vemos creamos dos propiedades (nombre y edad) y las inicializamos.

Como todo Python, la indentación entre bloques es obligatoria

Hay que tener en cuenta que la clase Persona nos servirá para crear tantos objetos **Persona** como necesitemos, cada uno con su nombre y edad. Los métodos nos permitirán manipular los datos de cada objeto. En este caso creamos dos objetos (p, q) y los inicializamos en el momento de la creación (Persona(...)). También hacemos uso de sus propiedades y métodos mediante la notación punto (p.nombre, p.metodo())

Nombres de la clase

El nombrado de las clases, métodos y propiedades es muy importante en cualquier definición y deberemos usar las normas del lenguaje (PEP) o las del proyecto en concreto. Las normas básicas que seguiremos en nuestros proyectos son:

- ∞ Las clases se nombran en formato primera letra de palabra en mayúscula: Persona, AulasClase, etc. Usando nombres para la definición.
- ∞ Los métodos se nombran en minúsculas completamente, separando cada palabra con un subrayado, usando verbos como primera palabra: imprimir_nombre, grabar_datos, etc.
- ∞ Las propiedades o atributos se escriben como los métodos, usando solo nombres: numero_lineas, nombre_persona, etc.
- ∞ Se implementarán primero todos los métodos mágicos (comienzan por dos subrayados) y a continuación los nuestros.
- ∞ Se implementarán primero un método utilizado que el método que lo usa.
- ∞ Los métodos relacionados se implementarán cerca unos de otros.

Estructura y miembros de una clase

Creación y utilización de atributos.

Los atributos o propiedades en las clases son los datos de la misma. Estos datos se pueden definir en cualquier método de la clase con solo inicializarlos, pero es mucho más claro utilizar el constructor para tal fin.

```
def __init__(self, nombre, edad):  
    self.nombre = nombre # definimos las propiedades  
    self.edad = edad
```

Todos los atributos o propiedades en Python son públicos con lo que no es necesario especificar la visibilidad del mismo. En algunos entornos de POO critican esta característica al “mermar” la POO de alguna manera. No es verdad, ya que todas las opciones en las que surgen los atributos privados o protegidos se pueden implementar de otra manera eficiente bajo Python.

Independientemente que todos los atributos son públicos, podemos indicar a otros programadores que no deben hacer uso de un atributo anteponiendo un subrayado (_) al nombre. Esto **no impedirá nada**, no se controlará nada, es simplemente una sugerencia al programador.

```
...  
def __init__(self, nombre, edad):  
    self._nombre = nombre # definimos un atributo privado  
  
print(p_nombre)
```

Si aun así tenemos la imperiosa necesidad de impedir el acceso a un atributo, se puede anteponer al nombre dos subrayados (_) de tal manera que “impedirá” de forma externa al objeto el acceso (*Nota:* En verdad no se impide, se accede a una característica de renombrado en la que internamente mantiene el nombre original y externamente tiene otro nombre, si conocemos las reglas de nombrado podremos acceder desde fuera también, pero allá cada uno con no seguir lo que se le sugiere).

Para acceder a un atributo, utilizaremos la notación punto (.), utilizaremos el objeto (o la clase si es un atributo de clase) seguido de un punto y del nombre del atributo. Al ser una variable, podrá estar en la parte izquierda de una asignación para darle un valor y en la parte derecha para acceder.

```
p.nombre = "María"  
print(p_nombre)  
cadena = f"{p.nombre} tiene {p.edad} años"
```

Creación y utilización de métodos.

Un método en POO, función o procedimiento en programación estructurada, es una sección de un programa que calcula un valor de manera independiente al resto del programa o realiza una acción sobre los datos, que tiene tres componentes importantes:

- ∞ Los parámetros, que son los valores que recibe como entrada.
- ∞ El código de la función, que son las operaciones que hace.
- ∞ El resultado (o valor de retorno), que es el valor final que entrega.

En esencia, una función o método (formalmente hablando, un método es una operación de la clase sobre los datos de la misma) es un mini programa y sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.

Para Python la única diferencia entre métodos y funciones es que los métodos tienen obligatoriamente un parámetro como mínimo, el objeto sobre el que el método ha sido invocado (**self**). Hay que tener en cuenta que este mecanismo es automático y que no hace falta que nosotros pasemos ese primer argumento al llamar al método, solo los siguientes parámetros.

```
print(p.imprimir()) # no se pasa el self, es automático
```

Funciones en la programación estructurada

Python permite diversos paradigmas de programación y se pueden mezclar en un mismo fichero. En concreto el paradigma de programación estructurada determina crear códigos independientes que realizan una función y pueden o no retornar un valor: funciones y procedimientos. La definición es similar a la que veamos de un método, pero al nivel cero de indentación, fuera de las clases. Para llamar a una función se utilizará su nombre y entre paréntesis se le pasarán los parámetros correspondientes.

Todo lo explicado en este punto también se puede extrapolar a la creación de funciones.

Diferencia entre procedimientos y funciones

Formalmente una función y un procedimiento no es lo mismo. Una función es una sección de un programa que calcula un valor de manera independiente al resto del programa que tiene tres partes: parámetros de entrada, proceso o código y resultado. Por otro lado, un procedimiento solo tiene dos partes: entrada y proceso, no devuelve nada.

A nivel de código la diferencia principal es que en los procedimientos no encontraremos una sentencia **return**, mientras que en las funciones sí.

Bajo Python, no existen los procedimientos, incluso si omitimos la sentencia **return** en nuestra definición, el intérprete devolverá de forma predefinida **None** como valor de nuestra función.

Para Python como para el resto de lenguajes, una función es un bloque de código con autonomía propia de ejecución que podremos reutilizar. La mayor diferencia con los lenguajes tradicionales es que es un objeto con sus propiedades y métodos. Una función se puede definir en cualquier parte del código, aunque es recomendable estructurar el código en módulos.

Definición

```
def fib(n):      # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
    return False
```

La palabra reservada **def** se usa para definir métodos. Debe seguirle el nombre del método y la lista de parámetros formales entre paréntesis (si es un método debe incluir obligatoriamente **self**). Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente y deben estar con sangría. La sentencia **return** final es optativa y representa el valor devuelto por la función cuando es utilizada como parte derecha de una asignación.

Esta definición se encontrará dentro de una definición de clase si es un método o fuera de ella si es una función, el resto del apartado trataremos como si se encontrase dentro de una clase, pero omitiremos el primer parámetro.

La primera sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o docstrings para documentación.

Ámbito de las variables

El ámbito es la zona de código que tiene acceso una variable. Python, como todos los lenguajes, tiene varios ámbitos para definir las variables uno **local**, uno **nonlocal** y otro **global**. Por defecto estos ámbitos no se mezclan y no podemos hacer uso de uno dentro del otro. El ámbito local se corresponde con el nivel de función, el ámbito global con el de programa, el **nonlocal** con una función interna.

Para modificar una variable definida en el programa deberemos anteponer la palabra clave **global** y el nombre de la variable para poder modificarla. Nunca podremos cambiar una variable local fuera de la función que la defina.

```
mi_var_global = 3

def mi_fun():
    global mi_var_global  #Quitar global y ver qué pasa
    mi_var_global = 3
    print(mi_var_global)
```

Para las clases el ámbito clase y local se mantiene, de tal manera que cualquier variable definida dentro de un método será local a toda la clase, por lo que se podrá hacer uso de ella a través del objeto **self**.

En una clase no se debería acceder nunca a una variable **global**.

Del mismo modo, si dentro de una función interna a otra, queremos acceder al bloque de la función padre deberemos usar la palabra clave **nonlocal**, que nos da acceso al bloque inmediatamente superior, como en el ejemplo siguiente.

```
g = 5
def miFunc2():
    global g
    g = 6
    b = 6
    def miInterna():
        nonlocal b  # accedemos a la b externa
        global g
        b = 7
        g = 8
    miInterna()
    print(b, g)

miFunc2() # 7,8
print(g) # 8
```

Parámetros

Un parámetro es un valor que se le pasa al método o función, un valor que espera para realizar su trabajo. Como Python define el tipo de una variable (parámetro en este caso) en tiempo de ejecución, no es necesario determinarlo en el parámetro, y no solo eso, el mismo parámetro aceptará cualquier tipo de datos para su ejecución, por lo que tendremos que realizar una conversión explícita si fuera necesario.

```
def suma(numero_1, numero_2):
    print(numero_1 + numero_2)

suma(2,3)
suma(2) # Error faltan parámetros
```

Parámetros pasados por Valor - por Referencia

Dependiendo del tipo de dato que envíemos a la función o método, podemos diferenciar dos comportamientos:

- **Paso por valor:** Se crea una copia local de la variable dentro de la función, los cambios internos no se reflejan en la variable externa.
- **Paso por referencia:** Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera. Se pasa la dirección del objeto.

Tradicionalmente

- Los tipos simples se pasan por valor: Enteros, flotantes, cadenas, lógicos...
- Los tipos compuestos se pasan por referencia: Listas, diccionarios, conjuntos, tuplas, ...

```
def mi_funcion(numero_1, obj):
    numero_1 = 1
    obj.nombre = "Nombre"
    print(id(numero_1))
    print(obj)

valor = 2
p = Persona("Juan", 23)
print(valor, p.nombre)

print(p)
print(id(valor))
mi_funcion(valor, p)
print(valor, p.nombre)

# 2 Juan
# <__main__.Persona object at 0x000002736CE34F70>
# 2694761769296
# 2694761769264
# <__main__.Persona object at 0x000002736CE34F70>
# 2 Nombre
```

Se puede comprobar como el objeto Persona es el mismo dentro y fuera de la función, se pasa la referencia de memoria, pero el parámetro número se crea una nueva copia, el identificador es diferente.

Se debe proporcionar todos los parámetros obligatorios a la hora de llamar a la función. Un parámetro es obligatorio si en la definición no se le ha proporcionado un valor por defecto.

Parámetros de Entrada – Salida

El comportamiento común de un parámetro es como entrada, esto quiere decir que recibirá un dato almacenado en el parámetro enviado desde el contexto en el que se llama a la función, se utilizará dentro de la función y no se modificará.

Los parámetros de salida permiten modificar el valor de una variable externa a la función relacionada con el parámetro. De esta manera, si la función modifica el valor del parámetro, la variable externa relacionada a este parámetro se modificará también, inclusive después de que la función haya terminado de ejecutarse.

En Python todos los parámetros son de entrada, pero no confundir con que se pasen por referencia.

Parámetros por defecto

Al igual que en muchos lenguajes se pueden usar parámetros por defecto, deben ir a continuación de los parámetros tradicionales. Estos parámetros si no se pasan en la llamada tendrán el valor establecido en la definición.

```
def miFuncion(a, b = 0):
    print(a , b)

miFuncion(2)      # 2 0
```

Los parámetros por defecto se evalúan solo una vez, cuando se define la función, por lo que el siguiente código el valor por defecto para **arg** es 5 y no 6.

```
i = 5
def f(arg=i):
    print(arg)
i = 6
f()  #5, ya que se ha evaluado la i al definir la función
f(arg=7) #7
```

Paso de parámetros por palabras clave

Python no es un lenguaje tan rígido como otros, con lo que permite algunas licencias en el uso de los métodos o funciones. En Python los parámetros se pueden enviar a una función Python o bien por posición o explícitamente por clave. Por posición lo hemos visto en los puntos anteriores, por clave el orden no importa y se antepone al valor el nombre del parámetro.

```
def suma(numero_1, numero_2):
    print(numero_1 + numero_2)

suma(numero_2=2, numero_1=3)
```

Aunque la versatilidad de Python permite definir si es necesario la rigidez de otros lenguajes, indicando explícitamente qué parámetros son posicionales, cuáles se deberán pasar por clave y cuáles podrán utilizarse de forma indistinta.

```
def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)

combined_example(1, 2, kwd_only=3)    # 1 2 3
```

```
combined_example(1, standard=2, kwd_only=3) # 1 2 3
```

Número indeterminado de parámetros

Cuando no sabemos el número de parámetros a recibir podemos usar el operador asterisco para juntarlos todos en una tupla como vemos en el ejemplo siguiente. Se puede mezclar este tipo de parámetros con cualquiera siempre que se sitúe después de los parámetros por defecto y antes de los parámetros del siguiente punto.

```
def miFuncion(a, b = 0, *resto):
    print(a, b, resto)
    print(type(resto))

miFuncion(2, 5, "resto", 5)
```

Desempaquetado de parámetros

En este caso, similar al anterior, se recogen en un **diccionario** todos los parámetros no definidos, pero es obligatorio pasarlos por nombre, que serán las claves del diccionario. Si usamos este tipo de parámetros debe de situarse obligatoriamente al final de la definición, después de los parámetros indeterminados.

```
def miFuncion(a, b = 0, **resto):
    print(a, b)
    print(resto)

miFuncion(2, 5, c = "resto", d = 5)
# 2 5
# {'c': 'resto', 'd': 5}
```

Sugerencias de tipos en los parámetros

El lenguaje permite sugerencias de tipo para los parámetros, pero no se utilizan para forzar ninguna conversión ni generan errores en caso que el tipo no concuerde, tal y como se ve en el segundo ejemplo.

```
miFuncion(a:int, b:str) -> int:
    return str(a)+str(b) # no se fuerza a int

print(miFuncion("3",5)) #espera int, str
# 35, se pasa: str, int, al revés
```

Devolución de valores

Hemos dicho que los métodos o funciones tienen tres partes, y el resultado es la última de ellas. Ya hemos comentado que Python solo implementa funciones con lo que, si no proporcionamos un valor de vuelta, éste será **None**.

Para devolver un valor utilizaremos la palabra clave **return** y a continuación el valor a devolver. No es un buen hábito de programación que existan en el método varios **returns**, siendo generalmente esta instrucción siempre la última de la función.

```
def suma(numero_1, numero_2):
    return int(numero_1) + int(numero_2)

print(suma(2, 3)) # 5
print(suma("2", 3)) # 5
```

Python no puede devolver valores múltiples, siempre devolverá un único valor. Pero nada impide devolver un objeto que contenga otros valores, como por ejemplo tuplas, diccionarios y otros objetos. Estos tipos de datos se verán en la siguiente unidad de trabajo.

```

def suma(numero_1, numero_2):
    return (numero_1, numero_2) # devuelve 1 tupla no dos valores

print(suma(2, 3)) # (2,3)
print(suma("2",3)) # ('2', 3)

```

Funciones lambda o anónimas

Las funciones lambda son funciones sin nombre que se suelen usar aquellos parámetros que esperan una función para realizar su trabajo, como puede ser la ordenación.

```

def hacer_incrementador(n): #lambda
    return lambda x: x + n #devolvemos una función

f = hacer_incrementador(42)
#f es una función hacer_incrementador inicializada a 42
print (f(1)) #43

for x in filter(lambda i: i % 2 == 0, [1, 2, 3, 4]):
    print(x)

```

Paso de métodos a funciones

```

class Persona:
    def es_mayor_De_edad(self):
        return self.edad >= 18

    def imprimir(self):
        return self.nombre

# eliminar la línea class, disminuir el sangrado, eliminar self,
# convertir variables de clase en parámetros

def es_mayor_De_edad(edad):
    return edad >= 18

def imprimir(nombre):
    return nombre

```

Creación y utilización de constructores y destructores

Un constructor es un método que se llama en el momento de la creación del objeto para inicialización del mismo. Un destructor se llama como última ejecución del objeto cuando es eliminado. Las llamadas a estos métodos son automáticas y no tenemos que hacer nada. Generalmente, los constructores se utilizan para inicialización del objeto y establecimiento de conexiones; los destructores para eliminación de basura y cierre de conexiones.

Python tiene dos métodos en vez de uno para la “construcción” del objeto: el constructor y el inicializador. Raramente vamos a usar constructor a no ser que ahondemos en Metaclasses por lo que usaremos exclusivamente el método inicializador: __init__. A efectos funcionales de nuestro nivel no hay diferencia.

En cuanto al destructor, utilizaremos el método __del__ que técnicamente no es un destructor, es llamado por el sistema cuando se recupera la memoria (el recolector de memoria) del objeto al destruirlo. El mayor problema que presenta este método es que el sistema no asegura que se llame al destructor, puede que el script termine antes de que pueda ser llamado y no hacer una liberación ordenada de los recursos.

```

class Persona:
    def __init__(self):

```

```
    print("Inicializando")

    def __del__(self):
        print("destruyendo")

p = Persona("Juan", 23)
p = None
```

Propiedades

Una de las críticas que se hace a Python desde los puristas de la POO es la no existencia de visibilidad en los métodos y atributos, pero cuando ahondamos en los usos de estas características vemos que no es imprescindible la visibilidad para una adecuada programación. El caso que se utiliza para minorar las capacidades de POO de Python hace referencia a aquellos en los que queremos controlar el acceso a un atributo, verificarlo antes de cambiarlo o realizar algún cálculo con él antes de la asignación, en definitiva, si necesitamos ejecutar código antes de la modificación del atributo (en otros lenguajes se implementa a través de los métodos getters y setters también presentes en Python).

Para solucionar este problema Python ofrece la creación de propiedades. Para Python una propiedad es un atributo que puede ejecutar código en el momento del acceso, que se define dentro de una clase mediante un decorador (Los decoradores se verán más adelante) o al crear un objeto **property** en la definición.

```
# Using property object
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    # getter
    def get_temperature(self):
        print("Getting value...")
        return self._temperature

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)

# Using @property decorator
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature
```

```

@temperature.setter
def temperature(self, value):
    print("Setting value...")
    self._temperature = value

human = Celsius(37)
print(human.temperature)
print(human.to_fahrenheit())

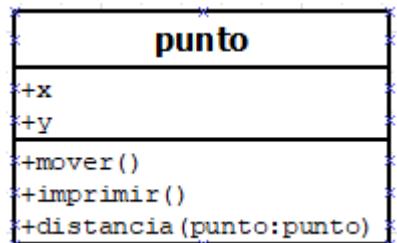
```

Ambas definiciones son similares y podremos usar cualquiera de las dos, pero no mezclarlas en una misma propiedad.

Con esta sintaxis, podemos hacer propiedades de solo lectura (con solo la parte **get**) o propiedades de lectura escritura con ambas partes. En cualquier caso, una propiedad definida con este mecanismo no es diferente a la creada en un constructor (Este mecanismo es conocido en muchos lenguajes POO como métodos **getters** and **setters**).

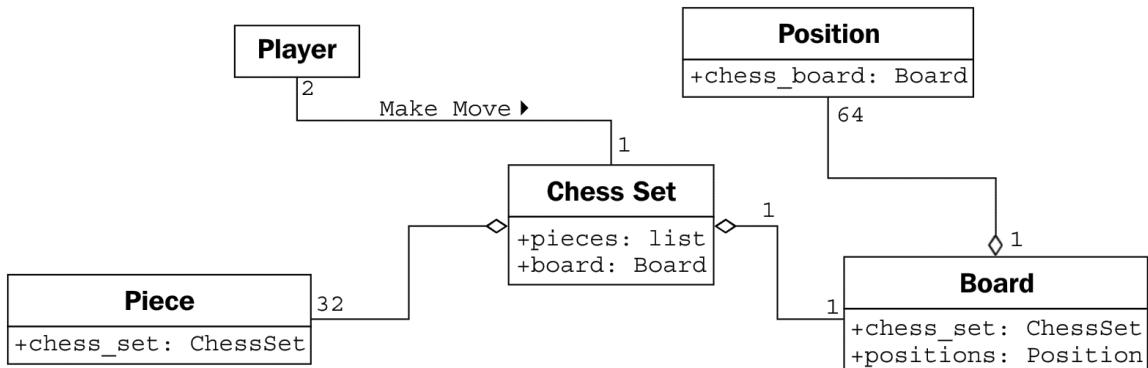
Ejemplos

Ejemplo: Definir la siguiente estructura de clases.



code_0002.py

Ejemplo: Definir la siguiente estructura de clases.



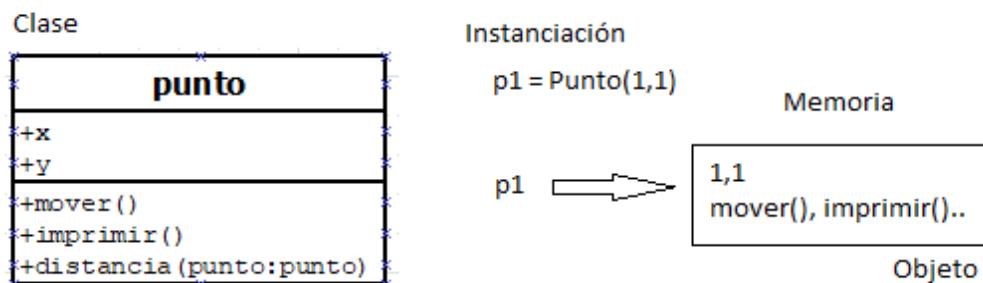
code_0003.py

Objetos

Desde un punto de vista técnico, un objeto es una instancia de una clase. Cuando creamos una instancia estamos reservando una zona de memoria dedicada para el objeto en cuestión y esa zona va a permanecer ahí mientras la variable en cuestión (u otra) la referencia. Tras leer lo anterior podemos pensar en un objeto como que un puntero a la zona de memoria que ocupa, pero en POO no suele hablarse siguiendo esa

terminología. A diferencia de otros lenguajes de programación, no es necesario liberar las zonas de memoria cuando dejan de utilizarse. Existe un mecanismo llamado recolector de basura que se encarga de ir liberando las zonas de memoria que no están siendo referenciadas por ningún objeto.

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos.



Creación y uso de objetos

Un objeto no se define, se crea o instancia en una variable a partir de un modelo (clase) de la que recoge todas sus características. Una vez instanciado se puede hacer uso de los atributos y métodos de la clase que hemos utilizado de base.

Si vemos el ejemplo anterior, el objeto **p1** se instancia de la clase punto y en nuestro código a través de la notación punto podremos hacer uso de los atributos x e y así como de los métodos.

```
human = Celsius(37) # constructor
print(human.temperature)
print(human.to_fahrenheit())
```

Cuando creamos un objeto, hacemos uso de su “constructor” de forma automática, este constructor se usa para inicializar el objeto con los datos iniciales. Si no proporcionamos todos los datos que son obligatorios no será posible instanciar el objeto.

Una vez instanciado el objeto deberemos hacer uso del mismo. En Python el acceso a propiedades o métodos se hace a través del operador punto (.).

```
human.temperature = 39
print(human.temperature)
print(human.to_fahrenheit())
```

Visibilidad

Visibilidad	Public	Private	Protected
Desde la misma clase	✓	✓	✓
Desde una subclase	✓	✗	✓
Desde otra clase (no subclase)	✓	✗	✗

La visibilidad permite que los métodos y propiedades de una clase sean utilizados de forma correcta por otros elementos del programa.

En la imagen podemos ver las clases de visibilidad que se pueden implementar bajo un lenguaje POO tradicional: pública, privada y protegida.

En Python la visibilidad es mucho más sencilla y solo implementa visibilidad pública, todos los métodos y

propiedades son accesibles desde fuera del objeto. Aunque solo se impone la visibilidad pública, podemos indicar a otros programadores que no deben hacer uso de un atributo o método anteponiendo un subrayado (_) al nombre. Esto **no impedirá nada**, no se controlará nada, es simplemente una sugerencia al programador.

```
class Celsius:  
    def __init__(self, temperature=0):  
        self._temperature = temperature # privada  
  
    def _to_fahrenheit(self): # protegido  
        print("en el método", self.__get_temperature())  
        return (self._temperature * 1.8) + 32  
  
    def __get_temperature(self): # privado  
        print("Getting value...")  
        return self._temperature  
  
human = Celsius(37)  
  
print(human._temperature)  
print(human._to_fahrenheit())  
print(dir(human))  
print(human._Celsius__get_temperature()) # cambio de nombre
```

Si aun así tenemos la imperiosa necesidad de impedir el acceso a un atributo o método, se puede anteponer al nombre dos subrayados (_) de tal manera que “impedirá” de forma externa al objeto el acceso (*Nota:* En verdad no se impide, se accede a una característica de renombrado o **name mangling** en la que internamente mantiene el nombre original y externamente tiene otro nombre (_NombreClase_nombre_método), si conocemos las reglas de nombrado podremos acceder desde fuera también, pero allá cada uno con no seguir lo que se le sugiere. Internamente mantiene el nombre original para las llamadas).

No se recomienda el uso de dobles subrayados, en su lugar si queremos decir que una variable es privada usaremos la nomenclatura de un subrayado.

Ejercicio

Desarrolla

Crea las clases **Bicicleta** y **Coche**. Crea los atributos de instancia kilómetros_recorridos, kilómetros_totales y color. Crea también algún método específico para cada una de las clases, los que necesites para implementar el menú VEHÍCULOS.

Prueba las clases creadas implementando dos objetos e interactuando con ellos mediante un programa con un menú como el que se muestra a continuación.

```
Menú crear objetos:  
¿Qué objeto deseas crear:  
1. Coche  
2.Bicicleta
```

Dicho programa debe ir pidiendo datos al usuario para construir el objeto deseado, una vez creado el objeto debe aparecer el siguiente menú.

```
VEHÍCULOS  
=====  
1. Anda con la bicicleta  
2. Haz el caballito con la bicicleta  
3. Anda con el coche  
4. Quema rueda con el coche  
5. Ver kilometraje de la bicicleta  
6. Ver kilometraje del coche  
7. Ver kilometraje total  
8. Salir  
Elige una opción (1-8):
```

- ∞ Añade a los métodos ver kilometraje una opción en la que si la bicicleta tiene más de 10.000 kilómetros o el coche más de 100.000 recomiende cambiar el vehículo.
- ∞ Añade una opción por si se desea volver al menú de crear objeto.
- ∞ Ten en cuenta que solo podrá existir un objeto de cada, si el usuario decide crear uno nuevo los datos se sobrescribirán sobre el anterior

code_0004

Paquetes

Módulo

Un módulo es un fichero Python que contiene código. Potencialmente cualquier fichero puede ser un módulo, o ser llamado para la ejecución. Como tal podemos determinar si un fichero ha sido llamado como módulo o para la ejecución a través de la variable global `__name__`. Esta variable contendrá el valor: `__main__` cuando se esté intentando ejecutar el código, en caso contrario el módulo se intentará usar como librería de importación.

Las variables que se definan en un módulo en el ámbito global serán accesibles desde el módulo y desde fuera como una función más.

```
def main():
    print("Ejecutando el código")

if __name__ == "__main__":
    main()
else:
    print("importando datos")
```

Paquete

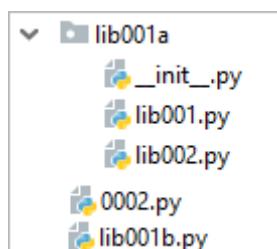
Un paquete (**namespace**) es un directorio en el que se le crea un directorio especial: `__init__.py` (**optativo a partir de la versión 3.3**). Este fichero puede estar vacío o puede tener una variable `__all__`, con lista de los módulos al cargar cuando se use. Además, cualquier variable o definición que se haga en este fichero, estará disponible para todos los módulos del paquete usando el nombre del mismo para importarlo (**namespace**).

```
pqt_1\
    __init__.py
        from .database import db

    modulo_1.py
        from pqt_1 import db
```

Ejemplo

En este ejemplo desarrollaremos una librería muy básica. En primer lugar, se crea un módulo librería (`lib001b.py`) y a continuación el paquete `lib001a` con dos módulos dentro. El fichero `0002.py` hace uso de las librerías y todos sus ejemplos de importación.



```
lib001b.py
def suma(*valores):
    rest=0
    for val in valores:
        rest+=int(val)
    return rest
```

```
lib001.py
varg=3
def suma(*valores):
    rest=0
    for val in valores:
        rest+=val
    print (varg)
    return rest
```

```
lib002.py
varg=5
def suma(*valores):
    rest=0
    for val in valores:
        rest+=val
    print (varg)
    return rest
```

```
0002.py
import lib001b # se importa todo el módulo
import lib001a.lib001 # se importa un módulo de un pqt.
import lib001a.lib002 as lb # se renombra un módulo

print(lib001b.suma(1, 3, 5, 7))          #16
print(lib001a.lib001.suma(1,2,3,4))      #3 10
print(lb.suma(1,2,3,4))                  #5 10
lb.varg = 99    #acceso a la variable de módulo
print(lb.suma(1,2,3,4))                  #99 10

from lib001b import suma
print(suma(1,2,3,4))                      #10
```

Uso de import

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos, pero para ello, es necesario importar los módulos que se quieran utilizar. Para importar un módulo, se utiliza la instrucción **import**, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin extensión) que se desee importar.

Todo código en un paquete importado será ejecutado en el momento de la importación, pero no se ejecutarán las clases o funciones definidas en él, hay que llamarlas de forma explícita.

```
import modulo # un módulo que no pertenece a un paquete
import paquete.modulo1 # un módulo que está dentro de un paquete
import paquete.subpaquete.modulo1

print modulo.CONSTANTE_1
print paquete.modulo1.CONSTANTE_1
print paquete.subpaquete.modulo1.CONSTANTE_1
```

Es posible también, importar mediante un alias. Para ello, durante la importación, se asigna la palabra clave **as** seguida del alias con el cual nos referiremos en el futuro:

```

import modulo as m
import paquete.modulo1 as pm
import paquete.subpaquete.modulo1 as psm

print m.CONSTANTE_1
print pm.CONSTANTE_1
print psm.CONSTANTE_1

```

En Python, es posible también, importar de un módulo solo los elementos que se desee utilizar. Para ello se utiliza la instrucción **from** seguida del **namespace**, más la instrucción **import** seguida del elemento que se desee importar. En este caso, se accederá directamente al elemento, sin recurrir a su **namespace**.

```

from paquete.modulo1 import CONSTANTE_1
print CONSTANTE_1

```

Es posible también, importar más de un elemento en la misma instrucción. Para ello, cada elemento irá separado por una coma (,) y un espacio en blanco.

```

from paquete.modulo1 import CONSTANTE_1, CONSTANTE_2

```

Pero ¿qué sucede si los elementos importados desde módulos diferentes tienen los mismos nombres? En estos casos, habrá que prevenir fallos, utilizando alias para los elementos:

```

from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2
from paquete.subpaquete.modulo1 import CONSTANTE_1 as CS1,
from .paquete.subpaquete.modulo1 import CONSTANTE_2 as CS2
print C1
print C2
print CS1
print CS2

```

Se puede anteponer un punto(.) o dos puntos (..) indicando que la búsqueda del paquete debe empezar en el directorio del fichero en vez del proyecto.

Recomendaciones de importación (PEP 8: importación)

- ∞ La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos.
- ∞ Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación.
- ∞ Entre cada bloque de **imports**, debe dejarse una línea en blanco.

Destrucción de objetos y Liberación de memoria

El recolector de basura o **garbage collector** es una de las piezas fundamentales del lenguaje, es la funcionalidad que libera al programador de la solicitud y liberación de memoria de forma explícita lo que facilita a los programadores la creación de programas, una mayor productividad, evita errores y fallos de seguridad. Uno de los motivos por los que programar en Python es cómodo y sencillo es que no tenemos que preocuparnos por el tiempo de vida de nuestros objetos ya que implementa el garbage collector. Es decir, una vez que deja de ser necesaria, una variable desaparece de la memoria “mágicamente”.

Python utiliza para manejar la memoria una estrategia combinada consistente en: conteo de referencias y colector de basura generacional.

- ∞ Conteo de referencias, La idea debajo del mismo es muy simple, se lleva la cuenta de las referencias que un objeto en memoria tiene y cuando éste pierde todas las referencias se llama de llama

- automáticamente al destructor específico del objeto (`__del__`). Las variables definidas dentro de bloques tienen un alcance local, pertenecen al bloque y su vida se limita al tiempo de ejecución del mismo. Cuando se sale del bloque se destruye todas las referencias creadas dentro del mismo. Las variables globales viven hasta el final del proceso de Python si no se eliminan explícitamente mediante `del`, Por lo que el conteo de referencias a los objetos que apuntan nunca cae a 0. Hay que tener en cuenta que `del` nunca elimina un objeto ni libera la memoria que usa. Esto solo ocurre si el nombre (variable) era la única referencia al objeto, en cuyo caso del nombre hace que el contador de referencias pase a 0 para el objeto asociado y el GC lo deslocalizará.
- ∞ Colector de basura generacional o cíclico. Una de las mayores desventajas del recolector de basura por conteo de referencias es que es incapaz de lidiar con referencias cíclicas. Las referencias cíclicas solo pueden darse en objetos que puedan contener a otros objetos como listas, clases, diccionarios, etc. El recolector de basura generacional solo monitoriza objetos susceptibles de producir referencias cíclicas, excluyendo a muchos otros principalmente inmutables.

La gestión del recolector de basuras

El módulo `gc` proporciona una interfaz para el recolector de basura opcional (recolector de basura cíclico generacional). Proporciona la capacidad de deshabilitar el recolector, ajustar la frecuencia de recolección y establecer opciones de depuración.

También proporciona acceso a objetos inaccesibles (`unreachable`) que el recolector encontró, pero no pudo liberar. Dado que el recolector de basura complementa el conteo de referencias es posible desactivarlo siempre que se esté seguro de que el programa no crea referencias cíclicas.

El módulo `gc` proporciona las siguientes funciones:

- ∞ `gc.enable()`. Habilita la recolección automática de basura.
- ∞ `gc.disable()`. Deshabilita la recolección automática de basura.
- ∞ `gc.isenabled()`. Retorna True si la recolección automática está habilitada.
- ∞ `gc.collect(generation=2)`. Sin argumentos, ejecuta una recolección completa. El argumento opcional `generation` debe ser un número entero que especifica qué generación recolectar (de 0 a 2. Se retorna el número de objetos inaccesibles encontrados).

Las listas libres mantenidas para varios tipos incorporados son borradas cada vez que se ejecuta una recolección completa o una recolección de la generación más alta. No obstante, no todos los elementos de algunas listas libres pueden ser liberados, particularmente `float`, debido a su implementación particular.

Librería estándar

Cadenas, clase str

Para un uso básico de la clase `str` o cadenas de caracteres revisar la U.T. 3.

El tipo `str` es una secuencia inmutable de caracteres Unicode (cuidado cuando trabajes con texto procedente de ficheros u otras fuentes de datos. Fíjate en qué codificación está y haz las transformaciones necesarias si no quieres tener problemas. Por defecto, la codificación de un `string` en Python es `Unicode`, concretamente UTF-8).

Una singularidad de la clase `str` es que a su constructor se le puede pasar cualquier objeto. Al hacer esto, la función `str()` devuelve la representación en forma de cadena de caracteres del propio objeto (si se pasa un `string` devuelve el `string` en sí). Normalmente, al llamar a la función `str(objeto)` lo que se hace internamente es llamar al método `__str__()` del objeto. Si este método no existe, entonces devuelve el resultado de invocar a `repr(objeto)` o `__repr__()`.

Cadenas de caracteres.

Todas las cadenas de caracteres implementan las operaciones comunes de las secuencias, junto con los métodos descritos a continuación.

- ∞ **str.capitalize()**. Retorna una copia de la cadena con el primer carácter en mayúsculas y el resto en minúsculas. Distinto en la versión 3.8: El primer carácter se pasa ahora a título, más que a mayúsculas. Esto significa que caracteres como dígrafos solo tendrán la primera letra en mayúsculas, en vez de todo el carácter.
- ∞ **str.casefold()**. Retorna el texto de la cadena, normalizado a minúsculas. Los textos así normalizados pueden usarse para realizar búsquedas textuales independientes de mayúsculas y minúsculas. El texto normalizado a minúsculas es más agresivo que el texto en minúsculas normal, porque se intenta unificar todas las grafías distintas de las letras.
- ∞ **str.center(width[, fillchar])**. Retorna el texto de la cadena, centrado en una cadena de longitud *width*. El relleno a izquierda y derecha se realiza usando el carácter definido por el parámetro *fillchar* (Por defecto se usa el carácter espacio ASCII). Si la cadena original tiene una longitud *len(s)* igual o superior a *width*, se retorna el texto sin modificar.
- ∞ **str.count(sub[, start[, end]])**. Retorna el número de ocurrencias no solapadas de la cadena *sub* en el rango *[start, end]*. Los parámetros opcionales *start* y *end* se interpretan como en una expresión de rebanada.
- ∞ **str.encode(encoding="utf-8", errors="strict")**. Retorna una versión codificada en forma de bytes. La codificación por defecto es 'utf-8'. El parámetro *errors* permite especificar diferentes esquemas de gestión de errores. El valor por defecto de *errors* es 'strict', que significa que cualquier error en la codificación eleva una excepción de tipo UnicodeError. Otros valores posibles son 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' y cualquier otro nombre que se haya registrado mediante la función *codecs.register_error()*.
- ∞ **str.endswith(suffix[, start[, end]])**. Retorna True si la cadena termina con el sufijo especificado con el parámetro *prefix*, y False en caso contrario. También podemos usar *suffix* para pasar una tupla de sufijos a buscar. Si especificamos el parámetro opcional *start*, la comprobación empieza en esa posición. Con el parámetro opcional *stop*, la comprobación termina en esa posición.
- ∞ **str.expandtabs(tabsize=8)**. Retorna una copia de la cadena, con todos los caracteres de tipo tabulador reemplazados por uno o más espacios, dependiendo de la columna actual y del tamaño definido para el tabulador. Las posiciones de tabulación ocurren cada *tabsize* caracteres (Siendo el valor por defecto de *tabsize* 8 (lo que produce las posiciones de tabulación 0, 8, 16,...)). Para expandir la cadena, la columna actual se pone a cero y se va examinando el texto carácter a carácter. Si se encuentra un tabulador, (\t), se insertan uno o más espacios hasta que sea igual a la siguiente posición de tabulación (El carácter tabulador en sí es descartado). Si el carácter es un indicador de salto de línea (\n) o de retorno (\r), se copia y el valor de columna actual se vuelve a poner a cero. Cualquier otro carácter es copiado sin cambios y hace que el contador de columna se incremente en 1, sin tener en cuenta como se representa gráficamente el carácter.
- ∞ **str.find(sub[, start[, end]])**. Retorna el menor índice de la cadena *s* donde se puede encontrar la cadena *sub*, considerando solo el intervalo *s[start:end]*. Los parámetros opcionales *start* y *end* se interpretan como si fueran "índices" de una slice. retorna -1 si no se encuentra la cadena.
El método *find()* se debe usar solo si se necesita saber la posición de la cadena *sub*. Si solo se necesita comprobar si *sub* es una parte de *s*, es mejor usar el operador *in*:

```
'Py' in 'Python'  
True
```

- ∞ **str.format(*args, **kwargs)**. Realiza una operación de formateo (Ver tema 3).
- ∞ **str.index(sub[, start[, end]])**. Como *find()*, pero eleva una excepción de tipo ValueError si no se encuentra la cadena a buscar.
- ∞ **str.isalnum()**. Retorna True si todos los caracteres de la cadena son alfanuméricos y hay, al menos, un carácter.

- ∞ **str.isalpha()**. Retorna True si todos los caracteres de la cadena son alfabéticos y hay, al menos, un carácter.
- ∞ **str.isascii()**. Retorna True si la cadena de caracteres está vacía, o si todos los caracteres de la cadena son ASCII.
- ∞ **str.isdecimal()**. Retorna True si todos los caracteres de la cadena son caracteres decimales y hay, al menos, un carácter.
- ∞ **str.isdigit()**. Retorna True si todos los caracteres de la cadena son dígitos y hay, al menos, un carácter.
- ∞ **str.isidentifier()**. Retorna True si la cadena de caracteres es un identificador válido de acuerdo a la especificación del lenguaje.
- ∞ **str.islower()**. Retorna True si todos los caracteres de la cadena están en minúsculas y hay, al menos, un carácter de ese tipo.
- ∞ **str.isnumeric()**. Retorna True si todos los caracteres de la cadena son caracteres numéricos y hay, al menos, un carácter.
- ∞ **str.isprintable()**. Retorna True si todos los caracteres de la cadena son imprimibles o si la cadena está vacía.
- ∞ **str.isspace()**. Retorna True si todos los caracteres de la cadena son espacios en blanco y hay, al menos, un carácter. Un carácter se considera espacio en blanco si, en la base de datos de Unicode, está clasificado en la categoría general ("Espacio, separador") o la clase bidireccional es WS, B, or S.
- ∞ **str.istitle()**. Retorna True si las palabras en la cadena tienen forma de título y hay, al menos, un carácter.
- ∞ **str.isupper()**. Retorna True si todos los caracteres de la cadena están en mayúsculas y hay, al menos, un carácter de ese tipo
- ∞ **str.join(*iterable*)**. Retorna una cadena de caracteres formada por la concatenación de las cadenas en el *iterable*. Se eleva una excepción de tipo TypeError si alguno de los elementos en el *iterable* no es una cadena, incluyendo objetos de tipo bytes. Se usa como separador entre los elementos la cadena de caracteres pasada como parámetro.
- ∞ **str.ljust(*width*[, *fillchar*])**. Retorna el texto de la cadena, justificado a la izquierda en una cadena de longitud *width*. El carácter de relleno a usar viene definido por el parámetro *fillchar* (Por defecto se usa el carácter espacio ASCII). Si la cadena original tiene una longitud len(s) igual o superior a *width*, se Retorna el texto sin modificar.
- ∞ **str.lower()**. Retorna una copia de la cadena de caracteres con todas las letras en minúsculas.
- ∞ **str.lstrip([*chars*])**. Retorna una copia de la cadena, eliminado determinados caracteres si se encuentren al principio. El parámetro *chars* especifica el conjunto de caracteres a eliminar. Si se omite o si se especifica None, se eliminan todos los espacios en blanco. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:
- ∞ **str.partition(*sep*)**. Divide la cadena en la primera ocurrencia de *sep*, y retorna una tupla de tres elementos, conteniendo la parte anterior al separador, el separador en sí y la parte posterior al separador. Si no se encuentra el separador, Retorna una tupla de tres elementos, el primero la cadena original y los dos siguientes son cadenas vacías.
- ∞ **str.replace(*old*, *new*[, *count*])**. Retorna una copia de la cadena con todas las ocurrencias de la cadena *old* sustituidas por *new*. Si se utiliza el parámetro *count*, solo se cambian las primeras *count* ocurrencias.
- ∞ **str.rfind(*sub*[, *start*[, *end*]])**. Retorna el mayor índice dentro de la cadena *s* donde se puede encontrar la cadena *sub*, estando *sub* incluida en *s[start:end]*. Los parámetros opcionales *start* y *end* se interpretan igual que en las operaciones de slice. retorna -1 si no se encuentra *sub*.
- ∞ **str.rindex(*sub*[, *start*[, *end*]])**. Como el método rfind(), pero eleva la excepción ValueError si no se encuentra la cadena *sub*.
- ∞ **str.rjust(*width*[, *fillchar*])**. Retorna el texto de la cadena, justificado a la derecha en una cadena de longitud *width*. El carácter de relleno a usar viene definido por el parámetro *fillchar* (Por defecto se usa el carácter espacio ASCII). Si *width* es menor o igual que len(s), se retorna el texto sin modificar.
- ∞ **str.rpartition(*sep*)**. Divide la cadena en la última ocurrencia de *sep*, y retorna una tupla de tres elementos, conteniendo la parte anterior al separador, el separador en sí y la parte posterior al

- separador. Si no se encuentra el separador, Retorna una tupla de tres elementos, las dos primeras posiciones con cadenas vacías y en la tercera la cadena original.
- ∞ **str.rsplit(*sep=None, maxsplit=-1*)**. Retorna una lista con las palabras que componen la cadena de caracteres original, usando como separador el valor de *sep*. Si se utiliza el parámetro *maxsplit*, se realizan como máximo *maxsplit* divisiones, retornando los que están más a la derecha. Si no se especifica *sep* o se pasa con valor None, se usa como separador cualquier carácter de espacio en blanco. Si no contamos la diferencia de empezar las divisiones desde la derecha, el comportamiento de este método rsplit() es equivalente al de split().
 - ∞ **str.rstrip([*chars*])**. Retorna una copia de la cadena, eliminado determinados caracteres si se encuentren al final. El parámetro *chars* especifica el conjunto de caracteres a eliminar. Si se omite o si se especifica None, se eliminan todos los espacios en blanco. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres:
 - ∞ **str.split(*sep=None, maxsplit=-1*)**. Retorna una lista con las palabras que componen la cadena de caracteres original, usando como separador el valor de *sep*. Si se utiliza el parámetro *maxsplit*, se realizan como máximo *maxsplit* divisiones, (Por tanto, la lista resultante tendrá *maxsplit+1* elementos). Si no se especifica *maxsplit* o se pasa con valor -1, entonces no hay límite al número de divisiones a realizar (Se harán todas las que se puedan).
 - ∞ **str.splitlines([*keepends*])**. Retorna una lista con las líneas en la cadena, dividiendo por los saltos de línea. Los caracteres de salto de línea en sí no se incluyen a no ser que se especifique lo contrario pasando el valor True en al parámetro *keepends*. Este método considera como saltos de línea los siguientes caracteres. En concreto, estos son un superconjunto de los saltos de líneas universales.

Representación	Descripción
\n	Salto de línea
\r	Retorno de carro
\r\n	Retorno de carro + salto de línea
\v o \x0b	Tabulación de línea
\f o \x0c	Avance de página
\x1c	Separador de archivo
\x1d	Separador de grupo
\x1e	Separador de registro
\x85	Siguiente línea (Código de control C1)
\u2028	Separador de línea
\u2029	Separador de párrafo

- ∞ **str.startswith(*prefix[, start[, end]]*)**. Retorna True si la cadena empieza por *prefix*, en caso contrario Retorna False. El valor de *prefix* puede ser también una tupla de prefijos por los que buscar. Con el parámetro opcional *start*, la comprobación empieza en esa posición de la cadena.
- ∞ **str.strip([*chars*])**. Retorna una copia de la cadena con los caracteres indicados eliminados, tanto si están al principio como al final de la cadena. El parámetro opcional *chars* es una cadena que especifica el conjunto de caracteres a eliminar. Si se omite o se usa None, se eliminan los caracteres de espacio en blanco. No debe entenderse el valor de *chars* como un prefijo, sino que se elimina cualquier combinación de sus caracteres.
- ∞ **str.swapcase()**. Retorna una copia de la cadena con los caracteres en mayúsculas convertidos a minúsculas, y viceversa. Nótese que no es necesariamente cierto que s.swapcase().swapcase() == s.
- ∞ **str.title()**. Retorna una versión en forma de título de la cadena, con la primera letra de cada palabra en mayúsculas y el resto en minúsculas.
- ∞ **str.translate(*table*)**. Retorna una copia de la cadena en la que cada carácter ha sido sustituido por su equivalente definido en la tabla de traducción dada. La tabla puede ser cualquier objeto que soporta el acceso mediante índices implementado en método *__getitem__()*, normalmente un objeto de tipo mapa o secuencia. Cuando se accede como índice con un código Unicode (Un entero), el objeto tabla puede hacer una de las siguientes cosas: retornar otro código Unicode o retornar una cadena de

- caracteres, de forma que se usaran uno u otro como reemplazo en la cadena de salida; retornar `None` para eliminar el carácter en la cadena de salida, o elevar una excepción de tipo `LookupError`, que hará que el carácter se copie igual en la cadena de salida.
- ∞ **`str.upper()`**. Retorna una copia de la cadena, con todos los caracteres con formas mayúsculas o minúsculas pasados a minúsculas.
 - ∞ **`str.zfill(width)`**. Retorna una copia de la cadena, rellena por la izquierda con los caracteres ASCII '0' necesarios para conseguir una cadena de longitud `width`. El carácter prefijo de signo ('+'/'-') se gestiona insertando el relleno *después* del carácter de signo en vez de antes. Si `width` es menor o igual que `len(s)`, se retorna la cadena original.

Expresiones regulares.

Las expresiones regulares son un potente lenguaje de descripción de texto. Tanto los patrones como las cadenas de texto a buscar pueden ser cadenas de Unicode, así como cadenas de 8 bits. Sin embargo, las cadenas Unicode y las cadenas de 8 bits no se pueden mezclar: es decir, no se puede hacer coincidir una cadena Unicode con un patrón de bytes o viceversa; del mismo modo, al pedir una sustitución, la cadena de sustitución debe ser del mismo tipo que el patrón y la cadena de búsqueda.

Las expresiones regulares usan el carácter de barra inversa ('\') para indicar formas especiales o para permitir el uso de caracteres especiales sin invocar su significado especial. Esto choca con el uso de Python de este carácter para el mismo propósito con los literales de cadena; por ejemplo, para hacer coincidir una barra inversa literal, se podría escribir '\\\\' como patrón, porque la expresión regular debe ser \\, y cada barra inversa debe ser expresada como \\\\" dentro de un literal de cadena regular de Python.

La solución es usar la notación de cadena raw de Python para los patrones de expresiones regulares; las barras inversas no se manejan de ninguna manera especial en un literal de cadena prefijado con 'r'. Así que `r"\n"` es una cadena de dos caracteres que contiene '\n', mientras que "\n" es una cadena de un carácter que contiene una nueva línea. Normalmente los patrones se expresan en código Python usando esta notación de cadena raw.

<https://docs.python.org/es/3/library/re.html>

<https://docs.python.org/es/3/howto/regex.html#regex-howto>

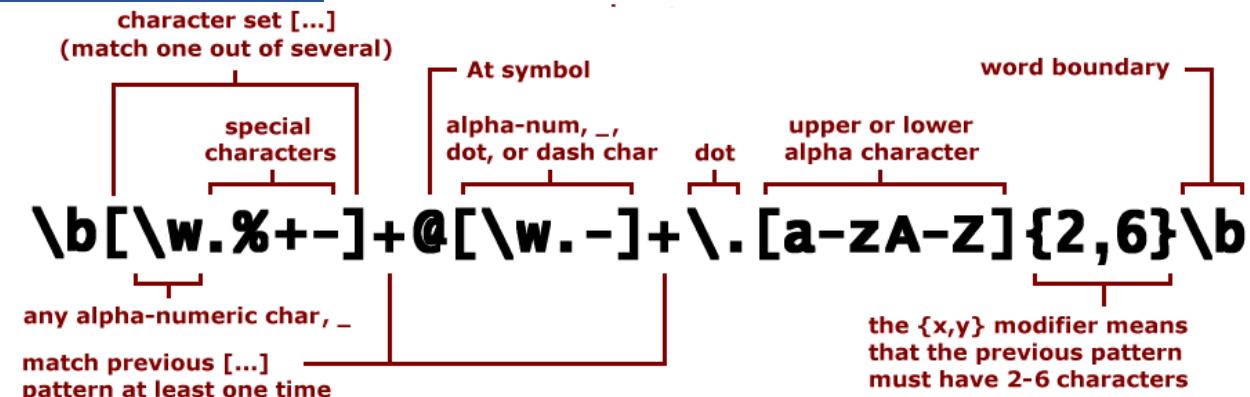
<http://w3.unpocodetodo.info/utiles/regex.php>

Patrones

Las reglas con las que se forman las expresiones son bastante simples y se denominan patrón. Pero aprender a combinarlas correctamente requiere de práctica. Utilizándolas podemos buscar una subcadena al principio o al final del texto. Incluso si queremos que se repita cierta cantidad de veces, si queremos que algo NO aparezca, o si debe aparecer una subcadena entre varias posibilidades. Permite, además, capturar aquellos trozos del texto que coincidan con la expresión para guardarlos en una variable o reemplazarlos por una cadena predeterminada; o incluso una cadena formada por los mismos trozos capturados.

Metacaracteres

Se conoce como metacaracteres a aquellos que, dependiendo del contexto, tienen un significado especial para las expresiones regulares. Por lo tanto, los debemos escapar colocándoles una contrabarra (\) delante para buscarlos explícitamente.

Elementos de una expresión**Parse: username@domain.TLD (top level domain)**

Una expresión es una cadena de texto formada por caracteres, metacaracteres o cualquier estructura que permita la sintaxis. Veamos los más importantes:

- ∞ **Cadena literal.** Coincide exactamente con lo escrito. `Abc` → buscará `Abc`.
- ∞ **Anclas:** Indican que lo que queremos encontrar se encuentra al principio o al final de la cadena. Combinándolas, podemos buscar algo que represente a la cadena entera:
 - `^patron`: coincide con cualquier cadena que comience con patrón.
 - `patron$`: coincide con cualquier cadena que termine con patrón.
 - `^patron$`: coincide con la cadena exacta patrón.
- ∞ **Clases de caracteres:** Se utilizan cuando se quiere buscar un carácter dentro de varias posibles opciones. Una clase se delimita entre corchetes y lista posibles opciones para el carácter que representa:
 - `[abc]`: coincide con a, b, o c
 - `[387ab]`: coincide con 3, 8, a o b
 - `niñ[oa]s`: coincide con niños o niñas.
 - Para evitar errores, en caso de que queramos crear una clase de caracteres que contenga un corchete, debemos escribir una barra `\` delante, para que el motor de expresiones regulares lo considere un carácter normal: la clase `[ab\[]` coincide con a, b y [.
- ∞ **Rangos.** Si queremos encontrar un número, podemos usar una clase como `[0123456789]`, o podemos utilizar un rango. Un rango es una clase de caracteres abreviada que se crea escribiendo el primer carácter del rango, un guion y el último carácter del rango. Múltiples rangos pueden definirse en la misma clase de caracteres.
 - `[a-c]`: equivale a `[abc]`
 - `[0-9]`: equivale a `[0123456789]`
 - `[a-d5-8]`: equivale a `[abcd5678]`
 - Es importante notar que si se quiere buscar un guion debe colocarse al principio o al final de la clase. Es decir, inmediatamente después del corchete izquierdo o inmediatamente antes del corchete derecho; o, en su defecto, escaparse. Si no se hace de esta forma, el motor de expresiones regulares intentará crear un rango y la expresión no funcionará como debe (o dará un error). Si queremos, por ejemplo, crear una clase que coincida con los caracteres a, 4 y -, debemos escribirla así:
 - `[a4-]`
 - `[-a4]`
 - `[a\4-]`
- ∞ **Rango negado.** Así como podemos listar los caracteres posibles en cierta posición de la cadena, también podemos listar caracteres que no deben aparecer. Para lograrlo, debemos negar la clase, colocando un circunflejo inmediatamente después del corchete izquierdo.
 - `[^abc]`: coincide con cualquier carácter distinto de a, b y c

- ∞ **Clases predefinidas.** Existen algunas clases que se usan frecuentemente y por eso existen formas abreviadas para ellas. En Python, así como en otros lenguajes, se soportan las clases predefinidas de Perl y de POSIX. Algunos ejemplos de expresiones regulares son:
 - \d (POSIX [:digit:]): equivale a [0-9]
 - \s (POSIX [:space:]): caracteres de espacio en blanco (espacio, tabulador, nueva línea, etc)
 - \w (POSIX [:word:]): letras minúsculas, mayúsculas, números y guion bajo (_)
 - ":" : coincide con cualquier carácter.
- ∞ **Cuantificadores.** Son conjuntos de caracteres que multiplican el patrón que les precede. Mientras que con las clases de caracteres podemos buscar un dígito, o una letra; con los cuantificadores podemos buscar cero o más letras, al menos 7 dígitos, o entre tres y cinco letras mayúsculas. Los cuantificadores son.
 - * cero o más, similar a {0,}.
 - + una o más, similar a {1,}.
 - ? cero o una, similar a {0,1}.
 - {n} exactamente n veces.
 - {n,} por lo menos n veces.
 - {n, m} por lo menos n, pero no más de m veces.
 - *? cero o más, similar a {0 , }?.
 - +? una o más, similar a {1, }?.
 - ?? cero o una, similar a {0,1}?.
 - {n}? exactamente n veces.
 - {n,}? por lo menos n veces.
 - {n, m}? por lo menos n pero no más de m veces.
- ∞ **Alternativa (|):** permite definir opciones para el patrón: perro|gato coincide con perro y con gato.
- ∞ **Secuencias de escape o metacaracteres.**
 - \n Nueva línea (new line). El cursor pasa a la primera posición de la línea siguiente.
 - \t Tabulador. El cursor pasa a la siguiente posición de tabulación.
 - \\ Barra diagonal inversa
 - \v Tabulación vertical.
 - \ooo Carácter ASCII en notación octal.
 - \xhh Carácter ASCII en notación hexadecimal.
 - \xhhhh Carácter Unicode en notación hexadecimal.

Ejemplos:

- ∞ .* : cualquier cadena, de cualquier largo (incluyendo una cadena vacía)
- ∞ [a-z]{3,6}: entre 3 y 6 letras minúsculas
- ∞ \d{4,}: al menos 4 dígitos
- ∞ .*hola!?: una cadena cualquiera, seguida de hola, y terminando (o no) con un !
- ∞ ^[(a-z0-9_\-\.)]+@[a-z0-9_\-\.]+\.[a-z]{2,15}. Valida un email.
- ∞ ^\(?[+|\d]{1,3}\)?\s?[\d]{1,5}\s?[\d][\s\-.]?{6,7}\\$. Para un número de teléfono.

Módulo re

Para utilizar Expresiones Regulares, Python provee el módulo re. Importando este módulo podemos crear objetos de tipo patrón y generar objetos tipo matcher, que son los que contienen la información de la coincidencia del patrón en la cadena.

Creando un patrón

Para crear un objeto patrón, debemos importar el módulo re y utilizamos la función compile.

```
import re

patron = re.compile(r'a[3-5]+')
# coincide con una letra, seguida de al menos 1 dígito entre
# 3 y 5
```

Desde este momento, podemos usar el objeto patrón para comparar cadenas con la expresión regular.

Buscar el patrón en la cadena

Para buscar un patrón en una cadena, Python provee los métodos **search** y **match**. La diferencia entre ambos es que, mientras **search** busca en la cadena alguna ocurrencia del patrón, **match** devuelve **None** si la ocurrencia no se da al principio de la cadena.

```
cadena = r'a44453'
patron.match(cadena) # <_sre.SRE_Match object at 0x02303BF0>
patron.search(cadena) # <_sre.SRE_Match object at 0x02303C28>
cadena = r'ba3455' # la coincidencia no está al principio!
patron.search(cadena) # <_sre.SRE_Match object at 0x02303BF0>
print patron.match(cadena) # None
```

Si sabemos que obtendremos más de una coincidencia, podemos usar el método **findall**, que recorre la cadena y devuelve una lista de coincidencias.

```
patron.findall(r'a455 a333b435') # ['a455', 'a333']
```

Reemplazo de cadenas

Similar a la combinación **search + expand**, existe el método **sub**; cuya función es encontrar todas las coincidencias de un patrón y sustituirlas por una cadena. Este recibe dos parámetros: el primero es la cadena con la que se sustituirá el patrón y el segundo es la cadena sobre la que queremos aplicar la sustitución. Y también se pueden utilizar referencias.

```
patron.sub(r"X", r'a455 a333b435')
# sustituye todas las ocurrencias del patrón por X 'X XX'
```

Modificadores para el patrón

Existen varios modificadores que podemos pasar al método **compile** para modificar el comportamiento del patrón. Los más usados son:

- ∞ **re.I o re.IGNORECASE**: hace que el patrón no distinga entre minúsculas y mayúsculas.
- ∞ **re.M o re.MULTILINE**: modifica el comportamiento de ^ y \$ para que coincidan con el comienzo y final de cada línea de la cadena, en lugar de coincidir con el comienzo y final de la cadena entera
- ∞ **re.S o re.DOTALL**: hace que el punto(.) coincida además con un salto de línea. Sin este modificador, el punto coincide con cualquier carácter excepto un salto de línea

Cada modificador se usa como segundo parámetro de la función. Podemos unir los efectos de más de un modificador separándolos con |.

```
patron = re.compile(r'el patron', re.I | re.MULTILINE)
```

División según el patrón

El método **re.split**: divide una cadena a partir de un patrón.

```
print(re.split(r' ', texto))
```

Math

Funciones numéricas

- ∞ math.ceil(x): Devuelve el entero más próximo mayor o igual que x.

```
math.ceil(5.4)
```

6

```
math.ceil(-5.4)
```

-5

- ∞ math.floor(x): Devuelve el entero más próximo menor o igual que x.
- ∞ math.gcd(a, b): Devuelve el máximo común divisor ("greatest common divisor") de los números a y b.

```
math.gcd(6, 21)
```

3

- ∞ math.isnan(x): Devuelve el booleano True si x es un NaN ("Not a Number").

Funciones de potencia y logarítmicas

- ∞ math.exp(x): Devuelve ex.
- ∞ math.log(x, [base]): Devuelve el logaritmo neperiano de x. Si se incluye el segundo argumento, devuelve el logaritmo de x en la base indicada.
- ∞ math.log2(x): Devuelve el logaritmo en base 2 de x.

```
math.log(50, 2)
```

5.643856189774724

- ∞ math.log10(x): Devuelve el logaritmo en base 10 de x.
- ∞ math.pow(x, y): Devuelve xy.
- ∞ math.sqrt(x): Devuelve la raíz cuadrada de x.

Funciones trigonométricas y de conversión de ángulos

- ∞ math.cos(x): Devuelve el coseno de x.
- ∞ math.sin(x): Devuelve el seno de x.
- ∞ math.tan(x): Devuelve la tangente de x.
- ∞ math.degrees(x): Convierte un ángulo de grados sexagesimales a radianes.
- ∞ math.radians(x): Convierte un ángulo de radianes a grados sexagesimales.

Constantes

- ∞ math.pi: Número pi.
- ∞ math.e: Número e:

```
math.e
```

2.718281828459045

- ∞ math.nan: Valor equivalente a "no es un número".

<https://docs.python.org/3/library/math.html>

Pandas

Es un paquete de Python que proporciona estructuras de datos similares a los dataframes de R. Pandas depende de Numpy, la librería que añade un potente tipo matricial a Python. Los principales tipos de datos que pueden representarse con pandas son:

- ∞ Datos tabulares con columnas de tipo heterogéneo con etiquetas en columnas y filas.
- ∞ Series temporales.

Pandas proporciona herramientas que permiten:

- ∞ leer y escribir datos en diferentes formatos: CSV, Microsoft Excel, bases SQL y formato HDF5
- ∞ seleccionar y filtrar de manera sencilla tablas de datos en función de posición, valor o etiquetas
- ∞ fusionar y unir datos
- ∞ transformar datos aplicando funciones tanto en global como por ventanas
- ∞ manipulación de series temporales
- ∞ hacer gráficas

En pandas existen tres tipos básicos de objetos todos ellos basados a su vez en Numpy:

- ∞ Series (listas, 1D).
- ∞ DataFrame (tablas, 2D).
- ∞ Panels (tablas 3D).

<https://pandas.pydata.org/pandas-docs/stable/index.html>

NumPy

Es una biblioteca para el lenguaje de programación Python que da soporte para crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas.

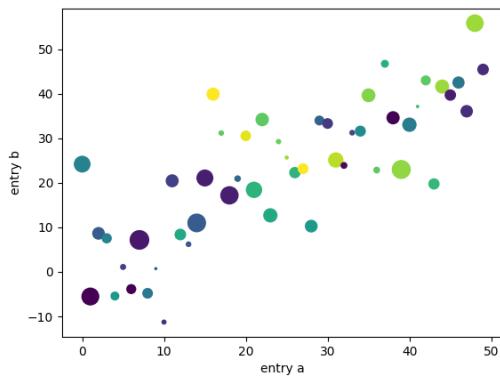
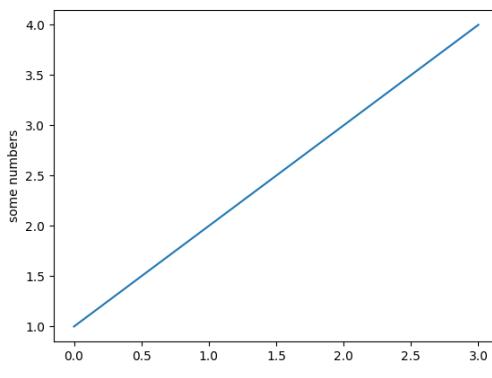
El uso de NumPy en Python brinda una funcionalidad comparable a MATLAB, ya que ambos se interpretan y ambos permiten al usuario escribir programas rápidos siempre que la mayoría de las operaciones funcionen en vectores o matrices en lugar de escalares.

NumPy es un paquete optimizado para trabajar bajo GPUs.

<https://numpy.org/doc/stable/>

Mathplotlib

Es una biblioteca de imágenes 2D y 3D que produce figuras de alta calidad en una variedad de formatos impresos y entornos interactivos en todas las plataformas. La librería Mathplotlib se puede utilizar en scripts de Python, el shell de Python y en iPython.



```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt

data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```

<https://matplotlib.org/stable/index.html>

Repasso

```
"""
Tienda vender zapatillas

z --> marca, numero, color, precio, stock
    -- crear, imprimir, modificar
p_c --> nombre, tlf, email, direccion
    -- crear, imprimir, modificar
p_v --> nombre, dni, direccion
    -- crear, imprimir, modificar
compra -> p_c, p_v, zatilla, cantidad
    -- add_linea_compra
    -- crear_factura
    -- pagar
"""
IVA = 1.21

class Zapatillas:
    def __init__(self, marca=None, numero=None, color="Rojo",
precio=0, stock=0):
        self._marca = str(marca).upper()
        self.numero = numero
        self.color = str(color).upper()
        self.precio = precio
        self.stock = stock

    def __str__(self):
        return f"Zapatilla {self._marca} {self.numero}-{self.color})"
```

```
def modificar(self, marca=None, numero=None, color="Rojo",
precio=0, stock=0):
    self._marca = marca if marca is not None else self._marca
    self.numero = numero
    self.color = color
    self.precio = precio
    self.stock = stock

class Persona:
    def __init__(self):
        self.nombre = None
        self.direccion = None

    def __str__(self):
        return f"{self.nombre} ({self.direccion})"

class Comprador(Persona):
    def __init__(self, n, d, t, em="dir@dir.com"):
        super().__init__()
        self.nombre = str(n).upper()
        self.direccion = str(d).upper()
        self.tlf = t
        self.email = str(em).upper()

    def __str__(self):
        return f"{self.nombre} ({self.direccion})- [{self.tlf} - {self.email}]"

    def modificar(self, n, d, t, em):
        self.nombre = n if n is not None else self.nombre
        # validar el resto de campos
        self.direccion = d
        self.tlf = t
        self.email = em

class Vendedor(Persona):
    def __init__(self, n, d, dn):
        super().__init__()
        self.nombre = n
        self.direccion = d
        self.dni = dn

    def __str__(self):
        return f"{self.nombre} ({self.direccion})- [{self.dni}]"

    def modificar(self, n, d, dn):
        self.nombre = n if n is not None else self.nombre
        # validar el resto de campos
        self.direccion = d
        self.dni = dn

class Compra:
    def __init__(self):
```

```
        self._comprador = None
        self._vendedor = None
        self._lineas_compra = [] # Zapatillas

    @property
    def comprador(self):
        return self._comprador

    @comprador.setter
    def comprador(self, c):
        if isinstance(c, Comprador):
            self._comprador = c

    @property
    def vendedor(self):
        return self._vendedor

    @vendedor.setter
    def vendedor(self, v):
        if isinstance(v, Vendedor):
            self._vendedor = v

    def add_linea(self, z):
        if type(z) is Zapatillas:
            self._lineas_compra.append(z)

    @property
    def num_lineas(self):
        return len(self._lineas_compra)

    @property
    def _total_a_pagar(self):
        total = 0
        for linea in self._lineas_compra:
            total += linea.precio * linea.stock
        return total

    def pagar(self):
        pagado = False
        total_a_pagar = self._total_a_pagar * IVA
        # conectarse a tpv
        # esperar respuesta tpv correcta
        # responder en función de respuesta tpv
        # modificar pagado en función de la respuesta
        return pagado, total_a_pagar

    def crear_factura(self):
        texto_factura = "FACTURA \n"
        texto_factura += " N° 32479342\n"
        texto_factura += "=" * 50 + "\n"
        for linea in self._lineas_compra:
            texto_factura += str(linea) + " --> " +
                str(linea.precio * linea.stock) + "\n"
        texto_factura += "-" * 50 + "\n"
        texto_factura += "Total:" + str(self._total_a_pagar)

        return texto_factura
```

```
class App:  
    def main(self):  
        z1 = Zapatillas("adidas", 45, "Azul", 30, 5)  
        print(str(z1).title())  
        c = Comprador("nombre", "dirección", "3456788", "EEEE")  
        print(c)  
        cp = Compra()  
        cp.comprador = c  
        cp.add_linea(z1)  
        print(cp.crear_factura())  
  
App().main()
```

Ejercicios

Apartado A

En cada ejercicio debes crear un programa con dos clases: una clase que solo contendrá la función main, además de otra clase (con sus atributos y métodos) que utilizarás desde el main de la clase principal para hacer pruebas sobre su funcionamiento.

En este apartado las clases solo contendrán atributos (variables) y haremos algunas pruebas sencillas con ellas para entender cómo se instancia objetos y se accede a sus atributos.

Ejercicio A1 – Punto

Crea un programa con una clase llamada Punto que representará un punto de dos dimensiones en un plano. Solo contendrá dos atributos enteros llamadas **x** e **y** (coordenadas).

En el main de la clase principal instancia 3 objetos Punto con las coordenadas (5,0), (10,10) y (-3, 7). Muestra por pantalla sus coordenadas (utiliza un print para cada punto). Modifica todas las coordenadas (prueba distintos operadores como = + - += *=...) y vuelve a imprimirlas por pantalla.

Ejercicio A2 – Persona

Crea un programa con una clase llamada Persona que representará los datos principales de una persona: **dni**, **nombre**, **apellidos** y **edad**.

En el main de la clase principal instancia dos objetos de la clase Persona. Luego, pide por teclado los datos de ambas personas (guárdalos en los objetos). Por último, imprime dos mensajes por pantalla (uno por objeto) con un mensaje del estilo “Azucena Luján García con DNI ... es / no es mayor de edad”.

Ejercicio A3 – Rectángulo

Crea un programa con una clase llamada Rectángulo que representará un rectángulo mediante dos coordenadas (x1, y1) y (x2, y2) en un plano, por lo que la clase deberá tener cuatro atributos enteros: **x1**, **y1**, **x2**, **y2**.

En el main de la clase principal instancia 2 objetos Rectángulo en (0,0)(5,5) y (7,9)(2,3). Muestra por pantalla sus coordenadas, perímetros (suma de lados) y áreas (ancho x alto). Modifica todas las coordenadas como consideres y vuelve a imprimir coordenadas, perímetros y áreas.

Ejercicio A4 – Artículo

Crea un programa con una clase llamada Artículo con los siguientes atributos: **nombre**, **precio** (sin IVA), **iva** (siempre será 21) y **stock** (representa cuantos quedan en el almacén).

En el main de la clase principal instancia un objeto de la clase artículo. Asignales valores a todos sus atributos (los que quieras) y muestra por pantalla un mensaje del estilo “Pijama - Precio: 10€ - IVA: 21% - PVP: 12,1€” (el PVP es el precio de venta al público, es decir, el precio con IVA). Luego, cambia el precio y vuelve a imprimir el mensaje.

Apartado B

En este apartado tienes que modificar los programas del apartado anterior (o haz una copia del proyecto si lo prefieres) y realizar los cambios indicados.

Ejercicio B1 – Punto

Añade a la clase Punto los siguientes métodos públicos:

- **imprime()** // Imprime por pantalla las coordenadas. Ejemplo: "(7, -5)"
- **set_xy(x, y)** // Modifica ambas coordenadas. Es como un setter doble.
- **desplaza(int dx, int dy)** // Desplaza el punto la cantidad (dx, dy) indicada. Ejemplo: Si el punto (1, 1) se desplaza (2, 5) entonces estará en (3, 6).
- **distancia(Punto p)** // Calcula y devuelve la distancia entre el propio objeto y otro objeto (Punto p) que se pasa como parámetro: distancia entre dos coordenadas.

Prueba a utilizar estos métodos desde el main para comprobar su funcionamiento.

Ejercicio B2 – Persona

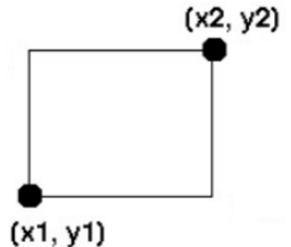
Añade a la clase Persona los siguientes métodos públicos:

- **imprime()** // Imprime la información del objeto: "DNI:... Nombre:... etc."
- **es_mayor_edad()** // Devuelve true si es mayor de edad (false si no).
- **es_jubilado()** // Devuelve true si tiene 65 años o más (false si no).
- **diferencia_edad(Persona p)** // Devuelve la diferencia de edad entre 'this' y p.

Prueba a utilizar estos métodos desde el main para comprobar su funcionamiento.

Ejercicio B3 – Rectángulo

En nuestro software necesitamos asegurarnos de que la coordenada (x1, y1) represente la esquina inferior izquierda y la (x2, y2) la superior derecha del rectángulo, como en el dibujo.



Incluye un **if** que compruebe los valores. Si son válidos guardará los parámetros en el objeto. Si no lo inicializará a al valor de la coordenada inferior.

Añade a la clase Rectángulo métodos públicos con las siguientes funcionalidades:

- Método para **imprimir** la información del rectángulo por pantalla.
- Métodos setters dobles y cuadruples: **set_x1y1**, **set_x2y2** y **set_all(...)**.
- Métodos **get_perímetro** y **get_area** que calculen y devuelvan el perímetro y área del objeto.

Prueba a utilizar estos métodos desde el main para comprobar su funcionamiento.

Ejercicio B4 – Artículo

Añade a la clase Artículo métodos públicos con las siguientes funcionalidades:

- Método para **imprimir** la información del artículo por pantalla.
- Método **get_pvp** que devuelva el precio de venta al público (PVP) con iva incluido.
- Método **get_pvp_descuento** que devuelva el PVP con un descuento pasado como argumento.
- Método **vender** que actualiza los atributos del objeto tras vender una cantidad 'x' (si es posible). Devolverá true si ha sido posible (false en caso contrario).
- Método **almacenar** que actualiza los atributos del objeto tras almacenar una cantidad 'x'. Devolverá true.

Ejercicios de cadenas

- ∞ Escribir por pantalla cada carácter de una cadena introducida por teclado.
- ∞ Realizar un programa que comprueba si una cadena leída por teclado comienza por una subcadena pedida por teclado.
- ∞ Pide una cadena y un carácter por teclado (valida que sea un carácter) y muestra cuantas veces aparece el carácter en la cadena.

- ∞ Suponiendo que hemos introducido una cadena por teclado que representa una frase (palabras separadas por espacios), realiza un programa que cuente cuantas palabras tiene.
- ∞ Si tenemos una cadena con un nombre y apellidos, realizar un programa que muestre las iniciales en mayúsculas.
- ∞ Realizar un programa que, dada una cadena de caracteres por caracteres, genere otra cadena resultado de invertir la primera.
- ∞ Pide una cadena y dos caracteres por teclado (valida que sea un carácter), sustituye la aparición del primer carácter en la cadena por el segundo carácter.
- ∞ Realizar un programa que lea una cadena por teclado y convierta las mayúsculas a minúsculas y viceversa.
- ∞ Realizar un programa que compruebe si una cadena contiene una subcadena. Las dos cadenas se introducen por teclado.
- ∞ Escribir funciones que dada una cadena de caracteres:
 - Imprima los dos primeros caracteres.
 - Imprima los tres últimos caracteres.
 - Imprima dicha cadena cada dos caracteres. Ej.: recta debería imprimir rca
 - Dicha cadena en sentido inverso. Ej.: **hola mundo!** debe imprimir **!odnum aloh**
 - Imprima la cadena en un sentido y en sentido inverso. Ej.: **reflejo** imprime **reflejoojelfer**
- ∞ Escribir funciones que dada una cadena y un carácter:
 - Inserte el carácter entre cada letra de la cadena. Ej.: **separar** , debería devolver **s,e,p,a,r,a,r**
 - Reemplace todos los espacios por el carácter. Ej.: mi archivo de **texto.txt** y **_** debería devolver **mi_archivo_de_texto.txt**
 - Reemplace todos los dígitos en la cadena por el carácter. Ej.: su clave es: 1540 y X debería devolver su clave es: XXXX
 - Inserte el carácter cada 3 dígitos en la cadena. Ej. 2552552550 y . debería devolver 255.255.255.0
- ∞ Escribir una función que reciba una cadena que contiene un largo número entero y devuelva una cadena con el número y las separaciones de miles. Por ejemplo, si recibe 1234567890, debe devolver 1.234.567.890.
- ∞ Escribir una función que, dada una cadena de caracteres, devuelva:
 - La primera letra de cada palabra. Por ejemplo, si recibe Universal Serial Bus debe devolver USB.
 - Dicha cadena con la primera letra de cada palabra en mayúsculas. Por ejemplo, si recibe república argentina debe devolver Repùblica Argentina.
 - Las palabras que comienzan con la letra A. Por ejemplo, si recibe Antes de ayer debe devolver Antes ayer.
- ∞ Escribir funciones que dada una cadena de caracteres:
 - Devuelva solamente las letras consonantes. Por ejemplo, si recibe algoritmos o logaritmos debe devolver lgrtms.
 - Devuelva solamente las letras vocales. Por ejemplo, si recibe: **sin consonantes** debe devolver i ooae.
 - Reemplace cada vocal por su siguiente vocal. Por ejemplo, si recibe **vestuario** debe devolver vistaerou.
 - Indique si se trata de un palíndromo. Por ejemplo, **anita lava la tina** es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda).
 - Escribir funciones que devuelva la que sea anterior en orden alfáabetico. Por ejemplo, si recibe **kde y gnome** debe devolver **gnome**.
- ∞ Escribir una función que reciba una cadena de unos y ceros (es decir, un número en representación binaria) y devuelva el valor decimal correspondiente.

Caso Práctico Dawbank

La empresa *LibreCoders* te ha contratado para desarrollar un software de gestión de una cuenta bancaria para la cooperativa de banca ética y sostenible *DawBank*. Se trata de una aplicación Java formada por una clase principal *DawBank* y otra llamada *CuentaBancaria*.

El programa pedirá los datos necesarios para crear una cuenta bancaria. Si son válidos, creará la cuenta y mostrará el menú principal para permitir actuar sobre la cuenta. Tras cada acción se volverá a mostrar el menú.

1. **Datos de la cuenta.** Mostrará el IBAN, el titular y el saldo.
2. **IBAN.** Mostrará el IBAN.
3. **Titular.** Mostrará el titular.
4. **Saldo.** Mostrará el saldo disponible.
5. **Ingreso.** Pedirá la cantidad a ingresar y realizará el ingreso si es posible.
6. **Retirada.** Pedirá la cantidad a retirar y realizará la retirada si es posible.
7. **Movimientos.** Mostrará una lista con el historial de movimientos.
8. **Salir.** Termina el programa.

Clase CuentaBancaria

Una cuenta bancaria tiene como datos asociados el iban (international bank account number, formado por dos letras y 22 números, por ejemplo, ES6621000418401234567891), el titular (un nombre completo), el saldo (dinero en euros) y los movimientos (histórico de los movimientos realizados en la cuenta, un máximo de 100^(*) para simplificar).

Cuando se crea una cuenta es obligatorio que tenga un iban y un titular (que no podrán cambiar nunca). El saldo será de 0 euros y la cuenta no tendrá movimientos asociados.

El saldo solo puede variar cuando se produce un ingreso (entra dinero en la cuenta) o una retirada (sale dinero de la cuenta). En ambos casos se deberá registrar la operación en los movimientos. Los ingresos y retiradas solo pueden ser de valores superiores a cero.

El saldo de una cuenta nunca podrá ser inferior a -50^(*) euros. Si se produce un movimiento que deje la cuenta con un saldo negativo (no inferior a -50) habrá que mostrar el mensaje “AVISO: Saldo negativo”. Si se produce un movimiento superior a 3.000^(*) euros se mostrará el mensaje “AVISO: Notificar a hacienda”.

No se realizará ningún tipo de entrada por teclado. La única salida por pantalla permitida son los dos mensajes de aviso mencionados arriba, ninguna otra.

() Estos valores no pueden variar y son iguales para todas las cuentas bancarias.*

Clase DawBank

Clase principal con función main. Encargada de interactuar con el usuario, mostrar el menú principal, dar feedback y/o mensajes de error, etc. Utilizará la clase CuentaBancaria. Puedes implementar las funciones que consideres oportunas.

U. T. 6 POO (II).



Tipos avanzados: Secuencias

Una secuencia es un contenedor de objetos de cualquier tipo con un orden asociado, es el concepto de array indexado tradicional de lenguajes como C. Como mantiene el orden se puede insertar el mismo objeto varias veces. Dentro de las secuencias tenemos dos tipos: inmutables (tuplas) y que se pueden modificar (listas).

Las tuplas y las listas mantienen un gran número de métodos en común y el método de acceso, siendo este un índice siempre entre corchetes. Del mismo modo se pueden crear secuencias multidimensionales reutilizando la sintaxis de la definición.

Toda secuencia incorpora el método **index**, que devuelve el índice del objeto y el método **count** que recogerá el número de ocurrencias de un objeto.

Podemos comparar dos secuencias con los operadores tradicionales: **==**, **!=**, **in**, **len**, **del** Secuencia[index], etc. Además, podemos usar los operadores más (+) y repetición (*). También se pueden usar las siguientes funciones del lenguaje: **reverse**, **max**, **min**, **sum**, **all** (todos verdaderos), **any** (uno al menos verdadero) y **not** así como el módulo **itertools** para iteraciones complejas.

Por último, si queremos aplicar una función a toda la secuencia podemos usar las funciones **zip** (une dos secuencias del mismo tipo), **filter** (aplica una función filtro a cada elemento de la lista), **map** (aplica una función a cada elemento de la lista) y **enumerate** (devuelve un iterador donde cada elemento es una tupla del tipo (índice, valor)).

```
l = [1, 2, 3, 4]
for ele in l:
    if ele == 1:
        del l[1] #Qué pasa al borrar dentro de un bucle
    print(ele)

l = [1, 2, 3, 4]
def f(a):
    return a+2
def fil(a):
    if a > 5:
        return False
    else:
        return True
for ele in map(f, l):
    print(ele)
for ele in zip([1, 2, 3], ["a", "b", "c", "d"]):
    print(ele)
for ele in filter(fil, range(1, 10)):
    print(ele)
```

Acceso a las secuencias

El acceso a los elementos de las secuencias es mediante el operador corchetes, siendo posible la extracción de una parte de la misma usando el operador dos puntos (:) similar a las cadenas, incluso los índices negativos. También podemos recorrer una secuencia mediante un bucle **for** (en el ejemplo recogemos tanto los valores como los índices a través de la función **enumerate**).

```
for index, valor in enumerate(secuencia):
    print(index, valor)
```

Tipos avanzados: Listas

Las listas son un tipo secuencia mutable (se puede cambiar) que guardan el orden de inserción, usadas normalmente para almacenar colecciones de objetos homogéneos pero que pueden almacenar elementos heterogéneos, que utilizan un índice como acceso que empieza en cero. Las listas se pueden construir de diferentes formas:

- ∞ Usando un par de corchetes para definir una lista vacía: []
- ∞ Usando corchetes, separando los elementos incluidos con comas: [a], [a, b, c]
- ∞ Usando una lista intensiva o por comprensión: [x for x in iterable]
- ∞ Usando el constructor de tipo: **list()** o **list(iterable)**

```
l = []
l = [1, 2, 3, 4]
l = [x//2 for x in range(5)]
l = list("abc") # produce tres elementos en la lista, a, b y c
l = ["abc"] # produce un elemento en la lista: abc
```

Operaciones

Operación	Resultado
s[i] = x	el elemento <i>i</i> de <i>s</i> es reemplazado por <i>x</i>
s[i:j] = t	la rebanada de valores de <i>s</i> que van de <i>i</i> a <i>j</i> es reemplazada por el contenido del iterador <i>t</i>
del s[i:j]	equivalente a <i>s[i:j] = []</i>
s[i:j:k] = t	los elementos de <i>s[i:j:k]</i> son reemplazados por los elementos de <i>t</i>
del s[i:j:k]	borra los elementos de <i>s[i:j:k]</i> de la lista
s.append(x)	añade <i>x</i> al final de la secuencia (Equivale a <i>s[len(s):len(s)] = [x]</i>)
s.clear()	elimina todos los elementos de <i>s</i> (Equivale a <i>del s[:]</i>)
s.copy()	crea una copia superficial de <i>s</i> (Equivale a <i>s[:]</i>)
s.extend(t) o s += t	extiende <i>s</i> con los contenidos de <i>t</i> (En la mayoría de los casos equivale a <i>s[len(s):len(s)] = t</i>)
s *= n	actualiza <i>s</i> con su contenido repetido <i>n</i> veces
s.insert(i, x)	inserta <i>x</i> en <i>s</i> en la posición indicada por el índice <i>i</i> (Equivale a <i>s[i:i] = [x]</i>)
s.pop(i)	retorna el elemento en la posición indicada por <i>i</i> , y a la vez lo elimina de la secuencia <i>s</i>
s.remove(x)	elimina el primer elemento de <i>s</i> tal que <i>s[i]</i> sea igual a <i>x</i>
s.reverse()	invierte el orden de los elementos de <i>s</i> , a nivel interno
s	True si un elemento de <i>s</i> es igual a <i>x</i> , False en caso contrario
x not in s	False si un elemento de <i>s</i> es igual a <i>x</i> , True en caso contrario
s + t	la concatenación de <i>s</i> y <i>t</i>
s * n o n * s	equivale a concatenar <i>s</i> consigo mismo <i>n</i> veces
s[i]	El elemento <i>i</i> -ésimo de <i>s</i> , empezando a contar en 0
s[i:j]	la rebanada de <i>s</i> desde <i>i</i> hasta <i>j</i>
s[i:j:k]	la rebanada de <i>s</i> desde <i>i</i> hasta <i>j</i> , con paso <i>j</i>

len(s)	longitud de s
min(s)	el elemento más pequeño de s
max(s)	el elemento más grande de s
s.index(x[, i[, j]])	índice de la primera ocurrencia de x en s (en la posición i o superior, y antes de j)
s.count(x)	número total de ocurrencias de x en s
sort(*, key=None, reverse=False)	Este método ordena la lista <i>in situ</i> (se modifica internamente), usando únicamente comparaciones de tipo <.

```

fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple',
'banana']
fruits.count('apple')
fruits.count('tangerine')
fruits.index('banana')
fruits.index('banana', 4) # Find next banana starting 4
fruits.reverse()
fruits.append('grape')
fruits.sort()
fruits.pop()

```

Borrado de elementos

Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción **del**. Esta es diferente del método **pop()**, el cual retorna un valor. La instrucción **del** también puede usarse para quitar secciones de una lista o vaciar la lista completa (lo que hacíamos antes asignando una lista vacía a la sección).

```

a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0] # [1, 66.25, 333, 333, 1234.5]
del a[2:4] # [1, 66.25, 1234.5]
del a[:] # []

```

del puede usarse también para eliminar variables:

```
del a
```

Hacer referencia al nombre **a** de aquí en adelante es un error (al menos hasta que se le asigne otro valor).

Ordenación

La ordenación de listas es una tarea fundamental ya que, aunque guardan el orden de inserción, no significa que sus elementos estén ordenados. Las listas de Python tienen un método incorporado **list.sort()** que modifica la lista internamente ordenándola. También hay una función incorporada **sorted()** que crea una nueva lista ordenada a partir de otra secuencia dejando sin modificar la inicial.

```

a = [5, 2, 3, 1, 4]
print(sorted(a))
print(a)
a.sort()
print(a)

```

Tanto **list.sort()** como **sorted()** tienen un parámetro clave (**key**) para indicar una función a la que llamar para comparar dos elementos y poder ordenarlos.

Por ejemplo, aquí hay una comparación de cadenas que no distingue entre mayúsculas y minúsculas:

```

sorted("This is a test string from Andrew".split(),
      key=str.lower)
# ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']

```

El valor del parámetro **key** tienen que ser una función que recoge un único argumento y devuelve una clave para realizar las comparaciones. Un uso frecuente es ordenar objetos complejos utilizando algunos de los índices del objeto como claves.

```
student_tuples = [
    ('john', 'A', 15),
    ('jane', 'B', 12),
    ('dave', 'B', 10),
]
sorted(student_tuples, key=lambda student: student[2])
# sort by age, index 2
# [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Los métodos **list.sort()** y **sorted()** aceptan un parámetro **reverse** con un valor booleano. Este valor se usa para hacer ordenaciones descendentes. Por ejemplo, para obtener los datos de los estudiantes en orden inverso de edad.

```
sorted(student_tuples, key=lambda student: student[2],
       reverse=True)
```

Para más información sobre métodos de ordenación complejos visitar la siguiente dirección

<https://docs.python.org/es/3/howto/sorting.html#sortinghowto>

Pasar Listas como Parámetros

Como ya hemos visto en Python se puede pasar cualquier tipo de datos en un parámetro, con lo que el uso de las listas como parámetros es muy sencillo. Además, las listas son tipos complejos con lo que el paso se hace por referencia, lo que significa que podremos modificar el contenido de la lista dentro de un método o función sin tener que recurrir a la variable **global**, pero no podremos crearla o destruirla.

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

def mi_funcion(p):
    if type(p) == list and len(p) > 0:
        p[0] = "cambiado"
    del p # no hace nada fuera

print(a) # [5, 2, 3, 1, 4]
mi_funcion(a)
print(a) # ['cambiado', 2, 3, 1, 4]
```

Listas multidimensionales

Las listas no representan el concepto de Array de otros lenguajes, en donde se definen las dimensiones de forma fija. Para ese funcionamiento hay que utilizar el tipo **numpy.array**

Como todos los tipos de Python, las listas pueden contener cualquier otro tipo de dato, inclusive otra lista, lo que hace que podamos crear listas de varias dimensiones. La ventaja de las listas al usarlas en varias dimensiones es que no tienen la limitación que tienen los Arrays, cada dimensión puede tener la longitud que necesitemos. Así, podremos tener una lista de dos dimensiones (una tabla) pero cada fila podrá contener diferente cantidad de elementos.

```
lista = [[1, 2, 3], # Array tradicional
         [4, 5, 6]]
```

```

print(lista)
print(lista[1]) # acceso a la línea dos
print(lista[1][0]) # línea dos primer elemento

lista = [[1, 2, 3], # Lista multidimensional
         [4, 5, 6, 7],
         [8]]
print(lista[2][0])

```

El principal problema que nos presentan las listas multidimensionales es precisamente el hecho de que cada elemento en la misma dimensión puede contener un número diferente de objetos. Este problema es mínimo al recordar que el bucle **for** de Python itera sobre listas de forma innata.

```

lista = [[1, 2, 3],
         [4, 5, 6, 7],
         [8]]

for linea in lista:
    for elemento in linea:
        print("El elemento es:", elemento)

```

Otros tipos avanzados

Tuplas

Las tuplas son un tipo de secuencias inmutables, ordenadas, que se acceden por índice empezando en cero. La definición de una tupla se lleva a cabo con los paréntesis o el constructor: **tuple()**. Debemos conocer la tupla vacía **()**.

```

tupla = (1, 3, 5, 7)
for index, valor in enumerate(tupla):
    print(index, valor)

```

Los métodos son los mismos que las lisas exceptuando aquellos que permiten modificación de la misma: asignación (=), añadir (append), etc. Funcionan igual que las listas excepto que son **inmutables**, este debe ser el tipo elegido para datos que no cambien nunca ya que son muy eficientes.

Diccionarios

Un objeto de tipo **mapping** relaciona valores (que deben ser **hashable** y por tanto inmutables las claves) con objetos de cualquier tipo (Arrays asociativos en Php). Los mapas son objetos mutables. En este momento solo hay un tipo estándar de mapa, los diccionarios. Los diccionarios se pueden crear usando:

- ∞ una lista separada por comas de pares key: value entre llaves, por ejemplo: {'jack': 4098, 'sjoerd': 4127} o {4098: 'jack', 4127: 'sjoerd'}
- ∞ Un diccionario de compresión {}, {x: x ** 2 for x in range(10)}
- ∞ Use el constructor: dict(), dict([('foo', 100), ('bar', 200)]), dict(foo=100, bar=200)
- ∞ El diccionario vacío: {}

Los siguientes ejemplos retornan el mismo diccionario {"one": 1, "two": 2, "three": 3}.

```

a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
f = dict({'one': 1, 'three': 3}, two=2)
a == b == c == d == e == f

```

Operaciones

Operación	Resultado
<code>s[i] = x</code>	el elemento <i>i</i> de <i>s</i> es reemplazado por <i>x</i> , <i>si no está se añade</i>
<code>del s[i]</code>	borra la clave <i>i</i>
<code>s.clear()</code>	elimina todos los elementos de <i>s</i>
<code>s.copy()</code>	crea una copia de <i>s</i>
<code>s.pop(key, default)</code>	retorna el elemento clave indicada por <i>i</i> , y a la vez lo elimina de la secuencia <i>s</i> , <i>si no lo encuentra devuleve default</i>
<code>s.reversed()</code>	Retorna invirtiendo el orden de los elementos de <i>s</i>
<code>key in s</code>	True si <i>key</i> está en <i>s</i>
<code>key not in s</code>	False si <i>key</i> no está en <i>s</i>
<code>len(s)</code>	longitud de <i>s</i>
<code>get(key[, default])</code>	Retorna el elemento dentro de <i>d</i> almacenado bajo la clave <i>key</i> , si <i>key</i> está en el diccionario; si no, retorna <i>default</i> .
<code>items()</code>	Retorna una nueva vista de los contenidos del diccionario (Pares (<i>key, value</i>)).
<code>keys()</code>	Retorna una nueva vista de las claves del diccionario
<code>popitem()</code>	Elimina y retorna una pareja (<i>key, value</i>) del diccionario. Las parejas se retornan en el orden LIFO
<code>setdefault(key[, default])</code>	Si <i>key</i> está incluida en el diccionario, retorna el valor almacenado. Si no, inserta con la clave <i>key</i> el valor definido en <i>default</i> y retorna <i>default</i>
<code>update([other])</code>	Actualiza el diccionario con las parejas clave/valor obtenidas de <i>other</i> , escribiendo encima de las claves existentes
<code>values()</code>	Retorna una nueva vista de los valores del diccionario
<code>d other</code>	Retorna la unión de <i>d</i> y <i>other</i>
<code>d = other</code>	Actualiza <i>d</i> con la unión de <i>d</i> y <i>other</i>

Los objetos retornados por los métodos `dict.keys()`, `dict.values()` y `dict.items()` son objetos tipo vista o view. Estos objetos proporcionan una vista dinámica del contenido del diccionario, lo que significa que, si el diccionario cambia las vistas reflejan estos cambios. Las vistas de un diccionario pueden ser iteradas para retornar sus datos respectivos y soportan operaciones de comprobación de pertenencia.

Crea un programa que cuente de forma indefinida las palabras introducidas en una frase pedida al usuario

```

palabras = {}
while True:
    frase = input("¿Introduce un texto?")
    if len(frase):
        for palabra in frase.lower().strip().split(" "):
            if palabra not in palabras:
                palabras[palabra] = 1
            else:
                palabras[palabra] += 1
print(palabras)

```

Conjuntos

Python también incluye un tipo de dato para conjuntos (**set**). Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de

entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia y diferencia simétrica.

Las llaves o la función **set()** pueden usarse para crear conjuntos, pero las llaves no se pueden usar para crear un conjunto vacío, en ese caso se crea un diccionario vacío, hay que usar el constructor. Una pequeña demostración.

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket) # show that duplicates have been removed
# {'orange', 'banana', 'pear', 'apple'}
'orange' in basket # fast membership testing
'crabgrass' in basket
```

```
a = set('abracadabra')
b = set('alacazam')
a # unique letters in a
# {'a', 'r', 'b', 'c', 'd'}
a - b # letters in a but not in b
# {'r', 'd', 'b'}
a | b # letters in a or b or both
# {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
a & b # letters in both a and b
# {'a', 'c'}
a ^ b # letters in a or b but not both
# {'r', 'd', 'b', 'm', 'z', 'l'}
```

Comprensión de listas y diccionarios

Las comprensiones ofrecen una manera concisa de crear listas, tuplas o diccionarios. Sus usos comunes son para hacer nuevas listas donde cada elemento es el resultado de algunas operaciones aplicadas a cada miembro de otra secuencia o iterable, o para crear un segmento de la secuencia de esos elementos para satisfacer una condición determinada. Por ejemplo, asumamos que queremos crear una lista de cuadrados.

```
squares = []
for x in range(10):
    squares.append(x**2)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Podemos calcular la lista de cuadrados sin ningún efecto secundario haciendo:

```
squares = list(map(lambda x: x**2, range(10)))
squares = [x**2 for x in range(10)]
```

Una lista de comprensión consiste en corchetes rodeando una expresión seguida de la declaración **for** y luego cero o más declaraciones **for** o **if**. El resultado será una nueva lista que sale de evaluar la expresión en el contexto de los **for** o **if** que le siguen. Por ejemplo, esta lista de comprensión combina los elementos de dos listas si no son iguales:

```
[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
# [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

combs = []
for x in [1, 2, 3]:
    for y in [3, 1, 4]:
        if x != y:
```

```
combs.append((x, y))
```

Si la expresión es una tupla (como en `(x, y)` en el ejemplo anterior), debe estar entre paréntesis.

```
vec = [-4, -2, 0, 2, 4]
# create a new list with the values doubled
[x*2 for x in vec] #[-8, -4, 0, 4, 8]
# filter the list to exclude negative numbers
[x for x in vec if x >= 0] #[0, 2, 4]
# apply a function to all the elements
[abs(x) for x in vec] #[4, 2, 0, 2, 4]
# call a method on each element
freshfruit = [' banana', ' loganberry ', 'passion fruit  ']
[weapon.strip() for weapon in freshfruit]
# ['banana', 'loganberry', 'passion fruit']
# create a list of 2-tuples like (number, square)
[(x, x**2) for x in range(6)]
# [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
# the tuple must be parenthesized, otherwise an error is raised
[x, x**2 for x in range(6)]
File "<stdin>", line 1, in <module>
[x, x**2 for x in range(6)]
```

Las comprensiones pueden contener expresiones complejas y funciones anidadas.

```
from math import pi
[str(round(pi, i)) for i in range(1, 6)]
# ['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Recursividad

Introducción

Las funciones recursivas son funciones que se llaman a sí mismas durante su propia ejecución. Funcionan de forma similar a las iteraciones, pero debe encargarse de planificar el momento en que dejan de llamarse a sí mismas o tendrá una función recursiva infinita.

Las llamadas recursivas suelen ser muy útiles en casos muy puntuales, pero debido a su gran factibilidad de caer en iteraciones infinitas, deben extremarse las medidas preventivas adecuadas y, solo utilizarse cuando sea estrictamente necesario y no exista una forma alternativa viable, que resuelva el problema evitando la recursividad.

```
def jugar(intento=1):
    respuesta = input("¿De qué color es una naranja? ")
    if respuesta != "naranja":
        if intento < 3:
            print("\nFallaste! Inténtalo de nuevo")
            intento += 1
            jugar(intento) # Llamada recursiva
        else:
            print("\nPerdiste!")
    else:
        print("\nGanaste!")
jugar()
```

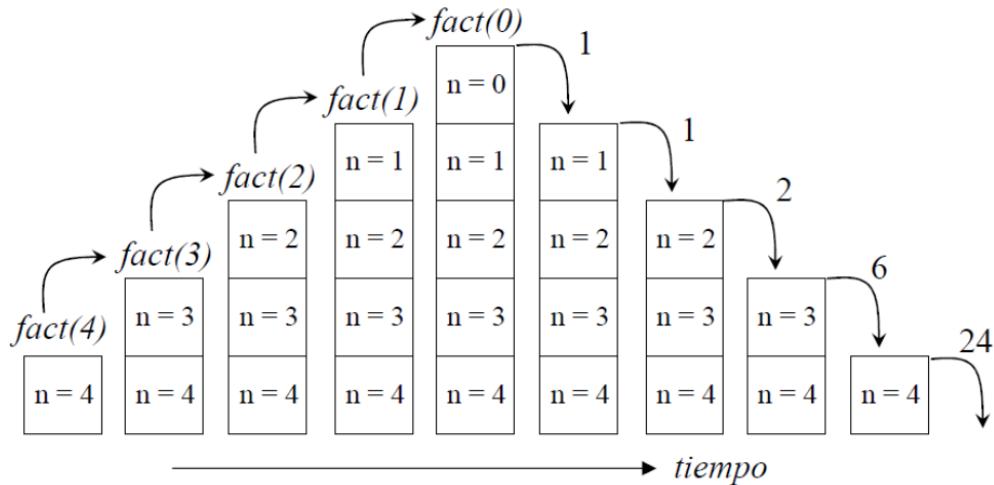
```
def factorial(num):
    print("Valor inicial ->", num)
```

```

if num > 1:
    num = num * factorial(num - 1)
print("valor final ->" ,num)
return num

print(factorial(5))

```



Hacer un buen uso de la recursividad

- ∞ Para simplificar el código.
- ∞ Cuando la estructura de datos es recursiva ejemplo: árboles.
- ∞ Cuando no exista una solución iterativa simple.

Ventajas y Desventajas

Ventajas

- ∞ Son más cercanos a la descripción matemática.
- ∞ Generalmente más fáciles de analizar
- ∞ Se adaptan mejor a las estructuras de datos recursivas.
- ∞ Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

Desventajas

- ∞ La recursividad consume mucha memoria y tiempo de ejecución.
- ∞ La recursividad puede dar lugar a la redundancia (resolver el mismo problema más de una vez)

Ejemplos

Desarrollar la sucesión de Fibonacci

[Code_0005.py](#)

Desarrollar el problema de las torres de Hanoi

[Code_0005.py](#)

Herencia

Introducción

La herencia es la capacidad que tienen las clases de usar las características ya declaradas en otras clases para su propia declaración. La herencia genera una relación de padres e hijos en las clases, una jerarquía, en la que las clases que heredan se sitúan en la parte inferior de la jerarquía, y las clases de las que se heredan en la parte superior.

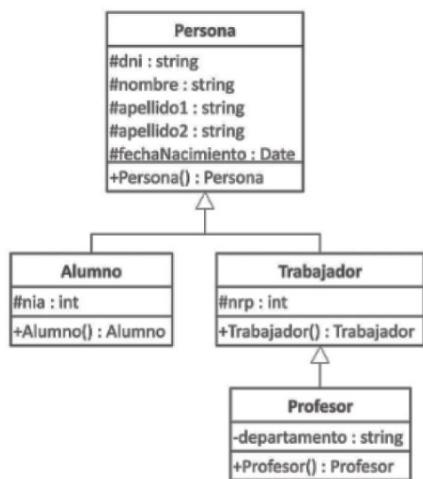
En la herencia influye la visibilidad de las propiedades y métodos, recordemos que puede ser pública, privada o protegida. En el primer caso la heredará siempre, en el segundo nunca y en el tercero también se heredará, pero para uso interno nada más de la clase hija.

Una vez explicada la herencia, tenemos que recordar que bajo Python todos los métodos y propiedades son públicos, no existe ningún otro tipo de visibilidad, solo son sugerencias (Protegida: _ y Privada: __).

Dentro de los tipos de herencia simple o múltiple que se puede implementar, Python eligió el modelo múltiple en contra de la mayoría de lenguajes POO. Este modelo impone muchos problemas en el mecanismo de inicialización desde las clases hijas, pero es mucho más flexible.

Sintaxis

Para crear una clase que herede de una o varias otras, no hay más que añadir a la definición de la clase unos paréntesis y separar por comas todas las clases desde las que hereda.



```

class Persona:
    def __init__(self):
        self.dni = 0
        self.nombre = ""
        self.apellido_1 = ""
        self.apellido_2 = ""
        self.fecha_nacimiento = ""

class Alumno(Persona):
    def __init__(self):
        super().__init__()
        self.nia = -1

class Trabajador(Persona):
    def __init__(self):
        super().__init__()
        self.nrp = -1

class Profesor(Trabajador):
    def __init__(self):
        super().__init__()
        self.departamento = ""

profesor_1 = Profesor()
profesor_1.nombre = "César"
profesor_1.nrp = 1
profesor_1.departamento = "Informática"
print(profesor_1.nombre, profesor_1.nrp,
      profesor_1.departamento)
  
```

```
class Punto:  
    def __init__(self, x: int = 0, y: int = 0):  
        self.x = x  
        self.y = y  
  
    def __str__(self):  
        return "({},{})".format(self.x, self.y)  
  
    @property  
    def mi_punto(self):  
        return "Punto"  
  
class Circulo:  
    def __init__(self, r: int = 0):  
        self.r = r  
  
    def __str__(self):  
        return "{}".format(self.r)  
  
    @property  
    def mi_circulo(self):  
        return "Circulo"  
  
class PuntoCirculo(Punto, Circulo):  
    def __init__(self, x: int = 0, y: int = 0, r: int = 0):  
        # inicialización clases padre.  
        # no se puede usar super al ser varias clases  
        Punto.__init__(self, x, y)  
        # solo aquí hay que pasar el self  
        Circulo.__init__(self, r)  
  
    # sobrescritura de un método heredado  
    def __str__(self):  
        return "({},{}, {})".format(self.x, self.y, self.r)  
  
    @property  
    def mi_punto_circulo(self):  
        return "PuntoCirculo"  
  
print(Punto(2, 3))  
print(Circulo(4))  
print(PuntoCirculo(5, 6, 7))  
print(  
    PuntoCirculo().mi_punto,  
    PuntoCirculo().mi_circulo,  
    PuntoCirculo().mi_punto_circulo)
```

Independientemente del tipo de herencia, es muy importante inicializar al padre siempre antes de empezar con nuestras propias inicializaciones en el método `__init__`, así nos aseguramos que todas las estructuras de datos están creadas.

```
class Punto:  
    def __init__(self, x: int = 0, y: int = 0):  
        self.__x = x  
        self.__y = y  
  
    def __str__(self):  
        return "({},{})".format(self.__x, self.__y)  
  
    @property  
    def mi_punto(self):  
        return "Punto"  
  
  
class Circulo:  
    def __init__(self, r: int = 0):  
        self._r = r  
  
    def __str__(self):  
        return "{}".format(self._r)  
  
    @property  
    def mi_circulo(self):  
        return "Circulo"  
  
  
class PuntoCirculo(Punto, Circulo):  
    def __init__(self, x: int = 0, y: int = 0, r: int = 0):  
        # inicialización clases padre.  
        # no se puede usar super al ser varias  
        Punto.__init__(self, x, y)  
        # solo aquí hay que pasar el self  
        Circulo.__init__(self, r)  
  
    # sobrescritura de un método heredado  
    def __str__(self):  
        return "({},{},{})".format(self._Punto__x,  
                                  self._Punto__y, self._r)  
  
    @property  
    def mi_punto_circulo(self):  
        return "PuntoCirculo"  
  
  
print(Punto(2, 3))  
print(Circulo(4))  
print(PuntoCirculo(5, 6, 7))  
print(  
    PuntoCirculo().mi_punto,  
    PuntoCirculo().mi_circulo,  
    PuntoCirculo().mi_punto_circulo)
```

Misma versión de las clases anteriores, pero intentando utilizar propiedades “protegidas” y “privadas” bajo Python, como se puede ver no existe ningún tipo de control.

Ejemplo

Desarrolla

- ∞ Escriba una clase **Multimedia** para almacenar información de objetos de tipo multimedia (películas, discos, mp3...). Esta clase contiene título, autor, formato y duración como atributos. El formato puede ser uno de los siguientes: wav, mp3, midi, avi, mov y dvd. El valor de todos los atributos se pasa por parámetro en el momento de crear el objeto. Esta clase tiene, además, un método para devolver cada uno de los atributos y un método `__str__` que devuelve la información del objeto.
- ∞ Escriba una clase **Pelicula** que herede de la clase Multimedia anterior. La clase Película tiene, además de los atributos heredados, un actor principal y una actriz principal. Se permite que uno de los dos sea nulo, pero no los dos. La clase debe tener dos métodos para obtener los dos nuevos atributos y debe sobrescribir el método `__str__` para que devuelva, además de la información heredada, la nueva información.
- ∞ Escriba una clase **Disco** que herede de la clase Multimedia ya realizada. La clase Disco tiene, además de los elementos heredados, un atributo para almacenar el género al que pertenece (rock, pop, dance, etc.). La clase debe tener un método para obtener el nuevo atributo y debe sobrescribir el método `__str__` para que devuelva, además de la información heredada, la nueva información.
- ∞ Escriba una clase **MultimediaMain** que cree dos objetos Pelicula y los muestre por pantalla y después pida datos al usuario para crear dos objetos disco y mostrarlos por pantalla.

[code_0006.py](#)

Sobreescritura de métodos heredados

La herencia permite compartir código entre clases, pero hay ocasiones en que dicho código no se ajusta al comportamiento esperado por la clase hija. En aquellos casos en los que sea esta la situación, podemos sobrescribir cualquier método de la clase padre simplemente rescribiéndolo, como en el ejemplo siguiente la clase **Punto3D** sobrescribe todos los de la clase padre.

```
from math import sqrt

class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x, self.y)

    def distancia(self, punto):
        distancia = None
        if isinstance(punto, Punto):
            distancia = sqrt((punto.x - self.x)**2 +
                             (punto.y - self.y)**2)
        return distancia

class Punto3D(Punto):
```

```

def __init__(self, x=0, y=0, z=0):
    super().__init__(x, y) # inicialización del padre
    self.z = 0

def __str__(self):
    return "({},{},{}).format(self.x, self.y, self.y)

def distancia(self, punto):
    distancia = None
    if isinstance(punto, Punto3D):
        distancia = sqrt((punto.x - self.x) ** 2 +
                         (punto.y - self.y) ** 2 +
                         (punto.z - self.z) ** 2)
    return distancia

```

En ocasiones, necesitaremos acceder a los cálculos o procesos que hace la clase padre desde los métodos que estamos sobrescribiendo. Para invocar a un método de la clase padre se usa la nomenclatura **super().nombre_método()**, facilitándole los parámetros que sean necesarios, no está restringido exclusivamente al constructor, se puede utilizar desde cualquier método de la clase hija.

Es muy importante llamar siempre al inicializador de la clase padre al principio del de la clase hija para crear todas las estructuras necesarias en la jerarquía. Todos aquellos métodos no sobrescritos serán accesibles de forma normal (`self`) a través de la herencia.

Sobrecarga de Métodos

Los lenguajes fuertemente tipados obligan que los parámetros tengan un único tipo. Esta restricción hace que, si definimos un método con una firma (tipos de parámetros y número de ellos junto al valor devuelto) y necesitamos utilizarlo para otra firma diferente, tengamos que crear dos métodos distintos.

```

Function sumar_enteros(x:int, y:int):int
Function sumar_cadenas(x:string, y:string): int

```

Para minimizar estas definiciones, la POO permite que el nombre del método sea el mismo en estos casos siempre que la firma sea diferente y se determinará en tiempo de ejecución, cuando se sepan los tipos con los que se llama al método, cuál usar.

```

Function sumar(x:int, y:int):int
Function sumar(x:string, y:string): int
Function sumar(x:int, y:string): int
Function sumar(x:string, y:int): int

```

Por contra, los lenguajes débilmente tipados como Python, no necesitan este mecanismo, ya que los parámetros aceptarán cualquier tipo de entrada. Esta ventaja en la definición, también implica una desventaja, el no saber el tipo del parámetro con lo que nos obliga a realizar conversiones que antes no hacían falta.

```

Function sumar(x, y)
    Si x es cadena x = convertir_entero(x)
    Si y es cadena y = convertir_entero(y)
    Si x es entero and y es entero
        Devolver x + y
    Sino
        Devolver error

```

Métodos especiales o mágicos.

Bajo Python se denominan métodos mágicos a todos aquellos que son llamados de forma automática por parte del sistema en respuesta a algún evento o necesidades de ejecución. Por ejemplo, al crear un objeto:

`__init__`, etc. Estos métodos empiezan y terminan sus nombres con dos subrayados (`__`). A continuación mostramos una pequeña selección.

Queremos...	Escribimos...	Y se ejecuta...
Iniciar un objeto	<code>x = MyClass()</code>	<code>x.__init__()</code>
Representación “official” en formato cadena	<code>repr(x)</code>	<code>x.__repr__()</code>
Usado por las funciones printf y str	<code>str(x)</code>	<code>x.__str__()</code>
Para iterar: devuelve el iterador	<code>iter(seq)</code>	<code>seq.__iter__()</code>
Para iterar: siguiente valor	<code>next(seq)</code>	<code>seq.__next__()</code>
Número de elementos	<code>len(s)</code>	<code>s.__len__()</code>
Pertenencia al conjunto	<code>x in s</code>	<code>s.__contains__(x)</code>
Recoger un atributo existente	<code>x[key]</code>	<code>x.__getitem__(key)</code>
Establecer un atributo existente	<code>x[key] = value</code>	<code>x.__setitem__(key, value)</code>
Borrar un atributo	<code>del x[key]</code>	<code>x.__delitem__(key)</code>
Suma	<code>x + y</code>	<code>x.__add__(y)</code>
Resta	<code>x - y</code>	<code>x.__sub__(y)</code>
Multiplicación	<code>x * y</code>	<code>x.__mul__(y)</code>
División	<code>x / y</code>	<code>x.__truediv__(y)</code>
División entera	<code>x // y</code>	<code>x.__floordiv__(y)</code>
Modulo	<code>x % y</code>	<code>x.__mod__(y)</code>
Suma	<code>x += y</code>	<code>x.__iadd__(y)</code>
Resta	<code>x -= y</code>	<code>x.__isub__(y)</code>
Multiplicación	<code>x *= y</code>	<code>x.__imul__(y)</code>
División	<code>x /= y</code>	<code>x.__itruediv__(y)</code>
División entera	<code>x //= y</code>	<code>x.__ifloordiv__(y)</code>
Modulo	<code>x %= y</code>	<code>x.__imod__(y)</code>
Igualdad	<code>x == y</code>	<code>x.__eq__(y)</code>
Distinto	<code>x != y</code>	<code>x.__ne__(y)</code>
Menor que	<code>x < y</code>	<code>x.__lt__(y)</code>
Menor o igual que	<code>x <= y</code>	<code>x.__le__(y)</code>
Mayor que	<code>x > y</code>	<code>x.__gt__(y)</code>
Mayor o igual que	<code>x >= y</code>	<code>x.__ge__(y)</code>
Cuando se pide una copia	<code>copy.copy(x)</code>	<code>x.__copy__()</code>
Copia en profundidad	<code>copy.deepcopy(x)</code>	<code>x.__deepcopy__()</code>
Al entrar en un bloque with	<code>with x:</code>	<code>x.__enter__()</code>
Al salir del bloque with	<code>with x:</code>	<code>x.__exit__(exc_type, exc_value, traceback)</code>
Destructor del objeto	<code>del x</code>	<code>x.__del__()</code>

<https://docs.python.org/es/3.9/reference/datamodel.html>

```
class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Punto(self.x+other.x, self.y+other.y)

    def __str__(self):
```

```

        return f"({self.x},{self.y})"

print(Punto(1, 2) + Punto(3, 4))

```

```

# excerpt from zipfile.py
class _ZipDecrypter:
    .
    .
    .
    def __init__(self, pwd):
        self.key0 = 305419896
        self.key1 = 591751049
        self.key2 = 878082192
        for p in pwd:
            self._UpdateKeys(p)

    def __call__(self, c):
        assert isinstance(c, int)
        k = self.key2 | 2
        c = c ^ (((k * (k^1)) >> 8) & 255)
        self._UpdateKeys(c)
        return c
    .
    .
    .
zd = _ZipDecrypter(pwd)
bytes = zef_file.read(12)
h = list(map(zd, bytes[0:12]))

```

Con los métodos mágicos podemos hacer que una clase se pueda usar en una estructura **for in**.

```

class Invertir:
    def __init__(self, cadena):
        self.cadena = cadena
        self.puntero = len(cadena)

    def __iter__(self):
        return (self)

    def __next__(self):
        if self.puntero == 0:
            raise StopIteration # Obligatorio para finalizar
        self.puntero = self.puntero - 1
        return (self.cadena[self.puntero])

    # Declara iterable y recorre caracteres
    cadena_invertida = Invertir('Iterable')
    for caracter in cadena_invertida:
        print(caracter, end=' ')

```

Métodos *gétters and setters*

Ya conocemos las propiedades, pero también podemos usar otro mecanismo para crear propiedades dinámicamente además de lo ya tratado en el tema anterior. En Python se hacen según el siguiente ejemplo:

```

class Punto:
    def __init__(self, x: int = 0, y: int = 0):
        self.x = x
        self.y = y

    def __getattr__(self, name):
        if name == "long":
            return self.long

    def __setattr__(self, key, value):
        if key == "long" and value in range(1, 10):
            self.__dict__[key] = value

miPunto = Punto(2, 3)
miPunto.long = 5
print(miPunto.long)      # 5
print(miPunto.lon)       # None
miPunto.long = 10
print(miPunto.long)      # 5, ya que no cumple la
                        # condición de entre 1 y 9, range va de n a m-1

```

Toda la lógica se introduce dentro de los métodos especiales `__getattr__` y `__setattr__` que serán llamados respectivamente cuando haya que recoger el valor de una propiedad que no existe o establecer su valor.

En el ejemplo anterior vemos que cuando se establece una propiedad (`long`) se crean sus restricciones de acceso y si no existe se añade al diccionario del objeto (`__dict__`) cuyo efecto es crear la propiedad `long` en dicho objeto.

Clase object

En Python todo es un objeto, aunque no lo parezca, por lo que tiene que existir una jerarquía bien definida, y por tanto un parente de dicha jerarquía. Bajo Python será la clase `object` la que actúa como parente. Si no definimos un parente como mínimo en la definición de una clase, será el sistema el que asigne la clase `object` como parente, es imprescindible que toda clase tenga al menos un parente excepto `object`.

Esta clase implementa la funcionalidad básica para todo objeto, encontramos los siguientes métodos y propiedades: `'__class__'`, `'__delattr__'`, `'__dir__'`, `'__doc__'`, `'__eq__'`, `'__format__'`, `'__ge__'`, `'__getattribute__'`, `'__gt__'`, `'__hash__'`, `'__init__'`, `'__init_subclass__'`, `'__le__'`, `'__lt__'`, `'__ne__'`, `'__new__'`, `'__reduce__'`, `'__reduce_ex__'`, `'__repr__'`, `'__setattr__'`, `'__sizeof__'`, `'__str__'`, `'__subclasshook__'`.

Polimorfismo

El polimorfismo implica que, si en una porción de código se invoca un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Utilizando polimorfismo podemos invocar un mismo método de diferentes objetos y obtener diferentes resultados según la clase de estos. Podemos llamar a un método exactamente igual a otro y el intérprete automáticamente detectará a cuál de ellos nos referimos según diversos parámetros, por ejemplo, el tipo de dato que pasamos como argumento al momento de llamarlo, la clase a la que pertenece, o hasta podemos especificarle a qué método nos referimos. El polimorfismo está estrechamente ligado al concepto de Herencia

```

class Marino: #Clase Padre
    def hablar(self): #Método Hablar
        print ("Hola..")

class Pulpo(Marino): #Clase Hija
    def hablar (self): #Método Hablar
        print ("Soy un Pulpo")

```

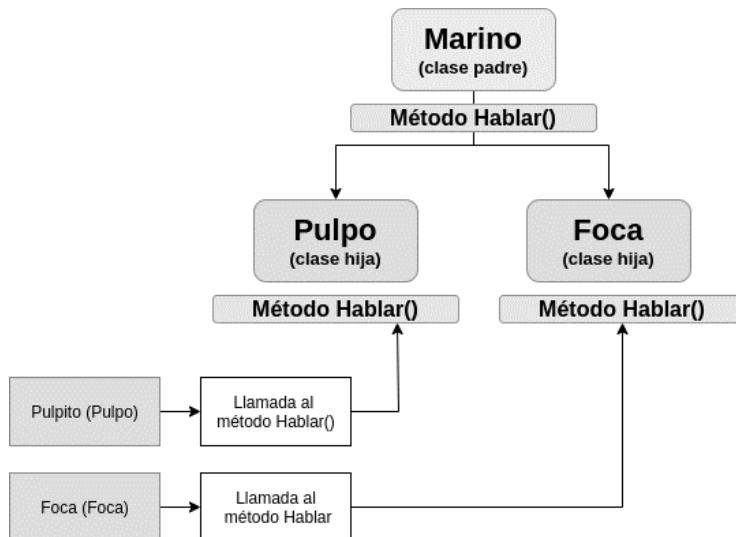
```

class Foca(Marino): #Clase Hija
    def hablar (self, mensaje): #Método Hablar
        print (mensaje)

Pulpito = Pulpo() #Instancia
Foca = Foca() #Instancia
Pulpito.hablar()
Foca.hablar("Soy una foca, este es mi mensaje")

```

Como podemos ver en el ejemplo el nombre de método es el mismo, cuando lo llamamos Python sabe cuál ejecutar porque se lo dice la clase a la que pertenece el objeto (en ese caso pulpito, pertenece a la clase "pulpo" por ende obviamente llamará al método hablar de dicha clase y no de ninguna otra.)



Determinación del tipo de un objeto

Una de las tareas más comunes a la hora de implementar nuestros métodos es determinar la clase de un objeto. Esto nos puede hacer falta al ser Python un lenguaje que determina en tiempo de ejecución el tipo de un parámetro, en vez de en tiempo de compilación. Como ya hemos comentado nuestros métodos pueden que necesiten realizar conversiones o comprobar tipos antes de efectuar una operación para evitar excepciones de ejecución.

Para poder determinar el tipo de un objeto tenemos dos "funciones": **instanceof** y **type**. La primera pide dos parámetros, el tipo y el objeto y devuelve True o False; la segunda devuelve una cadena con el tipo correspondiente que podremos comparar con el operador **is**.

```

mi_variable = 10
print(isinstance(mi_variable, int))
print(isinstance(mi_variable, str))
print(type(mi_variable))
print(type(mi_variable) is int)

```

Atributos de clase

Las variables de clases son definidas en la cabecera de la clase, pero nunca dentro de un método. Estas variables serán accesibles a todos los objetos de la clase a través de la nomenclatura **NombreClase.variable_clase**, no a través del objeto **self** (aunque si no existe la propiedad de instancia también accederemos a ella a través de **self**). La propiedad es compartida entre todos los objetos o, mejor dicho, es única para todos los objetos y cualquiera que la modifique se reflejará en los demás. También se puede acceder como si fuera una variable estática, con lo que es accesible al resto del programa.

```
class Clase:
```

```

valor_de_clase = 0

def metodo(self):
    print(self.valor_de_clase)

clase_1 = Clase()
clase_2 = Clase()
clase_1.metodo()
clase_2.metodo()
Clase.valor_de_clase += 1
clase_1.metodo()
clase_2.metodo()
clase_1.valor_de_clase += 3 # crea una propiedad local
clase_1.metodo()
clase_2.metodo()
Clase.valor_de_clase += 1
clase_1.metodo() # ya accede a la local
clase_2.metodo()
print(Clase.valor_de_clase)

```

Métodos de clase

Un método de clase está enlazado a la clase en vez de a la instancia, con lo que no necesita un objeto para poder utilizarlo. Es muy similar al estático que veremos más adelante, pero en este caso toma como primer parámetro la clase (atributo **cls**) por la que puede acceder a las variables de clase directamente a través del parámetro en vez del nombre de clase, por lo que puede modificar el estado de la clase directamente.

```

# Python program to demonstrate
# use of class method and static method.
from datetime import date

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a class method to create a Person object by birth year.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check if a Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print (person1.age)
print (person2.age)
print (Person.isAdult(22))

```

Clases y métodos abstractos

Un método abstracto es aquel que obliga a la clase hija a sobrescribirlo. Para la creación de métodos abstractos es necesario importar el módulo **abc** con al menos dos funcionalidades: **abstractmethod** y

ABCMeta. Deberemos definir la clase heredando de **ABCMeta** y a continuación determinar qué métodos son abstractos o la clase entera.

Clases y métodos estáticos

Un método estático es aquel que no necesita instanciar un objeto para llamarlo. Para definirlo, al igual que el punto anterior trataremos la clase como **ABCMeta** y con el decorador **@staticmethod** definiremos el método. No tomará parámetros de clase (**cls**) u objeto (**self**) solo podrá acceder a las variables de clase a través del nombre de clase y no podrá modificar el estado de clase directamente.

Clases y métodos finales

Un método final es aquel que no puede ser sobrescrito por las clases hijas, se implementó a partir de la versión 3.8 con el decorador **@final**, se debe importar del paquete **typing** y calificar la clase entera o el método que deseemos.

Ejemplos

```
from abc import abstractmethod
from abc import ABCMeta as ABCMeta
from typing import final
class miClaseAbstracta(metaclass=ABCMeta):
    def __init__(self):
        self.valor = 0

    @abstractmethod
    def miMetodoAbstracto(self, valor: int):
        self.valor = valor

    @staticmethod
    def miMetodoEstatico():
        print("Si")

    @final
    def miMetodoFinal(self):
        print("Si")

class miClase(miClaseAbstracta):
    def miMetodoAbstracto(self, valor: int):
        self.valor = valor
        print(self.valor)

    def miMetodoFinal(self): # debería lanzar un error
        print("No")
mc = miClase()
mc.miMetodoAbstracto(5)
miClase.miMetodoEstatico()
mc.miMetodoEstatico()
mc.miMetodoFinal()
```

Desarrolla

- ∞ Escriba una clase **Coche** de la que van a heredar **CocheCambioManual** y **CocheCambioAutomatico**. Los atributos de los coches son la matrícula, la velocidad y la marcha. Para este ejercicio no se permite la marcha atrás, por tanto, no se permiten ni velocidad negativa, ni marcha negativa. En el constructor se recibe el valor de la matrícula por parámetro y se inicializa el valor de la velocidad y la marcha a 0. Además, tendrá los siguientes métodos:
 - `get_matricula`: que devuelve el valor de la matrícula.
 - `get_marcha()`: devuelve el valor de la marcha.
 - `get_Velocidad()`: devuelve el valor de la velocidad
 - `acelerar()`: recibe por parámetro un valor al acelerar el coche.
 - `frenar()`: recibe por parámetro un valor al frenar el coche.
 - `__str__()`: devuelve en forma de String la información del coche.
 - `cambiar_marcha`: recibe por parámetro la marcha a la que se tiene que cambiar el coche.
- ∞ La clase **CocheCambioManual** sobrescribe el método `cambiar_marcha()` y lo hace público, para que pueda ser llamado desde cualquier clase. No permite que se cambie a una marcha negativa.
- ∞ La clase **CocheCambioAutomatico** sobrescribe los métodos `acelerar()` y `frenar()` para que cambie automáticamente de marcha conforme se va acelerando y frenando.
- ∞ Genere una clase `CocheMain` que cree dos objetos de cada clase, y pruebe a acelerar, frenar y `cambiar_marcha` en cada uno con valores positivos y negativos.

[code_0007.py](#)

Desarrolla

- ∞ Se plantea desarrollar un programa que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos:
 - Productos frescos
 - Productos refrigerados
 - Productos congelados
 - Todos los productos llevan esta información común: fecha de caducidad y número de lote. A su vez, cada tipo de producto lleva alguna información específica.
 - Los productos frescos deben llevar la fecha de envasado y el país de origen.
 - Los productos refrigerados deben llevar el código del organismo de supervisión alimentaria.
 - Los productos congelados deben llevar la temperatura de congelación recomendada.

[code_0008.py](#)

Desarrolla

- ∞ Desarrollar una clase llamada Punto que:
 - Tenga dos atributos.
 - Tenga un constructor con dos parámetros que inicialice los dos atributos.
El constructor por defecto (sin parámetros) que inicialice los dos atributos al valor que se quiera.
 - Tenga un método calcular_distancia_desde que recibe un parámetro de tipo Punto y que devuelve la distancia euclídea.
- ∞ Desarrollar una clase llamada Circulo que:
 - Tenga dos atributos de tipo Punto (centro del círculo) y radio (float).
 - Tenga un constructor con dos parámetros de tipo Punto y float que inicialice los dos atributos.
El constructor por defecto (sin parámetros) que inicialice los dos atributos al valor que se quiera.
El constructor con tres parámetros de tipo float que inicialice los dos atributos.
 - Tenga un método calcular_distancia_desde que recibe un parámetro de tipo Punto y que devuelve la distancia euclídea al centro.
 - Tenga un método calcular_area que no recibe ningún parámetro y devuelve el área.
 - Tenga un método calcular_logitud que no recibe ningún parámetro y devuelve la longitud de la circunferencia.
- ∞ Desarrollar una clase llamada Triangulo que:
 - Tenga tres atributos de tipo Punto.
 - Tenga un constructor con tres parámetros de tipo Punto que inicialice los dos atributos.
 - Tenga un método calcular_distancia_desde que recibe un parámetro de tipo Punto y que devuelve la distancia euclídea al primer punto.
 - Tenga un método calcular_area que no recibe ningún.
 - Tenga un método calcular_perimetro que no recibe ningún parámetro y devuelve el perímetro.
- ∞ Desarrollar una clase llamada Practica1 que en su método main:
 - Cree e inicialice dos objetos de la clase Punto y muestre la distancia entre ambos.
 - Cree un objeto de la clase Circulo y muestre su área, perímetro y distancia a uno de los dos puntos creados al comienzo.
 - Cree un objeto de la clase Triangulo y muestre su área, perímetro y distancia a un nuevo punto.

code_0009**Interfaces**

En la POO un interfaz es un conjunto de métodos, sin ningún código generalmente, que estamos obligados a implementar cuando lo definimos en una clase. El interfaz se definió para solucionar los problemas de los lenguajes de herencia simple en la que tenían que heredar de varias clases y no era posible. En Python no se presente este problema al tener herencia múltiple. El otro problema que da solución un interfaz es definir un conjunto de métodos (protocolo de comunicaciones) de obligada implementación, en este caso bajo Python se puede usar este concepto con las clases y métodos abstractos que ya hemos estudiado, definiendo toda la clase y sus métodos como abstractos sin ningún tipo de implementación.

```
from abc import ABC, abstractmethod

class AccountingSystem(ABC):

    @abstractmethod
    def create_purchase_invoice(self, purchase):
        pass

    @abstractmethod
    def create_sale_invoice(self, sale):
        pass
```

Ejemplo de interfaz bajo **PHP** que implementa herencia simple.

```
interface Automovil {
    public function getTipo();
    public function getRuedas();
}

class Coche implements Automovil {
    public function getTipo(){
        echo "Coche";
    }
    public function getRuedas() {
        echo "4";
    }
}

class Moto implements Automovil {
    public function getTipo(){
        echo "Moto";
    }
    public function getRuedas() {
        echo "2";
    }
}
```

Decoradores

Los decoradores alteran de manera dinámica la funcionalidad de una función, método o clase sin tener que hacer subclases o cambiar el código fuente de la clase decorada. Los decoradores y su utilización en nuestros programas nos ayudan a hacer nuestro código más limpio, a autodocumentarlo y, a diferencia otros lenguajes, no requieren que nos aprendamos otro lenguaje de programación distinto (como pasa con las anotaciones de Java por ejemplo). En su utilización podemos simular la programación orientada a aspectos (AOP) o utilizarlos para añadir sistemas de control a nuestras funciones, de log, caché, ... Las posibilidades son infinitas. Los decoradores forman parte de Python desde la versión 2.4 aportan:

- ∞ Reducen el código común y repetitivo (el llamado código boilerplate).
- ∞ Favorecen la separación de responsabilidades del código
- ∞ Aumentan la legibilidad y la mantenibilidad
- ∞ Los decoradores son explícitos.

Hemos utilizado decoradores a la hora de crear nuestras funciones: **@staticmethod**, **@classmethod**, **@final**, pero en este curso no vamos a ver cómo crearlos.

Generadores

Son funciones que nos permitirán obtener sus resultados poco a poco. Es decir, cada vez que llamemos a la función nos darán un nuevo resultado. Por ejemplo, una función para generar todos los números pares que cada vez que la llamemos nos devuelva el siguiente número par.

Para construir generadores sólo tenemos que usar la orden **yield**. Esta orden devolverá un valor (igual que hace **return**) pero, además, congelará la ejecución de la función hasta la próxima vez que le pidamos un valor en la instrucción siguiente a **yield**.

```
def pares():
    index = 1
    # En este caso definimos un bucle infinito
    while True:
        # Devolvemos un valor
        yield index*2
        index = index + 1
```

```
def busqueda_general():
    def search_nested_bad(array, valor_buscado):
        cords = None
        for i, row in enumerate(array):
            for j, celda in enumerate(row):
                if celda == valor_buscado:
                    cords = (i, j)
                    break
            if cords is not None:
                break

        if cords is None:
            raise ValueError("No encontrado")
        return cords

    print(search_nested_bad(valores, 22))

def busqueda_optima():
    def generar_elementos(array):
        for i, row in enumerate(array):
            for j, celda in enumerate(row):
                yield (i, j), celda

    def search_nested(array, valor_buscado):
        try:
            cords = next(  # creamos un generador para la
            lista devuelta y cogemos el primero solo
            cords
            for (cords, cell) in generar_elementos(array)
            if cell == valor_buscado
        )
        except StopIteration as e:
            raise ValueError("No encontrado")

    return cords

    print(search_nested(valores, 22))
```

```

valores = [
    [1, 2, 3],
    [11, 22, 33],
    [111, 222, 333]
]
busqueda_general()
busqueda_optima()

```

Excepciones

Definición

Aunque una declaración o expresión sea sintácticamente correcta, puede generar un error cuando se intenta ejecutar. Los errores detectados durante la ejecución se llaman **excepciones** y se pueden gestionar.

Tipos

Python trata todos los errores de la misma forma, como excepciones, pero utiliza dos categorías para indicar la severidad del error: Excepciones y Warnings. Las primeras deben ser capturadas y tratadas y para las segundas se puede configurar el sistema para ignorarlas. El mecanismo usado para controlar la gestión de los warnings es el filtro de advertencias, que controla si las advertencias se ignoran, se muestran o se convierten en errores (planteando una excepción). Se definen los siguientes warnings.

Clase	Descripción
Warning	Esta es la clase principal para todas las clases que pertenecen a la categoría de advertencia. Es una subclase de Exception.
UserWarning	La categoría por defecto para warn().
DeprecationWarning	Categoría principal para advertencias sobre características obsoletas cuando esas advertencias están destinadas a otros desarrolladores de Python (ignoradas por defecto, a menos que sean activadas por el código en <code>__main__</code>).
SyntaxWarning	Categoría principal para las advertencias sobre características sintácticas dudosas.
RuntimeWarning	Categoría principal para las advertencias sobre características dudosas de tiempo de ejecución.
FutureWarning	Categoría principal para las advertencias sobre características obsoletas cuando esas advertencias están destinadas a los usuarios finales de aplicaciones escritas en Python.
PendingDeprecationWarning	Categoría principal para las advertencias sobre las características que serán desaprobadas en el futuro (ignoradas por defecto).
ImportWarning	Categoría principal para las advertencias que se activan durante el proceso de importación de un módulo (se ignoran por defecto).
UnicodeWarning	Categoría principal para las advertencias relacionadas con Unicode.
BytesWarning	Categoría principal para las advertencias relacionadas con bytes y bytearray.
ResourceWarning	Categoría base para las advertencias relacionadas con el uso de los recursos.

Jerarquías

En Python, todas las excepciones deben ser instancias de una clase que se derive de **BaseException**. En una instrucción **try** con una cláusula **except** que menciona una clase determinada, esa cláusula también controla las clases de excepción derivadas(hijas), es decir, si capturamos una excepción también se capturarán todas sus hijas si no lo hacemos explícitamente antes. Dos clases de excepción que no están relacionadas mediante subclases (hermanas o herederas de hermanas) nunca son equivalentes, incluso si tienen el mismo nombre.

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +- StopIteration
    +- StopAsyncIteration
    +- ArithmeticError
        |   +- FloatingPointError
        |   +- OverflowError
        |   +- ZeroDivisionError
    +- AssertionError
    +- AttributeError
    +- BufferError
    +- EOFError
    +- ImportError
        |   +- ModuleNotFoundError
    +- LookupError
        |   +- IndexError
        |   +- KeyError
    +- MemoryError
    +- NameError
        |   +- UnboundLocalError
    +- OSError
        |   +- BlockingIOError
        |   +- ChildProcessError
        |   +- ConnectionError
            |       +- BrokenPipeError
            |       +- ConnectionAbortedError
            |       +- ConnectionRefusedError
            |       +- ConnectionResetError
        |   +- FileNotFoundError
        |   +- InterruptedError
        |   +- IsADirectoryError
        |   +- NotADirectoryError
        |   +- PermissionError
        |   +- ProcessLookupError
        |   +- TimeoutError
    +- ReferenceError
    +- RuntimeError
        |   +- NotImplementedError
        |   +- RecursionError
    +- SyntaxError
        |   +- IndentationError
        |       +- TabError
    +- SystemError
    +- TypeError
    +- ValueError
        |   +- UnicodeError
            |       +- UnicodeDecodeError
            |       +- UnicodeEncodeError
            |       +- UnicodeTranslateError
    +- Warning
        +- DeprecationWarning
        +- PendingDeprecationWarning
        +- RuntimeWarning
```

```
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning
```

Captura

Es posible escribir programas que gestionen determinadas excepciones. Véase el siguiente ejemplo, que le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando Control-C o lo que soporte el sistema operativo); nótese que una interrupción generada por el usuario es señalizada generando la excepción **KeyboardInterrupt**.

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break    # fin del while
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```

La declaración **try** funciona de la siguiente manera:

- ∞ Primero, se ejecuta la cláusula **try** (la(s) linea(s) entre las palabras reservadas **try** y la **except**).
- ∞ Si no ocurre ninguna excepción, la cláusula **except** se omite y la ejecución de la cláusula **try** finaliza.
- ∞ Si ocurre una excepción durante la ejecución de la cláusula **try** el resto de la cláusula se omite. Entonces, si el tipo de excepción coincide con la excepción indicada después de la **except**, la cláusula **except** se ejecuta, y la ejecución continua después de la **try**.
- ∞ Si ocurre una excepción que no coincide con la indicada en la cláusula **except** se pasa a los **try** más externos; si no se encuentra un gestor, se genera una **unhandled exception** (excepción no gestionada) y la ejecución se interrumpe con un mensaje.

Una declaración **try** puede tener más de un **except**, para especificar gestores para distintas excepciones. Una clase en una cláusula **except** es compatible con una excepción si es de la misma clase o de una clase derivada de la misma por lo que el orden de las sentencias **except** es muy importante.

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

En el ejemplo anterior si las cláusulas **except** estuvieran invertidas (con **except B** primero), habría impreso B, B, B — se usa la primera cláusula **except** coincidente.

Puede existir un último **except** omitiendo el nombre de la excepción capturada y servir como comodín para capturar cualquiera no indicada, así como un bloque **else** opcional, el cual, cuando está presente, debe seguir a los **except** y se ejecutará si el código dentro de try no genera errores.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

La cláusula **except** puede especificar una variable después del nombre de excepción para recoger los datos de la excepción y tratarla de forma adecuada.

```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst)) # the exception instance
    print(inst.args) # arguments stored in .args
    print(inst) # __str__ allows args to be printed directly,
    x, y = inst.args # unpack args
    print('x =', x)
    print('y =', y)
```

Para finalizar se puede incluir un bloque **finally** que se ejecutará siempre en última instancia.

```
try
...
} finally {
    # bloque que siempre se ejecutará el último
    # independientemente de si hay o no excepción
    # después del bloque else si se tiene que ejecutar
}
```

Los gestores de excepciones no se encargan solamente de las excepciones que ocurren en el bloque try, también gestionan las excepciones que ocurren dentro de las funciones que se llaman (inclusive indirectamente) dentro del bloque try.

Generación de Excepciones

La declaración **raise** permite al programador forzar a que ocurra una excepción específica. Por ejemplo:

```
raise NameError('HiThere')
```

El único argumento de **raise** indica la excepción a generarse. Tiene que ser o una instancia de excepción, o una clase de excepción (una clase que hereda de `Exception`). Si se pasa una clase de excepción, la misma será instanciada implícitamente llamando a su constructor sin argumentos:

```
raise ValueError # shorthand for 'raise ValueError()'
```

Si es necesario determinar si una excepción fue lanzada, pero sin intención de gestionarla, una versión simplificada de la instrucción **raise** te permite relanzarla:

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
```

```
raise
```

La cláusula **raise** permite un sentencia **from** opcional para habilitar las excepciones en cadena.

```
# exc must be exception instance or None.  
raise RuntimeError from exc
```

Es muy útil para transformar unas excepciones en otras.

```
def func():  
    raise IOError  
  
try:  
    func()  
except IOError as exc:  
    raise RuntimeError('Failed to open database') from exc  
  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
    File "<stdin>", line 2, in func  
  OSError  
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
RuntimeError: Failed to open database
```

El encadenado de excepciones se puede deshabilitar utilizando **None** como parámetro del **from**.

```
try:  
    open('database.sqlite')  
except IOError:  
    raise RuntimeError from None  
  
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
RuntimeError
```

Definición de Excepciones propias

Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción a partir de cualquier otra. Las clases de Excepción pueden ser definidas de la misma forma que cualquier otra clase, pero es habitual mantenerlas lo más simples posible, a menudo ofreciendo solo un número de atributos con información sobre el error que leerán los gestores de la excepción. Al crear un módulo que puede lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error.

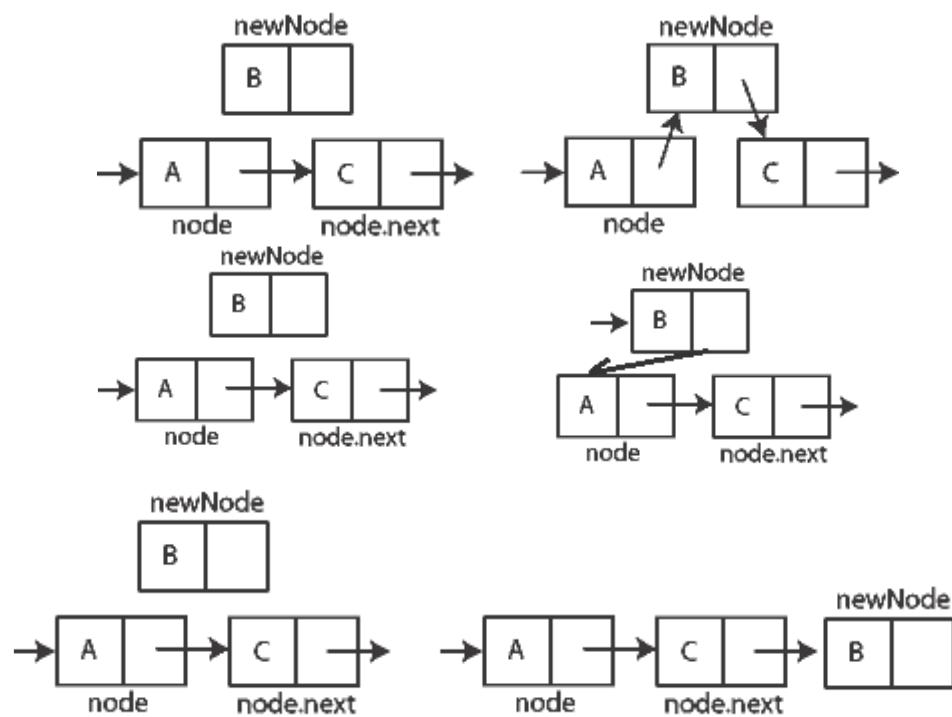
```
class Error(Exception):  
    """Base class for exceptions in this module."""  
    pass  
  
class InputError(Error):  
    def __init__(self, expression, message):  
        self.expression = expression  
        self.message = message  
  
class TransitionError(Error):  
    def __init__(self, previous, next, message):  
        self.previous = previous  
        self.next = next  
        self.message = message
```

Estructuras de datos

Qué es una lista

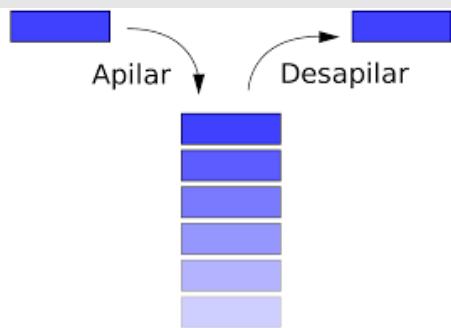
Una lista es una estructura de datos que almacena información en nodos independientes enlazados entre sí. Es un almacén dinámico que crece y decrece con el tiempo. En Python están implementados bajo la clase List.

Lista simplemente enlazada.



Usando listas como pilas

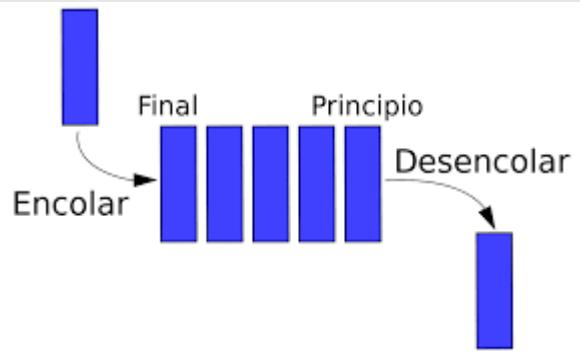
Los métodos de lista hacen que resulte muy fácil usar una lista como una pila, donde el último elemento añadido es el primer elemento retirado (“último en entrar, primero en salir” o LIFO). Para agregar un elemento a la cima de la pila, utiliza **append()**. Para retirar un elemento de la cima de la pila, utiliza **pop()** sin un índice explícito.



```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack.pop()
stack.pop()
stack.pop()
```

Usando listas como colas

También es posible usar una lista como una cola, donde el primer elemento añadido es el primer elemento retirado (“primero en entrar, primero en salir” o FIFO); sin embargo, las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento (porque todos los otros elementos tienen que ser desplazados uno a la izquierda).

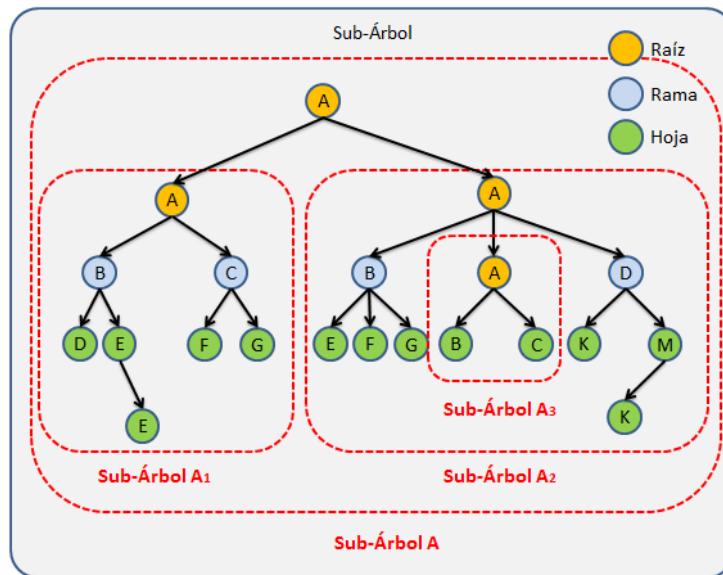


Para implementar una cola, se utiliza **collections.deque** el cual fue diseñado para añadir y quitar de ambas partes de forma rápida.

```
from collections import deque

queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
queue.append("Graham")         # Graham arrives
queue.popleft()                # The first to arrive now leaves
queue.popleft()                # The second to arrive now leaves
deque(['Michael', 'Terry', 'Graham'])
```

Árboles



Las estructuras en árboles son muy utilizadas en informática. Un árbol es una estructura que parte de una única raíz y cada nodo puede tener entre cero y n hijos, pero cada hijo puede tener exclusivamente un parente. Si el número de hijos se establece obligatoriamente a dos, se crean los árboles de decisión o árboles binarios como vemos en el ejemplo a continuación.

```
class Arbol(object):
    def __init__(self):
        self.der = None
        self.izq = None
        self.dato = None

    def inorder(self, h=None):
        if h is None: h = self
        derecha = h.inorden(h.der) if h.der else "\n"
        izquierda = h.inorden(h.izq) if h.izq else "\n"
        return izquierda + h.dato + derecha

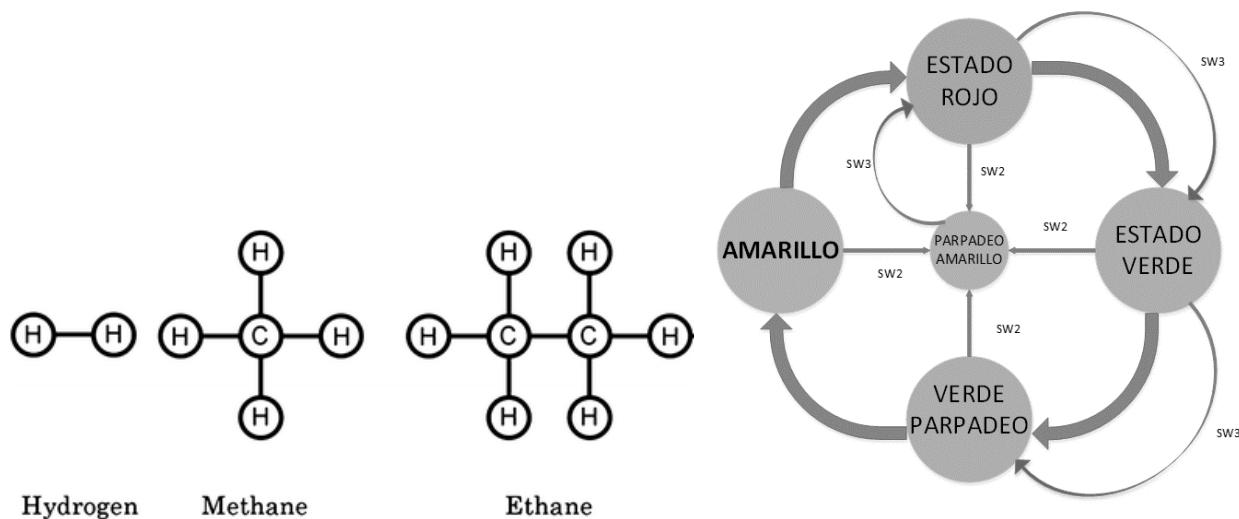
raiz = Arbol()
raiz.dato = 'Raiz'
raiz.izq = Arbol()
raiz.izq.dato = '1-izq'
raiz.der = Arbol()
raiz.der.dato = '2-der'

raiz.izq.izq = Arbol()
raiz.izq.izq.dato = '3-izq-izq'
raiz.izq.der = Arbol()
raiz.izq.der.dato = '4-izq-der'

print(raiz.inorden())
```

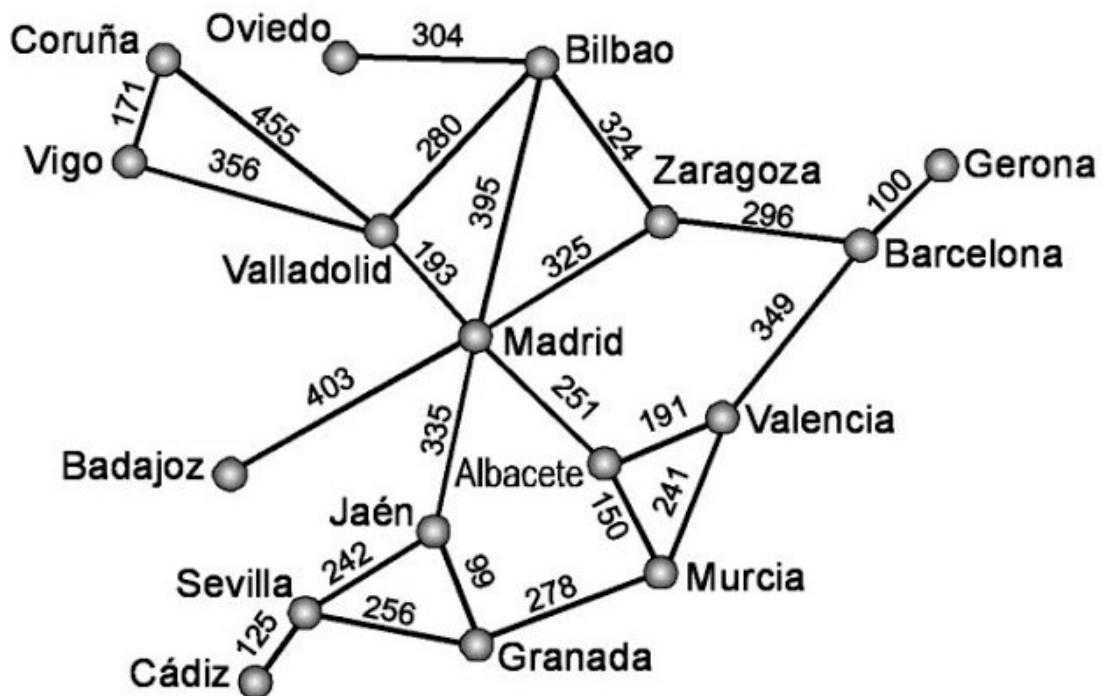
Grafos.

Un grafo es un conjunto de nodos unidos por aristas sin restricciones entre las conexiones y en el cual las aristas pueden tener un peso asociado y pueden o no tener una dirección. Desde Python existen varias librerías para este fin, pero vamos a utilizar **networkx**.



Con **nx.Graph()** creamos el grafo (podemos usar **nx.DiGraph()** para crear un grafo dirigido). Para añadir nodos usamos **add_node()** para añadir un sólo nodo o **add_nodes_from()** para añadir varios de una vez. Para las aristas usamos **add_edge()** o **add_edges_from()** dependiendo de si queremos agregar una o varias aristas.

Es posible asignar atributos tanto a los nodos como a las aristas. Por ejemplo, podemos especificar que Tom Hanks tiene dos Oscars. Respecto a las aristas, como representan que un actor ha trabajado con otro, podemos añadir un atributo que sea la película en la que ambos han coincidido. Un uso muy habitual que tienen los atributos sobre las aristas es indicar el peso si se trata de un grafo ponderado, mediante el atributo **weight**.



```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph() # crear un grafo

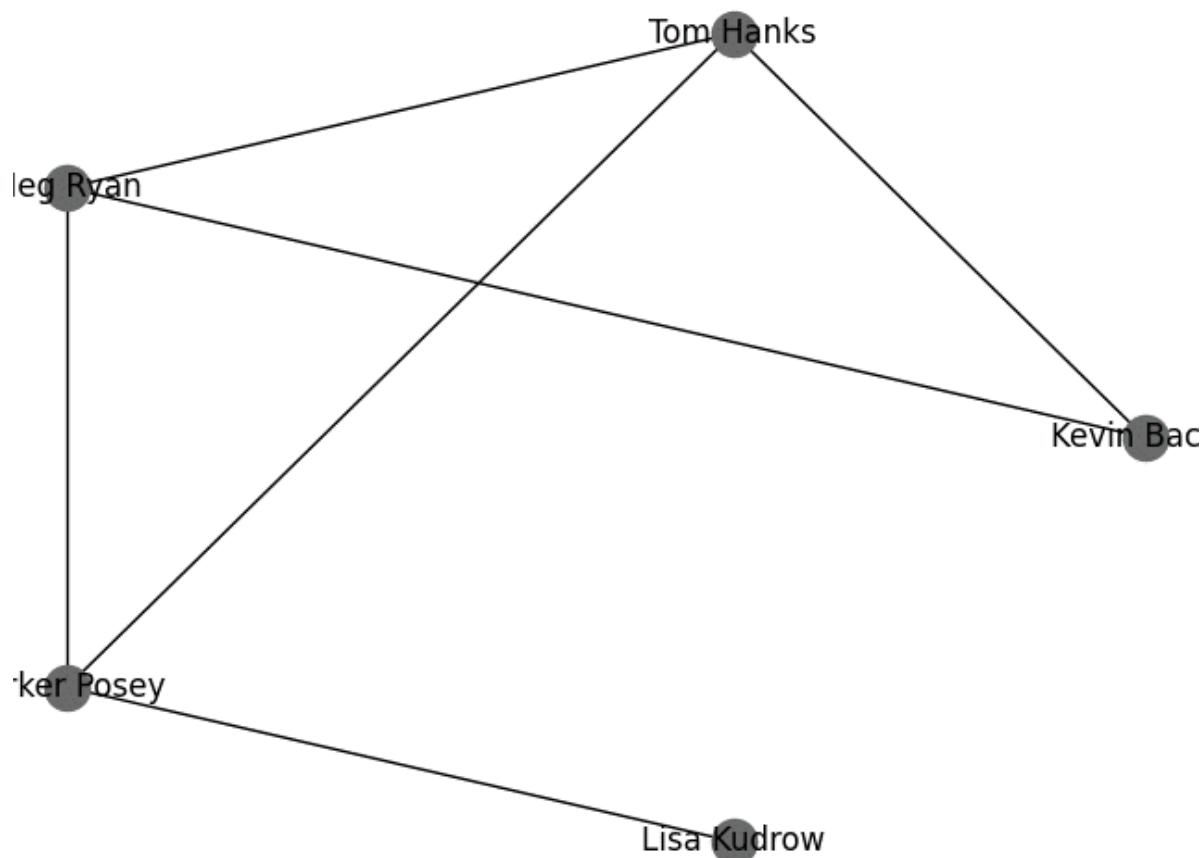
# Añadir nodos
G.add_node("Kevin Bacon")
G.add_node("Tom Hanks")
G.add_nodes_from(["Meg Ryan", "Parker Posey", "Lisa Kudrow"])

# Añadir aristas
G.add_edge("Kevin Bacon", "Tom Hanks")
G.add_edge("Kevin Bacon", "Meg Ryan")
G.add_edges_from([('Tom Hanks', 'Meg Ryan'), ('Tom Hanks', 'Parker Posey')])
G.add_edges_from([('Parker Posey', 'Meg Ryan'), ('Parker Posey', 'Lisa Kudrow')])

print(len(G.nodes))
print(len(G.edges))
print(G.nodes)
print(G.edges)

# asignar atributos a nodos y aristas
G.nodes["Tom Hanks"]["oscars"] = 2
G.edges["Kevin Bacon", "Tom Hanks"]["pelicula"] = "Apolo 13"
G.edges["Kevin Bacon", "Meg Ryan"]["pelicula"] = "En carne viva"
G.edges["Parker Posey", "Meg Ryan"]["pelicula"] = "Algo para recordar"
G.edges["Parker Posey", "Tom Hanks"]["pelicula"] = "Tienes un email"
G.edges["Parker Posey", "Lisa Kudrow"]["pelicula"] = "Esperando la hora"
print(G.nodes(data=True))
print(G.edges(data=True))
print(G["Kevin Bacon"])

pos = nx.circular_layout(G)
nx.draw(G, pos, with_labels=True)
plt.show()
```



<https://networkx.org/documentation/stable/tutorial.html>

Creación de casos de prueba

Las pruebas son la base del sólido desarrollo de software. Hay muchos tipos de pruebas, pero el tipo más importante es la prueba unitaria. La prueba unitaria te da mucha confianza de que puedes usar piezas bien probadas como primitivas y confiar en ellas cuando las compongas para crear tu programa. Aumentan tu inventario de código confiable más allá de tu lenguaje y la biblioteca estándar. Además, Python proporciona un gran soporte para escribir pruebas unitarias.

El módulo unittest viene con la biblioteca estándar de Python. Proporciona una clase llamada **TestCase**, de la que se puede derivar su clase. A continuación, puedes sobrescribir un método **setUp()** para preparar un dispositivo de prueba antes de cada prueba y/o un método de clase **classSetUp()** para preparar un dispositivo de prueba para todas las pruebas (no se restablece entre pruebas individuales). Existen métodos correspondientes de **tearDown()** y **classtearDown()** que también puedes reemplazar para hacer la liberación.

Aquí están las partes relevantes de nuestra clase `SelfDrivingCarTest`. Sólo utilizo el método `setUp()`. Creo una nueva instancia `SelfDrivingCar` y la almaceno en `self.car` para que esté disponible en cada prueba.

```
from unittest import TestCase

class SelfDrivingCarTest(TestCase):
    def setUp(self):
        self.car = SelfDrivingCar()
```

El siguiente paso es escribir métodos de prueba específicos para probar que el código está haciendo lo que se supone que debe hacer. La estructura de un método de prueba es bastante estándar:

- ∞ Preparar el entorno (opcional).
- ∞ Prepara el resultado esperado.
- ∞ Llama el código bajo prueba.
- ∞ Asegúrate que el resultado real coincide con el resultado esperado.

Hay que tener en cuenta que el resultado no tiene que ser la salida de un método. Puede ser un cambio de estado de una clase, un efecto secundario como añadir una nueva fila en una base de datos, escribir un archivo o enviar un correo electrónico.

Por ejemplo, el método `stop()` de la clase `SelfDrivingCar` no devuelve nada, pero cambia el estado interno estableciendo la velocidad en 0. El método `assertEqual()` proporcionado por la clase base `TestCase` se utiliza aquí para verificar que el parámetro llamado `stop()` funcionó como se esperaba.

```
def test_stop(self):  
    self.car.speed = 5  
    self.car.stop()  
    # Verify the speed is 0 after stopping  
    self.assertEqual(0, self.car.speed)  
    # Verify it is Ok to stop again if the car is already stopped  
    self.car.stop()  
    self.assertEqual(0, self.car.speed)
```

En realidad, hay dos pruebas aquí. La primera prueba es asegurarse de que si la velocidad del coche es 5 y si se llama el método `stop()`, entonces la velocidad se convierte en 0. Luego, la otra prueba es asegurar que nada va mal si se llama a `stop()` de nuevo cuando el coche ya está detenido.

El número de posibles comprobaciones (`assert`) son numerosas, comprobar la lista a continuación.

<https://docs.python.org/es/3.9/library/unittest.html>

Un ejemplo un poco más elaborado.

```
import unittest  
  
class TestStringMethods(unittest.TestCase):  
  
    def test_upper(self):  
        self.assertEqual('foo'.upper(), 'FOO')  
  
    def test_isupper(self):  
        self.assertTrue('FOO'.isupper())  
        self.assertFalse('Foo'.isupper())  
  
    def test_split(self):  
        s = 'hello world'  
        self.assertEqual(s.split(), ['hello', 'world'])  
        with self.assertRaises(TypeError):  
            s.split(2)  
  
    if __name__ == '__main__':  
        unittest.main()
```

Ejercicio

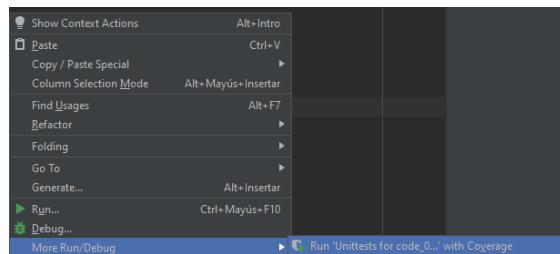
Desarrolla las pruebas unitarias para los ejercicios: **code_0006.py**, **code_0007.py**, **code_0008.py** y **code_0009**

Cobertura de las pruebas



```
18
19     class Productos:
20         def __init__(self):
21             self.fecha = None
22             self.numero = 0
23
24         class Frescos(Productos):
25             def __init__(self):
26                 super().__init__()
27                 self.fecha = None
28                 self.pais = None
29
30         class Refrigerado(Productos):
31             def __init__(self):
32                 super().__init__()
33                 self.codigo = None
34
35         class Congelados(Productos):
36             def __init__(self):
37                 super().__init__()
38                 self.temperatura = None
39
40             if __name__ == '__main__':
41                 print('Este es el código principal')
```

Se pueden realizar las pruebas de forma estándar, realizando los test, o con cobertura de código, lo que nos dará una visualización de nuestro código que está siendo testeado y cuál no.



A la izquierda vemos áreas en verde (si las ha testeado nuestras pruebas) y áreas en rojo las que no han sido testeadas. Podemos usar esta información para escribir nuestras pruebas y que no falte ni una línea de código sin testear,

Pero el que estén todas las líneas testeadas no significa que todas las condiciones estén comprobadas, así en un **if** que tenga dos condiciones, con comprobar una la cobertura nos mostrará como testeada, pero en verdad faltaría de comprobar todas las posibilidades de la condición con todos los valores límite.

Ejercicios

Apartado A

Retomar apartado A y B de ejercicios del tema anterior y modificarlos.

Ejercicio A1 – Punto

Necesitamos un método que nos permita crear un objeto Punto con coordenadas aleatorias. Esta funcionalidad no depende de ningún objeto concreto por lo que será estática. Deberá crear un nuevo Punto (utiliza el constructor) con x e y entre -100 y 100, y luego devolverlo (con return).

- `crear_punto_aleatorio()`

Pruébalo en el main para comprobar que funciona. Crea varios puntos aleatorios con Punto. `crear_punto_aleatorio ()` e imprime su valor por pantalla.

Ejercicio A2 – Persona

El DNI de una persona no puede variar.

La mayoría de edad a los 18 años es un valor común a todas las personas y no puede variar. Crea un nuevo atributo llamado **mayoria_edad** que no se pueda modificar. Tendrás que inicializarlo a 18 en la declaración. Utilízalo en el método que comprueba si una persona es mayor de edad.

Crea un método **validar_dni(String dni)** que devuelva true si dni es válido (tiene 8 números y una letra). Si no, devolverá false. Utilízalo en el constructor para comprobar el dni (si no es válido, muestra un mensaje de error y no guardes los valores).

Realiza algunas pruebas en el main para comprobar el funcionamiento de los cambios realizados. También puedes utilizar `Persona.validar_dni(...)` por ejemplo para comprobar si unos DNI introducidos por teclado son válidos o no (sin necesidad de crear ningún objeto).

Ejercicio A3 – Rectángulo

Necesitamos hacer algunos cambios para que todas las coordenadas estén entre (0,0) y (100,100). Añade a la clase Rectángulo dos atributos llamados min y max. Estos valores son comunes a todos los objetos y no pueden variar. Piensa qué modificados necesitas añadir a min y max.

Utiliza min y max en el constructor y en los setters para comprobar los valores (como de costumbre, si no son correctos muestra un mensaje de error y apliques los cambios).

También necesitamos un método no constructor para crear rectángulos aleatorios. Impleméntalo.

Realiza pruebas en el main para comprobar su funcionamiento.

Ejercicio A4 – Artículo

En España existen tres tipos de IVA según el tipo de producto:

- El IVA general (21%): para la mayoría de productos a la venta.
- El IVA reducido (10%): hostelería, transporte, vivienda, etc.
- El IVA super reducido (4%): alimentos básicos, libros, medicamentos, etc.

Estos tres tipos de IVA no pueden variar y a cada artículo se le aplicará uno de los tres.

Razona qué cambios sería necesario realizar a la clase Artículo e impleméntalos.

Apartado B - ASTROS

Define una jerarquía de clases que permita almacenar datos sobre los planetas y satélites (lunas) que forman parte del sistema solar.

Algunos atributos que necesitaremos almacenar son:

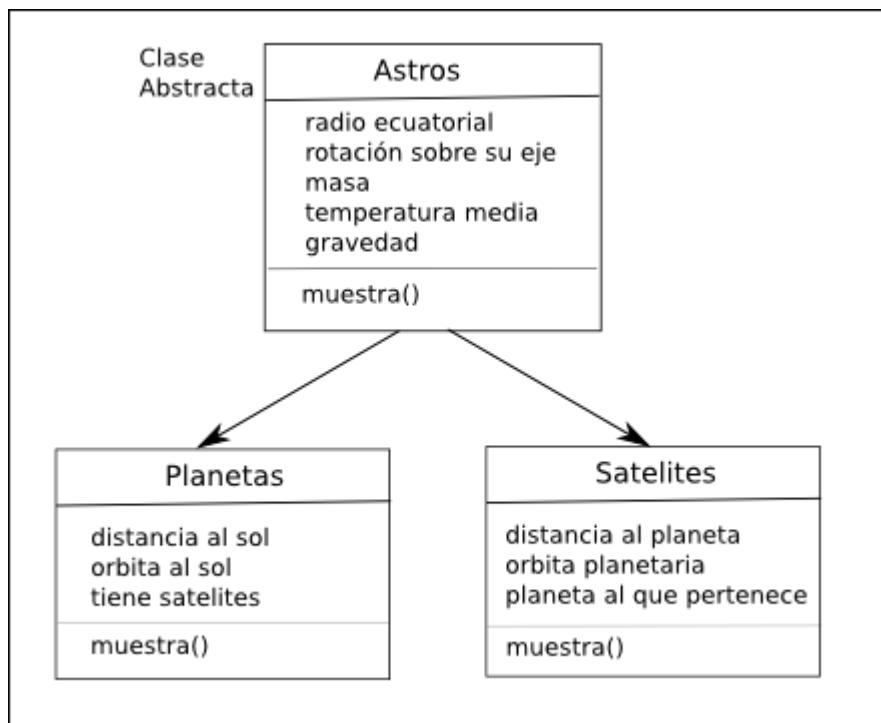
- Masa del cuerpo.
- Diámetro medio.
- Período de rotación sobre su propio eje.
- Período de traslación alrededor del cuerpo que orbitan.
- Distancia media a ese cuerpo.
- etc.

Define las clases necesarias conteniendo:

- Constructores.
- Métodos para recuperar y almacena atributos.
- Método para mostrar la información del objeto.

Define un método, que dado un objeto del sistema solar (planeta o satélite), imprima toda la información que se dispone sobre el mismo (además de su lista de satélites si los tuviera).

El diagrama UML sería:



Una posible solución sería crear una lista de objetos, insertar los planetas y satélites (directamente mediante código o solicitándolos por pantalla) y luego mostrar un pequeño menú que permita al usuario imprimir la información del astro que elija.

Apartado C - MASCOTAS

Implementa una clase llamada **Inventario** que utilizaremos para almacenar referencias a todos los animales existentes en una tienda de mascotas.

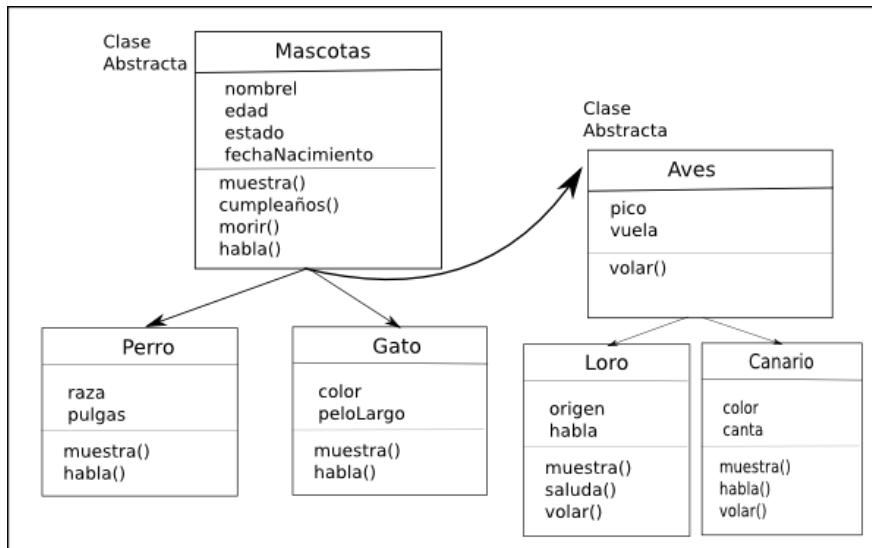
Esta clase debe cumplir con los siguientes requisitos:

- En la tienda existirán 4 tipos de animales: perros, gatos, loros y canarios.
- La clase debe permitir realizar las siguientes acciones:

- Mostrar la lista de animales (solo tipo y nombre, 1 línea por animal).
- Mostrar todos los datos de un animal concreto.
- Mostrar todos los datos de todos los animales.
- Insertar animales en el inventario.
- Eliminar animales del inventario.
- Vaciar el inventario.

Implementa las demás clases necesarias para la clase **Inventario**.

El diagrama UML sería:



Apartado D – BANCO

Vamos a hacer una aplicación que simule el funcionamiento de un banco.

Crea una clase **CuentaBancaria** con los atributos: **iban** y **saldo**. Implementa métodos para:

- Consultar los atributos.
- Ingresar dinero.
- Retirar dinero.
- Traspasar dinero de una cuenta a otra.

Para los tres últimos métodos puede utilizarse internamente un método privado más general llamado **añadir(...)** que añada una cantidad (positiva o negativa) al saldo.

También habrá un atributo común a todas las instancias llamado **interes_anual_basico**, que en principio puede ser constante.

La clase tiene que ser **abstracta** y debe tener un método **calcular_intereses()** que se dejará sin implementar.

También puede ser útil implementar un método para mostrar los datos de la cuenta.

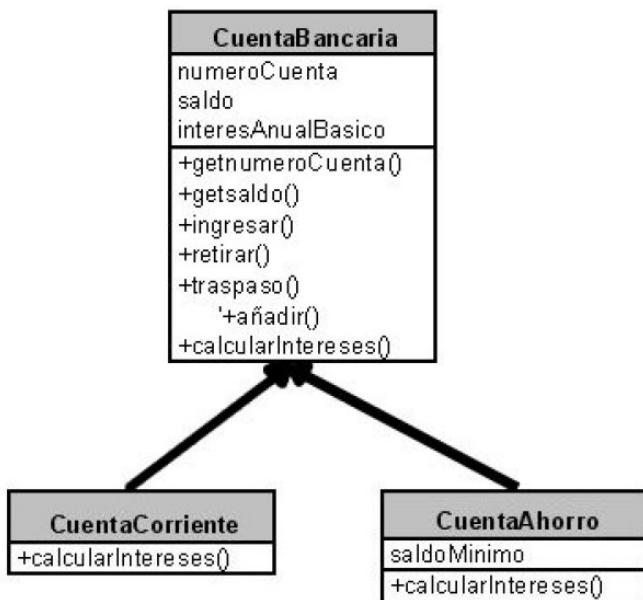
De esta clase heredarán dos subclases: **CuentaCorriente** y **CuentaAhorro**. La diferencia entre ambas será la manera de calcular los intereses:

- A la primera se le incrementará el saldo teniendo en cuenta el interés anual básico.
- La segunda tendrá una constante de clase llamada **saldo_minimo**. Si no se llega a este saldo el interés será la mitad del interés básico. Si se supera el saldo mínimo el interés aplicado será el doble del interés anual básico.

Implementa una clase principal con función main para probar el funcionamiento de las tres clases: Crea varias cuentas bancarias de distintos tipos, pueden estar en un list si lo deseas; prueba a realizar ingresos, retiradas

y transferencias; calcula los intereses y muéstralos por pantalla; etc.

El diagrama UML sería:



Apartado E – EMPRESA Y EMPLEADOS

Vamos a implementar dos clases que permitan gestionar datos de empresas y sus empleados.

Los **empleados** tienen las siguientes características:

- Un empleado tiene nombre, DNI, sueldo bruto (mensual), edad, teléfono y dirección.
- El nombre y DNI de un empleado no pueden variar.
- Es obligatorio que todos los empleados tengan al menos definido su nombre, DNI y el sueldo bruto. Los demás datos no son obligatorios.
- Será necesario un método para imprimir por pantalla la información de un empleado.
- Será necesario un método para calcular el sueldo neto de un empleado. El sueldo neto se calcula descontando del sueldo bruto un porcentaje que depende del IRPF. El porcentaje del IRPF depende del sueldo bruto anual del empleado (sueldo bruto x 12 pagas).

Sueldo bruto anual	IRPF
Inferior a 12.000 €	20%
De 12.000 a 25.000 €	30%
Más de 25.000 €	40%

Por ejemplo, un empleado con un sueldo bruto anual de 17.000 € tendrá un 30% de IRPF. Para calcular su sueldo neto mensual se descontará un 30% a su sueldo bruto mensual.

Las **empresas** tienen las siguientes características:

- Una empresa tiene nombre y CIF (datos que no pueden variar), además de teléfono, dirección y empleados. Cuando se crea una nueva empresa esta carece de empleados.
- Serán necesarios métodos para:
 - Añadir y eliminar empleados a la empresa.
 - Mostrar por pantalla la información de todos los empleados.
 - Mostrar por pantalla el DNI, sueldo bruto y neto de todos los empleados.
 - Calcular la suma total de sueldos brutos de todos los empleados.
 - Calcular la suma total de sueldos netos de todos los empleados.

Implementa las clases **Empleado** y **Empresa** con los atributos oportunos, un constructor, los getters/setters oportunos y los métodos indicados. Puedes añadir más métodos si lo ves necesario. Estas clases no deben realizar ningún tipo de entrada por teclado.

Implementa también una clase **App** con una función main para realizar pruebas: Crear una o varias empresas, crear empleados, añadir y eliminar empleados a las empresas, listar todos los empleados, mostrar el total de sueldos brutos y netos, etc.

Apartado F - VEHÍCULOS

Es muy aconsejable hacer el diseño UML antes de empezar a programar.

Debes crear varias clases para un software de una empresa de transporte. Implementa la jerarquía de clases necesaria para cumplir los siguientes criterios:

- Los vehículos de la empresa de transporte pueden ser terrestres, acuáticos y aéreos. Los vehículos terrestres pueden ser coches y motos. Los vehículos acuáticos pueden ser barcos y submarinos. Los vehículos aéreos pueden ser aviones y helicópteros.
- Todos los vehículos tienen matrícula y modelo (datos que no pueden cambiar). La matrícula de los terrestres debe estar formadas por 4 números y 3 letras. La de los vehículos acuáticos por entre 3 y 10 letras. La de los vehículos aéreos por 4 letras y 6 números.
- Los vehículos terrestres tienen un número de ruedas (dato que no puede cambiar).
- Los vehículos acuáticos tienen eslora (dato que no puede cambiar).
- Los vehículos aéreos tienen un número de asientos (dato que no puede cambiar).
- Los coches pueden tener aire acondicionado o no tenerlo.
- Las motos tienen un color.
- Los barcos pueden tener motor o no tenerlo.
- Los submarinos tienen una profundidad máxima.
- Los aviones tienen un tiempo máximo de vuelo.
- Los helicópteros tienen un número de hélices.
- No se permiten vehículos genéricos, es decir, no se deben poder instanciar objetos que sean vehículos sin más. Pero debe ser posible instanciar vehículos terrestres, acuáticos o aéreos genéricos (es decir, que no sean coches, motos, barcos, submarinos, aviones o helicópteros).
- El diseño debe obligar a que todas las clases de vehículos tengan un método imprimir() que imprima por pantalla la información del vehículo en una sola línea.

Implementa todas las clases necesarias con: atributos, constructor con parámetros, getters/setters y el método imprimir. Utiliza **abstracción** y **herencia** de la forma más apropiada.

Implementa también una clase Programa para hacer algunas pruebas: Instancia varios vehículos de todo tipo (coches, motos, barcos, submarinos, aviones y helicópteros) así como vehículos genéricos (terrestres, acuáticos y aéreos). Crea una lista y añade todos los vehículos. Recorre la lista y llama al método imprimir de todos los vehículos.

Apartado G - FIGURAS

Implementa una interface llamada **iFigura2D** que declare los métodos:

- ∞ perímetro(): Para devolver el perímetro de la figura
- ∞ área(): Para devolver el área de la figura
- ∞ escalar(escala): Para escalar la figura (aumentar o disminuir su tamaño). Solo hay que multiplicar los atributos de la figura por la escala (> 0).
- ∞ imprimir(): Para mostrar la información de la figura (atributos, perímetro y área) en una sola línea.

Existen 4 tipos de figuras.

- ∞ Cuadrado: Sus cuatro lados son iguales.
- ∞ Rectángulo: Tiene ancho y alto.
- ∞ Triángulo: Tiene ancho y alto.

- ∞ Círculo: Tiene radio.

Crea las 4 clases de figuras de modo que implementen la interface iFigura2D. Define sus métodos.

Crea una clase ProgramaFiguras con un main en el que realizar las siguientes pruebas:

1. Crea una lista de figuras.
2. Añade figuras de varios tipos.
3. Muestra la información de todas las figuras.
4. Escala todas las figuras con escala = 2.
5. Muestra de nuevo la información de todas las figuras.
6. Escala todas las figuras con escala = 0.1.

Muestra de nuevo la información de todas las figuras

Apartado H – Ejercicios de listas

- ∞ Crea un programa que pida diez números reales por teclado, los almacene en un array, y luego muestre todos sus valores.
- ∞ Crea un programa que pida diez números reales por teclado, los almacene en un array, y luego muestre la suma de todos los valores.
- ∞ Crea un programa que pida diez números reales por teclado, los almacene en un array, y luego lo recorra para averiguar el máximo y mínimo y mostrarlos por pantalla.
- ∞ Crea un programa que pida veinte números enteros por teclado, los almacene en un array y luego muestre por separado la suma de todos los valores pares e impares.
- ∞ Crea un programa que pida veinte números reales por teclado, los almacene en un array y luego lo recorra para calcular y mostrar la media: (suma de valores) / número de valores.
- ∞ Crea un programa que pida dos valores enteros N y M, luego cree un array de tamaño N, escriba M en todas sus posiciones y lo muestre por pantalla.
- ∞ Crea un programa que pida dos valores enteros P y Q, luego cree un array que contenga todos los valores desde P hasta Q, y lo muestre por pantalla.
- ∞ Crea un programa que cree un array con 100 números reales aleatorios entre 0.0 y 1.0, utilizando random(), y luego le pida al usuario un valor real R. Por último, mostrará cuántos valores del array son igual o superiores a R.
- ∞ Crea un programa que cree un array de enteros de tamaño 100 y lo rellene con valores enteros aleatorios entre 1 y 10. Luego pedirá un valor N y mostrará en qué posiciones del array aparece N.
- ∞ Crea un programa para realizar cálculos relacionados con la altura (en metros) de personas. Pedirá un valor N y luego almacenará en un array N alturas introducidas por teclado. Luego mostrará la altura media, máxima y mínima, así como cuántas personas miden por encima y por debajo de la media.
- ∞ Crea un programa que cree dos Arrays de enteros de tamaño 100. Luego introducirá en el primer array todos los valores del 1 al 100. Por último, deberá copiar todos los valores del primer array al segundo array en orden inverso, y mostrar ambos por pantalla.
- ∞ Crea un programa que cree un array de 10 enteros y luego muestre el siguiente menú con distintas opciones:
 - Mostrar valores.
 - Introducir valor.
 - Salir.
 - La opción ‘a’ mostrará todos los valores por pantalla. La opción ‘b’ pedirá un valor V y una posición P, luego escribirá V en la posición P del array. El menú se repetirá indefinidamente hasta que el usuario elija la opción ‘c’ que terminará el programa.
- ∞ Crea un programa que permita al usuario almacenar una secuencia aritmética en un array y luego mostrarla. Una secuencia aritmética es una serie de números que comienza por un valor inicial V, y continúa con incrementos de I. Por ejemplo, con V=1 e I=2, la secuencia sería 1, 3, 5, 7, 9... Con V=7

- e I=10, la secuencia sería 7, 17, 27, 37... El programa solicitará al usuario V, I además de N (número de valores a crear).
- ∞ Crea un programa que cree un array de enteros e introduzca la siguiente secuencia de valores: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, etc. hasta introducir 10 diez veces, y luego la muestre por pantalla.
 - ∞ Crea un programa que pida al usuario 20 valores enteros e introduzca los 10 primeros en un array y los 10 últimos en otro array. Por último, comparará ambos Arrays y le dirá al usuario si son iguales o no.
 - ∞ Crea un programa que cree un array de tamaño 30 y lo rellene con valores aleatorios entre 0 y 9. Luego ordena los valores del array y los mostrará por pantalla.
 - ∞ Necesitamos crear un programa para mostrar el ranking de puntuaciones de un torneo de ajedrez con 8 jugadores. Se le pedirá al usuario que introduzca las puntuaciones de todos los jugadores (habitualmente valores entre 1000 y 2800, de tipo entero) y luego muestre las puntuaciones en orden descendente (de la más alta a la más baja).
 - ∞ Crea un programa que cree un array de tamaño 1000 y lo rellene con valores enteros aleatorios entre 0 y 99. Luego pedirá por teclado un valor N y se mostrará por pantalla si N existe en el array, además de cuantas veces.

Listas Multidimensionales

- ∞ Crea un programa que cree una matriz de tamaño 5 x 5 que almacene los números del 1 al 25 y luego muestre la matriz por pantalla.
- ∞ Crea un programa que cree una matriz de 10 x 10 e introduzca los valores de las tablas de multiplicar del 1 al 10 (cada tabla en una fila). Luego mostrará la matriz por pantalla.
- ∞ Crea un programa que cree una matriz de tamaño N x M (tamaño introducido por teclado) e introduzca en ella N x M valores (también introducidos por teclado). Luego deberá recorrer la matriz y al final mostrar por pantalla cuántos valores son mayores que cero, cuántos son menores que cero y cuántos son igual a cero.
- ∞ Necesitamos crear un programa para almacenar las notas de 4 alumnos (llamados "Alumno 1", "Alumno 2", etc.) y 5 asignaturas. El usuario introducirá las notas por teclado y luego el programa mostrará la nota mínima, máxima y media de cada alumno.
- ∞ Necesitamos crear un programa para registrar sueldos de hombres y mujeres de una empresa y detectar si existe brecha salarial entre ambos. El programa pedirá por teclado la información de N personas distintas (valor también introducido por teclado). Para cada persona, pedirá su género (0 para varón y 1 para mujer) y su sueldo. Esta información debe guardarse en una única matriz. Luego se mostrará por pantalla el sueldo medio de cada género.

Apartado I – Ejercicios de diccionarios

Ejercicio 1

Escribe un programa que pida un número por teclado y que cree un diccionario cuyas claves sean desde el número 1 hasta el número indicado, y los valores sean los cuadrados de las claves.

Ejercicio 2

Escribe un programa que lea una cadena y devuelva un diccionario con la cantidad de apariciones de cada carácter en la cadena.

Ejercicio 3

Vamos a crear un programa donde vamos a declarar un diccionario para guardar los precios de las distintas frutas. El programa pedirá el nombre de la fruta y la cantidad que se ha vendido y nos mostrará el precio final de la fruta a partir de los datos guardados en el diccionario. Si la fruta no existe nos dará un error. Tras cada consulta el programa nos preguntará si queremos hacer otra consulta.

Ejercicio 4

Codifica un programa que nos permita guardar los nombres de los alumnos de una clase y las notas que han obtenido. Cada alumno puede tener distinta cantidad de notas. Guarda la información en un diccionario cuyas claves serán los nombres de los alumnos y los valores serán listas con las notas de cada alumno.

El programa pedirá el número de alumnos que vamos a introducir, pedirá su nombre e irá pidiendo sus notas hasta que introduzcamos un número negativo. Al final el programa nos mostrará la lista de alumnos y la nota media obtenida por cada uno de ellos. Nota: si se introduce el nombre de un alumno que ya existe el programa nos dará un error.

Ejercicio 5

Escribir un programa que implemente una agenda. En la agenda se podrán guardar nombres y números de teléfono. El programa nos dará el siguiente menú:

Añadir/modificar: Nos pide un nombre. Si el nombre se encuentra en la agenda, debe mostrar el teléfono y, opcionalmente, permitir modificarlo si no es correcto. Si el nombre no se encuentra, debe permitir ingresar el teléfono correspondiente.

Buscar: Nos pide una cadena de caracteres, y nos muestra todos los contactos cuyos nombres comiencen por dicha cadena.

Borrar: Nos pide un nombre y si existe nos preguntará si queremos borrarlo de la agenda.

Listar: Nos muestra todos los contactos de la agenda.

Ejercicio 6

Escribir un programa que guarde en una variable el diccionario {'Euro':'€', 'Dollar':'\$', 'Yen':'¥'}, pregunte al usuario por una divisa y muestre su símbolo o un mensaje de aviso si la divisa no está en el diccionario.

Ejercicio 7

Escribir un programa que pregunte al usuario su nombre, edad, dirección y teléfono y lo guarde en un diccionario. Después debe mostrar por pantalla el mensaje <nombre> tiene <edad> años, vive en <dirección> y su número de teléfono es <teléfono>. El proceso se debe poder realizar para varios usuarios. Crea un pequeño menú de gestión.

Ejercicio 8

Escribir un programa que pregunte una fecha en formato dd/mm/aaaa y muestre por pantalla la misma fecha en formato dd de <mes> de aaaa donde <mes> es el nombre del mes. Se guardarán en el diccionario todas las fechas introducidas.

Ejercicio 9

Escribir un programa que cree un diccionario vacío y lo vaya llenado con información sobre una persona (por ejemplo: nombre, edad, sexo, teléfono, correo electrónico, etc.) que se le pida al usuario. Cada vez que se añada un nuevo dato debe imprimirse el contenido del diccionario.

Ejercicio 10

Escribir un programa que cree un diccionario simulando una cesta de la compra. El programa debe preguntar el artículo y su precio y añadir el par al diccionario, hasta que el usuario decida terminar. Después se debe mostrar por pantalla la lista de la compra y el coste total, con el siguiente formato

Lista de la compra

Artículo 1 Precio

Artículo 2 Precio

Artículo 3 Precio

... ...

Total Coste

Ejercicio 11

Escribir un programa que cree un diccionario de traducción español-inglés. El usuario introducirá las palabras en español e inglés separadas por dos puntos, y cada par <palabra>:<traducción> separados por comas. El programa debe crear un diccionario con las palabras y sus traducciones. Después pedirá una frase en español y utilizará el diccionario para traducirla palabra a palabra. Si una palabra no está en el diccionario debe dejarla sin traducir.

Ejercicio 12

Escribir un programa que gestione las facturas pendientes de cobro de una empresa. Las facturas se almacenarán en un diccionario donde la clave de cada factura será el número de factura y el valor el coste de la factura. El programa debe preguntar al usuario si quiere añadir una nueva factura, pagar una existente o terminar. Si desea añadir una nueva factura se preguntará por el número de factura y su coste y se añadirá al diccionario. Si se desea pagar una factura se preguntará por el número de factura y se eliminará del diccionario. Después de cada operación el programa debe mostrar por pantalla la cantidad cobrada hasta el momento y la cantidad pendiente de cobro.

Apartado J – exercism.io

Apuntarse y realizar ejercicios de <https://exercism.io/>

U. T. 7 Proyecto.



Objetivo

Desarrollar un pequeño videojuego educativo para adolescentes llamado 'Star Wars Códigos Secretos' que fomente la comprensión lectora y la competencia matemática. Estará ambientado en el universo de Star Wars con un estilo de aventura conversacional sencillo en el que el usuario tendrá que superar varias pruebas matemáticas para conseguir destruir la estrella de la muerte.

Descripción del videojuego

- ∞ El videojuego es una aventura conversacional clásica en la que se muestra texto por pantalla (para contar lo que sucede en la aventura y mostrar la pregunta de cada prueba) y el usuario interactúa escribiendo por teclado para intentar superar cada prueba (introduciendo la respuesta correcta).
- ∞ La aventura estará estructurada en etapas (Inicio, Nivel 1, Nivel 2... Fin). En el apartado 3 de este documento se explica en detalle qué texto se ha de mostrar en cada etapa, así como las preguntas y respuestas. Es decir, el videojuego ya está pensado y diseñado, vuestro trabajo consiste en programarlo siguiendo el diseño y la descripción de este documento.
- ∞ El videojuego comienza siempre por la etapa Inicio en la que se mostrará un texto por pantalla (la introducción a la aventura) y se pasa al Nivel 1 (se sigue contando la historia y se plantea una prueba, una pregunta). Si el usuario responde correctamente pasa al Nivel 2 (sigue la historia y se plantea otra pregunta), etc. Así nivel tras nivel hasta superar el último y pasar a la etapa Ganar (se muestra un texto contando el final feliz y el juego termina).
- ∞ Si en algún nivel el usuario contesta mal a la pregunta, se pasa directamente a la etapa Perder (se muestra un texto y el juego termina).
- ∞ Las preguntas utilizan números aleatorios para que cada vez que juegues se utilicen números distintos y no sean siempre iguales (ver apartado 4).

Descripción de las etapas

Inicio

Texto a mostrar === STAR WARS CÓDIGOS SECRETOS ===

Hace mucho tiempo, en una galaxia muy, muy lejana... La Princesa Leia, Luke Skywalker, Han Solo, Chewbacca, C3PO y R2D2 viajan en una nave imperial robada en una misión secreta para infiltrarse en otra estrella de la muerte que el imperio está construyendo para destruirla. (Presiona Intro para continuar)

Acción Tras presionar Intro -> Nivel 1

Nivel 1

Texto a mostrar Los problemas empiezan cuando deben realizar un salto hiperespacial hasta al sistema S1 en el sector S2, pero el sistema de navegación está estropeado y el computador tiene problemas para calcular parte de las coordenadas de salto. Chewbacca, piloto experto, se da cuenta que falta el cuarto número de la serie. Recuerda de sus tiempos en la academia de pilotos que para calcularlo hay que calcular el sumatorio entre el número del sistema y el número del sector (ambos inclusive). ¿Qué debe introducir?

Variables S1: Número entero aleatorio entre 1 y 10. S2: Número entero aleatorio entre 20 y 30.

Respuesta Sumatorio desde S1 hasta S2. Por ejemplo, con S1=10, S2=20, la respuesta correcta sería $10+11+12+13+14+15+16+17+18+19+20 = 165$.

Acción Si correcto -> Nivel 2. Si no -> Perder

Nivel 2

Texto a mostrar Gracias a Chewbacca consiguen llegar al sistema correcto y ven a lo lejos la estrella de la muerte. Como van en una nave imperial robada se aproximan lentamente con la intención de pasar desapercibidos. De repente suena el comunicador. "Aquí agente de espaciopuerto P1 contactando con nave imperial P2. No están destinados en este sector. ¿Qué hacen aquí?". Han Solo coge el comunicador e improvisa. "Eh... tenemos un fallo en el... eh... condensador de flujo... Solicitamos permiso para atracar y reparar la nave". El agente, que no se anda con tonterías, responde "Proporcione código de acceso o abriremos fuego". Han Solo ojea rápidamente el manual del piloto que estaba en la guantera y da con la página correcta. El código es el productorio entre el número del agente y el número de la nave (ambos inclusive).

¿Cuál es el código?

Variables P1: Número entero aleatorio entre 1 y 7. P2: Número entero aleatorio entre 8 y 12.

Respuesta Productorio entre P1 y P2. Por ejemplo, con P1=5, P2=10, la respuesta correcta sería $5*6*7*8*9*10 = 151200$.

Acción Si correcto -> Nivel 3. Si no -> Perder

Nivel 3

Texto a mostrar Han Solo proporciona el código correcto. Atracan en la estrella de la muerte, se equipan con trajes de soldados imperiales que encuentran en la nave para pasar desapercibidos y bajan. Ahora deben averiguar en qué nivel de los N existentes se encuentra el reactor principal. Se dirigen al primer panel computarizado que encuentran y la Princesa Leia intenta acceder a los planos de la nave, pero necesita introducir una clave de acceso. Entonces recuerda la información que le proporcionó Lando Calrissian "La clave de acceso a los planos de la nave es el factorial de N/10 (redondeando N hacia abajo), donde N es el número de niveles".

¿Cuál es el nivel correcto?

Variables N: Número entero aleatorio entre 50 y 100.

Respuesta Factorial de N/10, redondeando hacia abajo. Por ejemplo si N=78, entonces la respuesta correcta sería $7! = 1*2*3*4*5*6*7 = 5040$.

Acción Si correcto -> Nivel 4. Si no -> Perder

Nivel 4

Texto a mostrar Gracias a la inteligencia de Leia llegan al nivel correcto y encuentran la puerta acorazada que da al reactor principal. R2D2 se conecta al panel de acceso para intentar hackear el sistema y abrir la puerta. Para desencriptar la clave necesita verificar si el número P es primo o no. Si es primo introduce un 1, si no lo es introduce un 0.

Variables P: Número entero aleatorio entre 10 y 100.

Respuesta 1 si P es primo, 0 en caso contrario.

Por ejemplo, si P=11 como 11 es primo se introduce un 1.

Acción Si correcto -> Nivel 5. Si no -> Perder

Nivel 5

Texto a mostrar Consiguen entrar al reactor. Ya solo queda que Luke Skywalker coloque la bomba, programe el temporizador y salir de allí corriendo. Necesita programarlo para que explote en exactamente M minutos y S segundos, el tiempo suficiente para escapar antes de que explote, pero sin que el sistema de seguridad anti-explosivos detecte y desactive la bomba. Pero el temporizador utiliza un reloj Zordgiano un tanto

peculiar. Para convertir los minutos y segundos al sistema Zordgiano hay que sumar el factorial de M y el factorial de S. ¿Qué valor debe introducir?

Variables M: Número entero aleatorio entre 5 y 10. S: Número entero aleatorio entre 5 y 10.

Respuesta Si por ejemplo M = 7 y S = 5, habría que calcular $(1*2*3*4*5*6*7) + (1*2*3*4*5)$ que sería $5040 + 120 = 5160$

Acción Si correcto -> Ganar. Si no -> Perder

Ganar

Texto a mostrar Luke Skywalker introduce el tiempo correcto, activa el temporizador y empiezan a sonar las alarmas. Salen de allí corriendo, no hay tiempo que perder. La nave se convierte en un hervidero de soldados de arriba a abajo y entre el caos que les rodea consiguen llegar a la nave y salir de allí a toda prisa. A medida que se alejan observan por la ventana la imagen de la colossal estrella de la muerte explotando en el silencio del espacio, desapareciendo para siempre junto a los restos del malvado imperio.

¡Has salvado la galaxia gracias a la Fuerza Jedi de las matemáticas! Enhorabuena ;D

Acción Se pasa automáticamente a la etapa Fin

Perder

Texto a mostrar Ese no era el código correcto... La misión ha sido un fracaso... :(:(

Todavía no eres un Maestro Jedi de las Matemáticas. ¡Vuelve a intentarlo!

Acción Se pasa automáticamente a la etapa Fin

Fin

Texto a mostrar Gracias por jugar :D

Acción Ninguna. El programa termina.

Aleatoriedad de las preguntas

Para que el juego sea más interesante se utilizarán números aleatorios en las preguntas. Así, aunque la historia sea la misma, en cada partida la respuesta correcta será diferente.

Para ello necesitarás utilizar el método random() que proporciona un número pseudoaleatorio entre 0.0 y 1.0. o choice() que elige entre un conjunto de opciones

U. T. 8 Gestión de datos.



Gestión de ficheros

Qué es un fichero

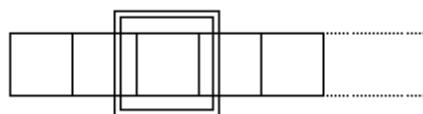
Todas las estructuras de datos que hemos visto hasta ahora utilizan memoria principal. Esto tiene dos limitaciones importantes:

- ∞ Los datos desaparecen cuando el programa termina.
- ∞ La cantidad de los datos no puede ser muy grande debido a la limitación de la memoria principal.

Por eso existen también estructuras especiales que utilizan memoria secundaria: los ficheros. El fichero es además una estructura dinámica, en el sentido de que su tamaño puede variar durante la ejecución del programa dependiendo de la cantidad de datos que tenga.

Tipos de acceso

Al estar en memoria secundaria, no todos los elementos del fichero son accesibles de forma inmediata. Solamente se puede acceder cada vez a un único elemento del fichero, que se denomina puntero del fichero. Dependiendo de cómo se desplaza por el fichero, podemos distinguir dos tipos de acceso:



- ∞ **Acceso secuencial:** El puntero del fichero sólo puede moverse hacia delante a partir del primer elemento y siempre de uno en uno.
- ∞ **Acceso directo:** El puntero del fichero se puede situar directamente en cualquier posición del fichero. Es un acceso similar al utilizado en los Arrays unidimensionales.

El acceso directo suele ser más eficiente, ya que para leer un dato no hace falta leer antes todos los anteriores.

Ficheros binarios y ficheros de texto

Existen dos tipos principales de ficheros:

- ∞ **Ficheros de texto:** Contienen secuencias de caracteres separadas por saltos de línea (tipo str). El teclado (entrada estándar) y la pantalla (salida estándar) se consideran también ficheros de texto. Al leer o escribir variables de un fichero de texto se pueden realizar ciertas conversiones. Por ejemplo, cuando escribimos un entero con valor 10, este entero se convierte en los caracteres '1' y '0'.
- ∞ **Ficheros binarios:** Contienen secuencias de bytes. Los elementos se almacenan en el fichero exactamente igual que están almacenados en memoria principal, es decir, al leer o escribir no se realiza ningún tipo de conversión.

Modo de acceso a los ficheros

Cuando intentamos acceder a un fichero existen dos únicas acciones posibles: leer o escribir, según estas acciones se pueden definir en ficheros de entrada (abiertos para lectura), ficheros de salida (abiertos para escritura) o ficheros de lectura / escritura (abiertos para ambas operaciones).

Los ficheros residen en un sistema de archivos y tendremos que saber la ruta de acceso al mismo para poder utilizarlo, dependiendo del SSOO de base los separadores cambian: / o \, con lo que recomendamos utilizar las herramientas de los lenguajes o librerías asociadas para crear estas rutas de acceso, evitando trabajar con características específicas de los SSOO.

Procesamiento de un fichero

Siempre que queramos realizar cualquier operación con ficheros se debe seguir el siguiente esquema:

Apertura de Fichero → Operaciones → Cierre del fichero

Es importante seguir el esquema y sobre todo no olvidar el cierre del fichero, si no queremos perder datos. Los SSOO no escriben inmediatamente los datos de memoria a disco, utilizan un buffer intermedio por eficiencia, si no cerramos el fichero no nos aseguramos que este buffer se descargue a disco y podremos perder las últimas operaciones.

Qué son los flujos

Los archivos o ficheros almacenados se tienen que tratar por parte del SSOO y de Python. Para esa gestión se crea un objeto especial en memoria. Por tanto, un flujo no es más que un objeto de control que gestiona un fichero en disco, creado en el momento de la apertura y destruido cuando se cierra el archivo. El flujo representa al archivo dentro de nuestro programa y da acceso a sus datos.

Bajo Python no se habla de flujos, se asocia más a lenguajes como C++ o Java, se utiliza directamente fichero o archivo, pero este concepto también existe y hay paquetes que lo implementan: **io** (<https://docs.python.org/es/3/library/io.html>) que utilizaría conceptos similares al lenguaje C++ o Java.

Acceso a ficheros de Texto y Binarios

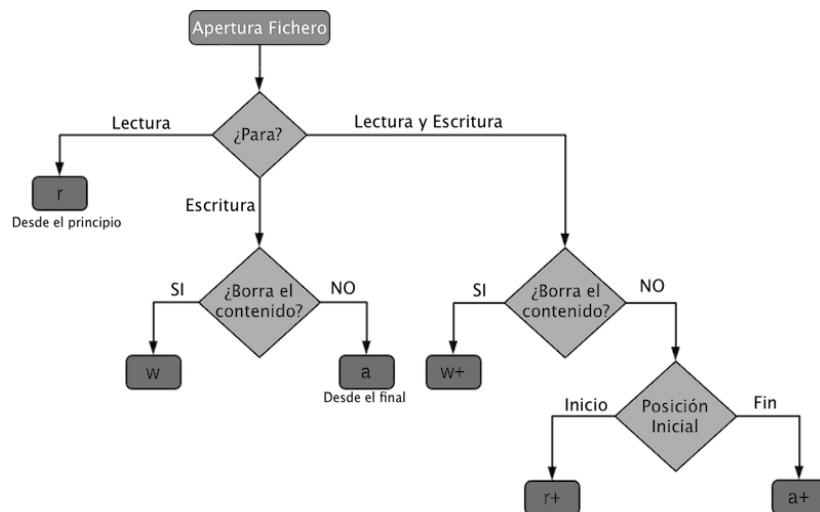
Apertura de ficheros

Python tiene una función **open()** incorporada para abrir un archivo. Esta función devuelve un objeto archivo, también llamado identificador, ya que se usa para leer o modificar el archivo en consecuencia (sería similar al concepto del lenguaje C).

```
f = open("test.txt")      # open file in current directory
f = open("C:/Python38/README.txt")  # specifying full path
```

Modos de acceso.

Podemos especificar el modo al abrir un archivo en el momento de su apertura. En el modo, especificamos si queremos leer (**r**), escribir (**w**) o agregar (**a**) al archivo. También podemos especificar si queremos abrir el archivo en modo texto (**t**) o en modo binario (**b**).



t Se abre en modo texto, no es necesario especificarlo. (default)

b Se abre en modo binario

El valor predeterminado es leer en modo texto (**rt**). En este modo, obtenemos cadenas (**str**) al leer del archivo. Por otro lado, el modo binario devuelve bytes (**bytes**) y este es el modo que se debe utilizar cuando se trata de archivos que no son de texto, como imágenes o archivos ejecutables.

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt", 'w')  # write in text mode
f = open("img.bmp", 'r+b') # read and write in binary mode
```

Python no utiliza en la lectura y escritura una codificación estricta, por lo que debemos especificarla en el momento de la apertura si no queremos tener problemas. De hecho, la codificación por defecto que se usará si no la especificamos es dependiente de la plataforma, así en Windows utilizará **cp1252** pero bajo Linux se utilizará **utf-8**.

Debemos especificar la codificación correcta o no se mostrarán los caracteres de forma adecuada, por lo que si el fichero no lo hemos creado nosotros habrá que encontrar la codificación y usarla. Cuando creemos nosotros los ficheros utilizaremos siempre que sea posible **utf-8** independientemente de la plataforma en la que se ejecute.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

Qué es la codificación de un fichero.

Cuando queremos representar el alfabeto y sus correspondientes letras que tenemos en el ordenador, hay que asignar a cada letra un valor numérico. La cantidad de letras que podemos usar dependerá del número de bits que usemos en la codificación. Por ejemplo, la codificación ASCII estándar utiliza 8 bits, lo que significa que solo podrá representar como máximo 256 letras, algunas menos ya que los 32 primeros códigos se utilizan para control.

El número de idiomas presentes en el mundo hizo que con estos 256 valores no se pudieran representar todos los guarismos y se hicieran conjuntos de ellos según el idioma. Así aparecieron las diferentes codificaciones que se determinan por el SSOO en los ajustes del mismo: latin1, cp1252, 8859-1, 8859-15. Cada uno de ellos asigna valores diferentes a sus símbolos y mismos símbolos pueden estar en posiciones diferentes, con lo que al intentar representarlos en pantalla no se corresponda (Las diferencias de las páginas de código estaban en los 128 caracteres superiores, los inferiores del código no cambiaban).

Con el tiempo la codificación ASCII se quedó obsoleta y no daba cabida a todos los caracteres en algunas lenguas con lo que se implementó una nueva codificación: UTF, esta permite que los bits a utilizar sean de 8, 16 o 32 bits pudiéndose representar todos los símbolos de todos los lenguajes en esta nueva representación.

No vamos a profundizar más en UTF, pero el estándar que más se está usando hoy en día es UTF-8, por lo que nuestros ficheros deben tener esa codificación.

Resumen: La codificación es la manera de asignar a cada valor numérico almacenado en modo texto la grafía que le corresponde, si usamos una tabla distinta a la que se usó para crearlo no veremos el texto de forma adecuada.

Cierre de ficheros

Cuando terminemos de realizar operaciones en el archivo, debemos cerrarlo correctamente. Cerrar un archivo liberará los recursos que estaban vinculados con el archivo. Se hace usando el método **close()** disponible en Python. Python tiene un recolector de basura para limpiar objetos sin referencia, pero no debemos confiar en él para cerrar el archivo.

```
f = open("test.txt", encoding = 'utf-8')  
# realizar operaciones de lectura escritura  
f.close()
```

Queda una cuestión: si en el programa se produce alguna excepción, el fichero no se cerrará correctamente y podemos crear malos funcionamientos, con lo que el protocolo que debemos usar para la gestión de archivos es el que mostramos a continuación.

```

try:
    f = open("test.txt", encoding = 'utf-8')
    # realizar operaciones de lectura escritura
finally:
    f.close()

```

Uso de with con ficheros

Hemos visto el protocolo estándar y los problemas que nos puede producir el no capturar correctamente las excepciones. Para evitar estas situaciones se ha implementado el uso de la palabra reservada **with**, que se encargará de cerrar adecuadamente el fichero y tratar las excepciones si ocurrieran.

```

with open("test.txt", encoding = 'utf-8') as f:
    # realizar operaciones de lectura escritura

```

Escritura y lectura de información en ficheros

Python trabaja con los ficheros de forma similar al C, por lo que no les dota de una estructura y es difícil la actualización y manejo de bases de datos de ficheros.

Para escribir en un archivo en Python, necesitamos abrirlo en escritura **w**, agregar **a** o modo exclusivo **x** creación. Debemos tener cuidado con el modo **w**, ya que se sobrescribirá en el archivo si ya existe. Debido a esto, se borran todos los datos anteriores.

La escritura de una cadena o secuencia de bytes (para archivos binarios) se realiza mediante el método **write()**. Este método devuelve el número de caracteres escritos en el archivo.

Debemos incluir los caracteres de nueva línea nosotros mismos para distinguir las diferentes líneas.

```

with open("test.txt", 'w', encoding = 'utf-8') as f:
    f.write("my first file\n")
    f.write("This file\n\n")
    f.write("contains three lines\n")

```

Para leer un archivo en Python, debemos abrir el archivo en modo (**r**) de lectura. Hay varios métodos disponibles para este propósito. Podemos usar el método de **read(tamaño)** para leer el número de tamaño de los datos. Si no se especifica el parámetro de tamaño, se lee y regresa hasta el final del archivo.

```

f = open("test.txt", 'r', encoding = 'utf-8')
f.read(4)      # read the first 4 data
f.read(4)      # read the next 4 data
f.read()       # read in the rest till end of file
f.read()       # further reading returns empty sting

```

Una vez que se llega al final del archivo, obtenemos una cadena vacía en la lectura adicional.

Podemos cambiar el cursor de lectura usando la instrucción **seek(posicion)**. También podemos determinar la posición actual mediante **tell()**. Ambos métodos trabajan a nivel de bytes.

```

f.tell()      # get the current file position
f.seek(0)     # bring file cursor to initial position

```

Ejemplo de acceso

```

for line in f:
    print(line, end = '')

```

En ficheros de texto podemos usar el método **readline()** para leer líneas individuales de un archivo. Este método lee un archivo hasta el salto de línea, incluido el carácter de salto de línea. O podemos usar el método **readlines()** que devuelve una lista con las líneas restantes de todo el archivo. Todos estos métodos de lectura devuelven valores vacíos cuando se alcanza el final del archivo (EOF).

```
f.readline()
f.readlines()
```

Resumen

Método	Descripción
close()	Cierra un fichero, no tiene efecto si ya lo está
flush()	Vacía el buffer subyacente del flujo
read(n)	Lee al menos n caracteres del fichero. En caso de None o negativo lee hasta el final del fichero.
readline(n=-1)	Lee una línea del fichero leyendo al menos n bytes.
readlines(n=-1)	Lee un conjunto de líneas del fichero.
seek(offset,from=SEEK_SET)	Establece la nueva posición del puntero de lectura – escritura a partir de un punto.
tell()	Devuelve la posición del puntero de lectura.
write(s)	Escribe la cadena en el fichero.
writelines(lines)	Escribe un conjunto de líneas en el fichero

Serialización

La serialización es el proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria), con el fin de transmitirlo a través de una conexión en red o almacenarlo en un fichero, como una serie de bytes o en un formato humanamente más legible como XML o JSON, entre otros. La serie de bytes o el formato pueden ser usados para crear un nuevo objeto que es idéntico en todo al original, incluido su estado interno (por tanto, el nuevo objeto es un clon del original).

El módulo **pickle** implementa un algoritmo para convertir un objeto arbitrario Python en una serie de bytes. El flujo de bytes que representa al objeto puede ser transmitido o almacenado y luego reconstruido para crear un nuevo objeto con las mismas características.

```
import pickle
import pprint

data1 = [{ 'a': 'A', 'b': 2, 'c': 3.0}]
print('BEFORE: ', end=' ')
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

# data1_string puede ser transmitido o almacenado en disco

data2 = pickle.loads(data1_string)
print('AFTER : ', end=' ')
pprint.pprint(data2)

print('SAME? :', (data1 is data2))
print('EQUAL?:', (data1 == data2))
```

No todos los objetos pueden ser serializados: Sockets, gestores de archivos, conexiones de bases de datos, y otros objetos con estado de ejecución que dependen del sistema operativo, u otro proceso puede no ser posible guardarlo. Los objetos que tienen atributos no serializables pueden definir **__getstate__()** y **__setstate__()** para devolver un subconjunto del estado de la instancia a ser serializada.

El protocolo pickle maneja automáticamente referencias circulares entre objetos, por lo que las estructuras de datos complejas no necesitan ningún manejo especial.

Utilización de los sistemas de ficheros

Al igual que podemos gestionar el contenido de ficheros, podemos gestionar los ficheros desde el nivel del sistema operativo. A este nivel podremos crear directorios, mover ficheros, etc. Veamos un ejemplo de las posibilidades que existen en la librería **os**, **pathlib** y la librería **shutil**.

- ∞ import os. Uso de la librería (No se recomienda su uso)
 - os.chdir(path), cambio de path actual.
 - os.getcwd(), path actual.
 - os.path, librería para manipulación de rutas a bajo nivel.
 - os.listdir(path='.'). Retorna la lista de ficheros y directorios de una ruta.
 - os.mkdir(path, mode=0o777, *, dir_fd=None). Crear un directorio.
 - os.rmdir(path, *, dir_fd=None). Borra un directorio vacío.
 - os.remove(path, *, dir_fd=None). Borrar un fichero.
 - os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None). Renombrar.
- ∞ Import pathlib, librería para manipulación de rutas a nivel de objetos independientemente del SSOO.
<https://docs.python.org/3/library/pathlib.html>
- ∞ import shutil. Uso de la librería
 - shutil.copyfile(src, dst, *, follow_symlinks=True), copiar un fichero usando descriptores.
 - shutil.copy(src, dst, *, follow_symlinks=True), copiar un fichero usando rutas.
 - shutil.copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2,
 - ignore_dangling_symlinks=False). Copia recursivamente un directorio entero.
 - shutil.rmtree(path, ignore_errors=False, onerror=None). Borra un directorio entero.
 - shutil.move(src, dst, copy_function=copy2). Mueve un fichero o directorio a otro sitio.

Tenemos que tener mucho cuidado con el uso de estas instrucciones ya que dependen del sistema operativo en las que las ejecutemos, por ejemplo, los parámetros **mode** de algunas instrucciones solo se utilizarán bajo Linux, no sobre Windows.

Ejemplo

Desarrolla

- Dado un fichero con una palabra por línea, crear un fichero de salida con las palabras ordenadas.
- Dados dos ficheros con una palabra por línea, crear un fichero de salida que sea la concatenación de las correspondientes líneas de cada fichero.
- Realiza la aplicación code_0009_0 (Listas que hacer TODO almacenadas)

Ejemplo

Desarrolla (Después de hacer los ejercicios de ficheros del final de la unidad).

Crear una aplicación que gestione las citas de una consulta veterinaria, que se puedan almacenar en disco todos los datos. Haz el desarrollo completo, desde UML hasta la implementación final, utiliza tres ficheros: para las citas, para los datos de los clientes y para las mascotas.

La carga de los datos se realizará al comienzo de la aplicación desde los ficheros, se realizarán todas las modificaciones en memoria y cuando se cierre la aplicación salvará el estado a disco, no utilizará ficheros en el tratamiento intermedio de la información.

code_0009_a

Bases de Datos

Una base de datos es un software que te permite almacenar de cualquier tipo de manera ordenada y estructurada, de forma que, cuando se vaya a consultar podamos escoger los diferentes fragmentos de todos esos datos almacenados y podamos consultar los que a nosotros nos interesan.

Una base de datos relacional es un tipo de base de datos que almacena y proporciona acceso a datos relacionados entre sí. Las bases de datos relacionales se basan en el modelo relacional. El modelo relacional implica que las estructuras lógicas de los datos (las tablas, las vistas y los índices) estén separadas de las estructuras de almacenamiento físico. Gracias a esta separación, los administradores de bases de datos pueden gestionar el almacenamiento físico de datos sin que eso influya en el acceso a esos datos como estructura lógica. El modelo relacional es sencillo pero muy potente (Oracle, MySql, Postgre, Db2, SQLite).

El modelo de base de datos orientado a objetos agrupa la información en paquetes relacionados entre sí: los datos de cada registro se combinan en un solo objeto, con todos sus atributos. De esta manera, toda la información está disponible en el objeto, ya que sus datos quedan agrupados en lugar de distribuidos en diferentes tablas. En los objetos no solo pueden guardarse los atributos, sino también los métodos, lo que refleja la afinidad de estas bases de datos con los lenguajes de programación orientados a objetos: al igual que en estos, cada objeto presenta un conjunto de acciones que pueden llevarse a cabo. Uno de los inconvenientes de este modelo es que su uso está poco extendido, pero es igual de potente que el modelo relacional.

Una base de datos documental, también llamada una base de datos orientada a documentos o tienda de documentos, es un subconjunto de un tipo de base de datos no SQL. A diferencia de las bases de datos relacionales tradicionales, el modelo de datos en una base de datos de documentos no está estructurado en un formato de tabla de filas y columnas. El esquema puede variar, proporcionando mucha más flexibilidad para el modelado de datos que las bases de datos relacionales. Las bases de datos documental almacenan cada registro y sus datos asociados en un solo documento. Cada documento contiene datos semiestructurados que pueden ser consultados con el uso de varias herramientas de consulta y análisis del DBMS. Las BBDD de documentos ofrecen importantes ventajas cuando se requieren características específicas, que incluyen: Modelado flexible de datos, Rendimiento de escritura rápido, Rendimiento rápido de consultas (MongoDB, DynamoDB, Couchbase, Azure CosmosDB).

Clases para el acceso a BD relacionales

Python tiene acceso a gran número de bases datos, la inmensa mayoría están implementadas como un paquete Python que se puede importar y usar. Para un correcto funcionamiento debemos leer cuidadosamente la documentación de cada paquete y ver los ejemplos. Dentro de los SGBD más populares encontramos:

- ∞ SQL Server: pymssql y pyodbc.
- ∞ Oracle: cx_Oracle.
- ∞ MySQL: Mysql.connector.
- ∞ SQLite: sqlite3.
- ∞ PostgreSQL: psycopg2.

Establecimiento de conexiones con diferentes SGBD

En Python, como en otros lenguajes, existe una propuesta de API estándar para el manejo de bases de datos, de forma que el código sea prácticamente igual independientemente de la base de datos que estemos utilizando por debajo. Esta especificación recibe el nombre de Python Database API o DB-API y se recoge en el PEP 249 (<http://www.python.org/dev/peps/pep-0249/>).

En el entorno educativo la mayoría de veces se usa MySql como servidor de desarrollo, con lo que vamos a realizar los ejemplos sobre esa base de datos, para tal fin instalaremos el servidor desde la página de descargas (<https://www.mysql.com/downloads/>) y crearemos una base de datos con la siguiente estructura.

```

CREATE DATABASE IF NOT EXISTS `daw_prog` DEFAULT CHARACTER SET
utf8 COLLATE utf8_spanish2_ci;
USE `daw_prog`;

DROP TABLE IF EXISTS `personas`;
CREATE TABLE `personas` (
  `cod` int(11) NOT NULL,
  `nombre` varchar(255) COLLATE utf8_spanish2_ci NOT NULL,
  `apellido_1` varchar(255) COLLATE utf8_spanish2_ci NOT NULL,
  `apellido_2` varchar(255) COLLATE utf8_spanish2_ci NOT NULL,
  `fecha_nacimiento` date NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish2_ci;

ALTER TABLE `personas`
  ADD PRIMARY KEY (`cod`);

COMMIT;

```

Aunque se use MySql los conceptos se pueden extrapolar a cualquier base de datos: Conexión → Operaciones → Desconexión, similar al protocolo con un fichero.

Instalación del driver Python

```
pip install mysql-connector-python
```

Conexión con la BBDD

```

import mysql.connector

cnn = mysql.connector.connect(user='root', password='',
                               host='127.0.0.1',
                               database='daw_prog')

print(cnn)
cnn.close()

```

Gestión de errores de conexión

```

import mysql.connector
from mysql.connector import errorcode

try:
    cnx = mysql.connector.connect(user='root', password='',
                                   host='127.0.0.1',
                                   database='daw_prog')
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Error de conexión")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("No existe la BBDD")
    else:
        print(err)
else:
    print(cnx)
    cnx.close()

```

Inserción de datos

```

from datetime import date, datetime, timedelta
import mysql.connector

```

```

cnx = mysql.connector.connect(user='root', password='',
                               host='127.0.0.1',
                               database='daw_prog')

cursor = cnx.cursor()
add_persona = ("INSERT INTO personas (nombre, apellido_1,
                                       apellido_2, fecha_nacimiento) " +
               "VALUES ('Jose', 'Pérez', 'González', '" +
               str(date(1977, 6, 14)) + "')")

print(add_persona)
cursor.execute(add_persona)
emp_no = cursor.lastrowid # devuelve el id en una autonumérico
print(emp_no)

cnx.commit()

cursor.close()
cnx.close()

```

Borrado de registros

```

import mysql.connector

cnx = mysql.connector.connect(user='root', password='',
                               host='127.0.0.1',
                               database='daw_prog')

cnx.autocommit = True
cursor = cnx.cursor()

del_persona = ("DELETE FROM personas WHERE cod = 1")
print(del_persona)

cursor.execute(del_persona)
emp_no = cursor.lastrowid
print(emp_no)

# cnx.commit() Ya no es necesario por el autocommit
cursor.close()
cnx.close()

```

Sentencias de selección

```

import mysql.connector

cnx = mysql.connector.connect(user='root', password='',
                               host='127.0.0.1',
                               database='daw_prog')

cursor = cnx.cursor()
query = "SELECT * FROM personas"

cursor.execute(query)

for cod, nombre, apellido_1, apellido_2, fecha_nacimiento in
cursor:
    print(f"({cod}) {nombre} {apellido_1} {apellido_2}
          nació el {fecha_nacimiento}")

```

```
cursor.close()
cnx.close()
```

Ejemplos de consultas sobre la base de datos

```
import mysql.connector

class GenerarEquipo:
    _cnn = None
    _data = None

    def __init__(self):
        try:
            self._cnn = mysql.connector.connect(user='root',
                                                password='', host='127.0.0.1', database='daw_prog')
            self._cnn.autocommit = True
            self._data = self._cnn.cursor()
            self._data.execute("SELECT * FROM personas")
        except mysql.connector.Error as e:
            print("-", e)

    def __del__(self):
        if self._data is not None:
            self._data.close()
        if self._cnn is not None:
            self._cnn.close()

    def next(self):
        while True:
            row = self._data.fetchone()
            if row is not None:
                yield row
            else:
                return

    misEquipos = GenerarEquipo()
    for eq in misEquipos.next():
        print(eq[0], " ", eq[1])

    del misEquipos
```

XML

XML es el acrónimo de Extensible Markup Language, es decir, es un lenguaje de marcado que define un conjunto de reglas para la codificación de documentos. El lenguaje de marcado es un conjunto de códigos que se pueden aplicar en el análisis de datos o la lectura de textos creados por computadoras o personas. El lenguaje XML proporciona una plataforma para definir elementos para crear un formato y generar un lenguaje personalizado.

Un archivo XML se divide en dos partes: DTD y DATA. La parte DTD consiste en metadatos administrativos, como declaración XML, instrucción de procesamiento opcional, declaración de tipo de documento y comentarios. La parte del DATA se compone de dos partes: estructural y de contenido (presente en los textos simples).

El diseño XML se centra en la simplicidad, la generalidad y la facilidad de uso y, por lo tanto, se utiliza para varios servicios web. Tanto es así que hay sistemas destinados a ayudar en la definición de lenguajes basados en XML, así como APIs que ayudan en el procesamiento de datos XML – que no deben confundirse con HTML.

Las interfaces de Python para procesar XML están agrupadas en el paquete **xml**.

Librerías Relacionadas con XML

SAX

El paquete **xml.sax** provee un número de módulos que implementan la API Simple para la interfaz XML (SAX) para Python. Una aplicación SAX típica usa tres tipos de objetos: lectores, gestores y fuentes de entrada. «Lector» en este contexto es otro término para analizador, por ejemplo, alguna pieza de código que lee los bytes o caracteres de la fuente de entrada, y produce una secuencia de eventos. Los eventos luego se distribuyen a los objetos gestores, por ejemplo, el lector invoca un método en el gestor. Una aplicación SAX debe por tanto obtener un objeto lector, crear o abrir una fuente de entrada, crear los gestores, y conectar esos objetos juntos. Como paso final de preparación, el lector es llamado para analizar la entrada. Durante el análisis, los métodos en los objetos gestores son llamados basados en eventos estructurales y sintácticos de los datos introducidos.

<https://docs.python.org/es/3/library/xml.sax.html>

DOM

El Modelo de Objetos del Documento, o «DOM» por sus siglas en inglés, es un lenguaje API del Consorcio World Wide Web (W3C) para acceder y modificar documentos XML. Una implementación del DOM presenta los documentos XML como un árbol, o permite al código cliente construir dichas estructuras desde cero para luego darles acceso a la estructura a través de un conjunto de objetos que implementaron interfaces conocidas.

El DOM es extremadamente útil para aplicaciones de acceso directo. SAX sólo te permite la vista de una parte del documento a la vez. Si estás mirando un elemento SAX, no tienes acceso a otro. Si estás viendo un nodo de texto, no tienes acceso al elemento contendor. Cuando desarrollas una aplicación SAX, necesitas registrar la posición de tu programa en el documento en algún lado de tu código. SAX no lo hace por ti. Además, desafortunadamente no podrás mirar hacia adelante (look ahead) en el documento XML.

Las aplicaciones DOM típicamente empiezan al diseccionar (parse) el XML en un DOM. Existe una clase objeto llamada **DOMImplementation** que da acceso a métodos de creación. Una vez que tengas un objeto del documento del DOM, puedes acceder a las partes de tu documento XML a través de sus propiedades y métodos. Estas propiedades están definidas en la especificación del DOM.

<https://docs.python.org/es/3.9/library/xml.dom.html#module-xml.dom>

Creación de un documento XML

```
<?xml version="1.0"?>
<genre catalogue="Pop">
    <song title="No Tears Left to Cry">
        <artist>Ariana Grande</artist>
        <year>2018</year>
        <album>Sweetener</album>
    </song>
    <song title="Delicate">
        <artist>Taylor Swift</artist>
        <year>2018</year>
        <album>Reputation</album>
    </song>
    <song title="Mrs. Potato Head">
        <artist>Melanie Martinez</artist>
        <year>2015</year>
        <album>Cry Baby</album>
    </song>
</genre>
```

SAX

```

import xml.sax
class SongHandler(xml.sax.ContentHandler):
    def __init__(self):
        self.CurrentData=''
        self.artist=''
        self.year=''
        self.album=''
    def startElement(self,tag,attributes):
        self.CurrentData=tag
        if tag=='song':
            print('Song:')
            title=attributes['title']
            print(f'Title: {title}')

    def endElement(self,tag):
        if self.CurrentData=='artist':
            print(f'Artist: {self.artist}')
        elif self.CurrentData=='year':
            print(f'Year: {self.year}')
        elif self.CurrentData=='album':
            print(f'Album: {self.album}')
        self.CurrentData=''

    def Characters(self,content):
        if self.CurrentData=='artist':
            self.artist=content
        elif self.CurrentData=='year':
            self.year=content
        elif self.CurrentData=='album':
            self.album=content

if __name__=='__main__':
    parser=xml.sax.make_parser() #creating an XMLReader
    parser.setFeature(xml.sax.handler.feature_namespaces,0)
    #turning off namespaces
    Handler=SongHandler()
    parser.setContentHandler(Handler)
    #overriding default ContextHandler
    parser.parse('songs.xml')

```

DOM

```

from xml.dom.minidom import parse
import xml.dom.minidom
import os
os.chdir('C:\\\\Users\\\\lifei\\\\Desktop')
DOMTree = xml.dom.minidom.parse("songs.xml") #Opening the XML document
genre=DOMTree.documentElement
if genre.hasAttribute('catalogue'):
    print(f'Root: {genre.getAttribute("catalogue")}')
songs=genre.getElementsByTagName('song') #Get all songs in the
for song in songs: #Print each song's details
    print('Song:')
    if song.hasAttribute('title'):
        print(f'Title: {song.getAttribute("title")}')

```

```
artist=song.getElementsByTagName('artist')[0]
print(f'Artist: {artist.firstChild.data}')
year=song.getElementsByTagName('year')[0]
print(f'Release Year: {year.firstChild.data}')
album=song.getElementsByTagName('album')[0]
print(f'Album: {album.firstChild.data}')
```

Ejercicios

1º Ejercicio

Escribir en un fichero llamado secuencias. las siguientes secuencias:

```
ACTG  
GATA
```

2º Ejercicio

Escribir los números del 1 al 100 en un fichero.

3º Ejercicio

Escribir en un fichero separado por tabuladores la información de un paciente que tenemos disponible en el siguiente diccionario:

```
paciente = {'nombre': 'Daniel', 'edad': 42, 'Diabetico': True}
```

4º Ejercicio

Dado el siguiente fichero con información sobre pacientes:

```
Nombre edad Diabetico  
Daniel 42 Si  
Jose 15 Si  
Manolo 50 No  
Alicia 12 No
```

Imprimir en un nuevo fichero los pacientes que tienen más de 20 años y no son diabéticos.

5º Ejercicio

Ejercicio - Máximo y mínimo

Implementa un programa que muestre por pantalla los valores máximos y mínimos del archivo ‘numeros.txt’.

Ejercicio - Notas de alumnos

El archivo ‘alumnos_notas.txt’ contiene una lista de 10 alumnos y las notas que han obtenido en cada asignatura. El número de asignaturas de cada alumno es variable. Implementa un programa que muestre por pantalla la nota media de cada alumno junto a su nombre y apellido, ordenado por nota media de mayor a menor.

Ejercicio - Ordenando archivos

Implementa un programa que pida al usuario un nombre de archivo A para lectura y otro nombre de archivo B para escritura. Leerá el contenido del archivo A (por ejemplo ‘usa_personas.txt’) y lo escribirá ordenado alfabéticamente en B (por ejemplo ‘usa_personas_sorted.txt’).

Ejercicio - Nombre y apellidos

Implementa un programa que genere aleatoriamente nombres de persona (combinando nombres y apellidos de ‘usa_nombres.txt’ y ‘usa_apellidos.txt’). Se le pedirá al usuario cuántos nombres de persona desea generar y a qué archivo **añadirlos** (por ejemplo ‘usa_personas_generado.txt’).

Ejercicio - Búsqueda en PI

Implementa un programa que pida al usuario un número de cualquier longitud, como por ejemplo “1234”, y le diga al usuario si dicho número aparece en el primer millón de decimales del número pi (están en el archivo ‘pi-million.txt’). No está permitido utilizar ninguna librería ni clase ni método que realice la búsqueda. Debes implementar el algoritmo de búsqueda tú.

Ejercicio - Estadísticas

Implementa un programa que lea un documento de texto y muestre por pantalla algunos datos estadísticos: número de líneas, número de palabras, número de caracteres y cuáles son las 10 palabras más comunes (y cuántas veces aparecen).

6º Ejercicio

Crear una aplicación que gestione las citas de una consulta veterinaria, que se puedan almacenar en una BBDD todos los datos. Haz el desarrollo completo, desde UML hasta la implementación final, utiliza las tablas necesarias para las citas, los datos de los clientes y mascotas. Se puede usar el código en el fichero code_0009.

7º Ejercicio

A partir de un fichero csv de las provincias y pueblos de España, crea un script que lo cargue en una BBDD. Se supone que las tablas provincias y pueblos ya están creadas. Utiliza la página del INE para descargar ficheros XLS con los datos y crear los ficheros csv.

<https://www.ine.es/>

**8º Ejercicios de ampliación varios***Uno*

Dado un fichero de texto, encontrar el número de veces que aparece cada palabra teniendo en cuenta que pueden estar separadas por un espacio en blanco o por un ; y los acentos hacen palabras diferentes: tu y tú son dos palabras diferentes.

Dos

1º Crear un script que genere un fichero con dígitos de forma aleatoria en una única fila. El número de ficheros se pide al usuario.

2º Crear un script que pida al usuario números y diga la posición en la que aparecen dichos números. Se finalizará al introducir -1. Si no se encuentra el número devolverá -1.

NO se puede usar ninguna característica de búsqueda de Python

Tres

Dadas tres listas de números desordenadas y con números repetidos, crear la lista unión de ambas sin elementos repetidos y ordenada de mayor a menor.

Cuatro

Dado un diccionario cuyos valores son listas de números, encontrar la clave que tiene mayor número de elementos no repetidos

Cinco

Dado un número de longitud indefinida, determinar si todas sus cifras son diferentes

Seis

Dada una lista ordenada, hacer una función que nos diga si un valor está o no en ella utilizando el algoritmo de búsqueda dicotómica.

U. T. 9 Interfaces gráficos.



Introducción al diseño GUI

Qué es la interfaz de usuario

La interfaz de usuario es el espacio donde se producen las interacciones entre seres humanos y máquinas. El objetivo de esta interacción es permitir el funcionamiento y control más efectivo de la máquina desde la interacción con el humano. Las interfaces básicas de usuario son aquellas que incluyen elementos como menús, ventanas, contenido gráfico, cursor, los beeps y algunos otros sonidos que la computadora hace, y en general, todos aquellos canales por los cuales se permite la comunicación entre el ser humano y la computadora. El objetivo del diseño de una interfaz es producir una interfaz que sea fácil de usar (explicarse por sí misma), eficiente y agradable para que al operar la máquina dé el resultado deseado.

La interfaz de usuario es el entorno de interacción persona-ordenador, la interfaz (o interfaz de usuario) es lo que permite que la interacción entre persona y ordenador ocurra. Es decir, la interfaz permite:

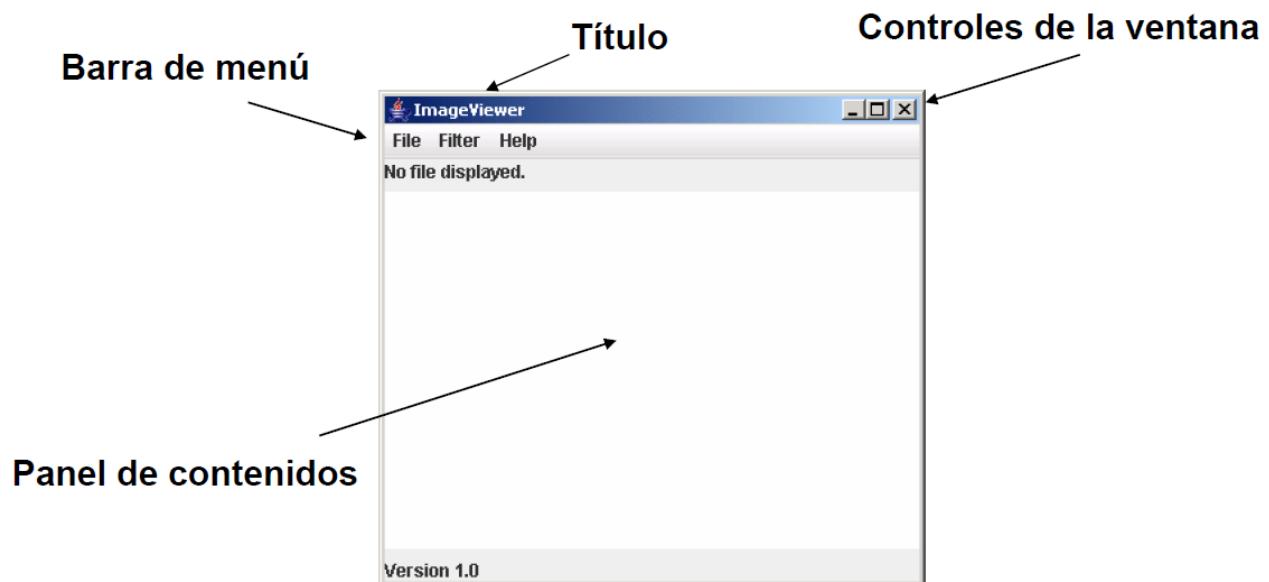
- ∞ Que la persona pueda controlar efectivamente las acciones de la máquina.
- ∞ Que la persona reciba respuestas de la máquina que le permitan saber si la interacción es correcta y cómo seguir actuando.

Por lo tanto, el diseñador de la interfaz se tiene que asegurar de que el proceso de interacción se puede efectuar de manera fácil e intuitiva y que la persona (a la que llamaremos de ahora en adelante usuario) puede acceder a la información o ejecutar las acciones que desea, de la manera más simple posible. Así, el diseño de interfaces implica conocimientos de disciplinas muy variadas, como por ejemplo, la psicología o el diseño visual.

En muchas ocasiones, la palabra interfaz se refiere en realidad a un concepto mucho más específico: la interfaz gráfica de usuario (GUI del inglés graphical user interface). La GUI es el entorno visual en el que se desarrolla la interacción entre la persona y el dispositivo, y puede ser el propio del sistema operativo o bien el particular de la aplicación que se está utilizando.

Aplicaciones con interfaz gráfica (GUI)

- ∞ Ventanas gráficas para entrada y salida de datos.
- ∞ Iconos.
- ∞ Dispositivos de entrada (p.ej. ratón, teclado).
- ∞ Interacción directa.



La GUI es una interfaz de usuario que permite a los usuarios comunicarse con el ordenador. Suele estar basada en la interacción a través del ratón y el teclado (aunque el control a través de gestos es cada vez más común): al mover el ratón, el puntero se desplaza en la pantalla. La señal del dispositivo se transmite al ordenador, que luego la traduce en un movimiento equivalente en la pantalla. Por ejemplo, si un usuario hace clic en un determinado ícono de programa en el menú, se ejecuta la instrucción correspondiente y se abre el programa.

La GUI es, por lo tanto, una especie de traductor en la comunicación entre el humano y la máquina. Sin la GUI, tendrías que utilizar la línea de comandos para llamar a programas y aplicaciones. Esto podría representarse así (el ejemplo muestra cómo abrir el explorador):

Qué es el diseño GUI

La interfaz persona-ordenador (IPO) se denomina en inglés human-computer interface (HCI). Tal como vamos a ver a lo largo de la unidad, otros conceptos estrechamente relacionados son arquitectura de la información, usabilidad y diseño de interacción.

El diseño de interfaz de usuario o ingeniería de la interfaz es el resultado de definir la forma, función, utilidad, ergonomía, imagen de marca y otros aspectos que afectan a la apariencia externa de las interfaces de usuario en sistemas de todo tipo (computadoras de uso general, sistemas de control, dispositivos de comunicación móviles, software de sistemas, software de aplicaciones, sitios web, etc.) El diseño de la interfaz de usuario es una disciplina asociada al diseño y se enfoca en maximizar la usabilidad y la experiencia de usuario. El objetivo final del diseño de la interfaz de usuario es hacer que la interacción entre el usuario y el sistema del que es interfaz sea tan simple y eficiente como sea posible, en términos de cumplimiento de los objetivos del usuario. Sigue por ello una filosofía de diseño centrado en el usuario.

Un buen diseño de la interfaz de usuario facilita la compleción de tareas a realizar sin que el usuario vea atraída su atención hacia la forma. El diseño gráfico y la tipografía se combinan para ofrecer usabilidad, influyendo en cómo el usuario realiza ciertas interacciones y mejorando la apariencia estética del diseño; la estética del diseño puede mejorar o dificultar la capacidad de los usuarios para utilizar las funciones de la interfaz. El proceso de diseño debe balancear la funcionalidad técnica y los elementos visuales (es decir, el modelo mental) para crear un sistema que no solo sea operativo, sino también usable y adaptable a la evolución de las necesidades del usuario.

Normalmente el diseño de interfaces de usuario es una actividad multidisciplinar que involucra a varias ramas tales como el diseño gráfico, el diseño industrial, el diseño web, el diseño de software y la ergonomía; y puede aparecer como actividad en un amplio rango de proyectos, desde el desarrollo de sistemas informáticos hasta el desarrollo de aviones comerciales. En este sentido las disciplinas del diseño industrial y diseño gráfico se encargan de que la actividad a desarrollar se comunique y aprenda lo más rápidamente, a través de recursos como los gráficos, los pictogramas, los estereotipos y la simbología, todo sin afectar un funcionamiento técnico eficiente.

Elementos de un Diseño GUI.

- ∞ Contenedores
 - Contienen otros componentes (u otros contenedores)
Estos componentes se tienen que añadir al contenedor y para ciertas operaciones se pueden tratar como un todo. Mediante un gestor de diseño controlan la disposición (layout) de estos componentes en la pantalla.
- ∞ Lienzo
 - Superficie simple de dibujo
- ∞ Componentes de interfaz de usuario
 - Botones, listas, menús, casillas de verificación, campos de texto, etc.
- ∞ Componentes de construcción de ventanas
 - Ventanas, marcos, barras de menús, cuadros de diálogo

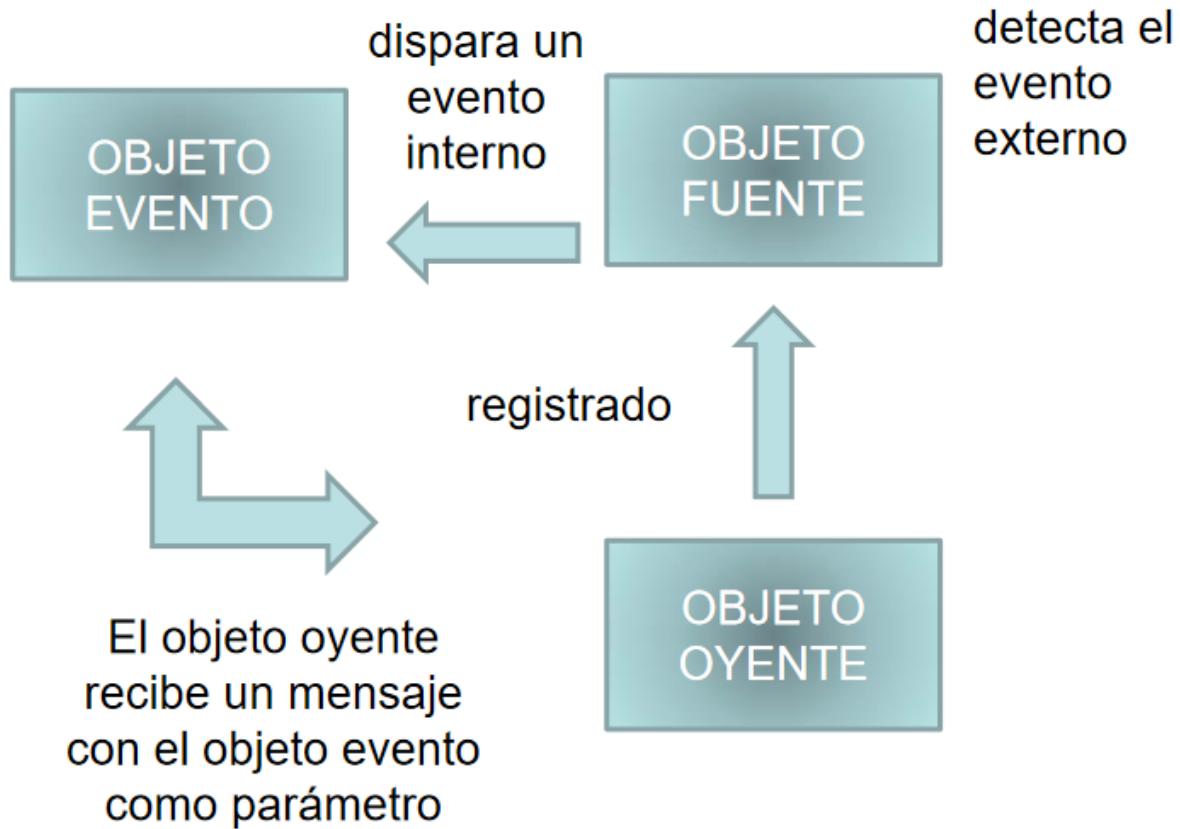
La construcción de una GUI utiliza un modelo de programación basado en eventos.

Diseño dirigido a eventos

En este modelo el orden en el cual se ejecutan las instrucciones de un programa va a quedar determinado por eventos. Un evento es una señal de que algo ha ocurrido. En esta materia consideraremos únicamente eventos generados por acciones del usuario al interactuar con la GUI.

Algunas componentes de una GUI van a ser reactivas, es decir tienen la capacidad de reaccionar ante las acciones del usuario. Un componente reactivo está asociada a un objeto fuente del evento creado por el programador. La reacción del sistema en respuesta a la acción del usuario va a quedar determinada por la clase a la que pertenece un objeto oyente. El objeto oyente está ligado al objeto fuente de evento a través de una instrucción de registro.

Un objeto fuente de evento tienen la capacidad de percibir un evento externo y disparar un evento interno, esto es, crear un objeto evento de software. Este objeto evento de software es el argumento de un mensaje enviado al objeto oyente. El método que se ejecuta en respuesta a este mensaje forma parte de una interface y es implementado por el programador en la clase del oyente



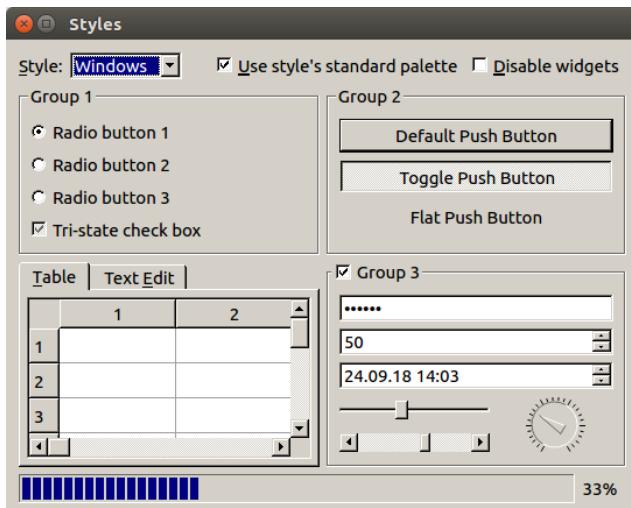
Creación de un GUI

La construcción de una GUI va a requerir:

- ∞ Crear objetos de las clases provistas o de las clases derivadas o implementadas.
- ∞ Elegir un organizador de espacio (layout) y especificar la apariencia de los componentes.
- ∞ Insertar los componentes en los contenedores.
- ∞ Crear los manejadores de eventos y asignarlos a los objetos y eventos correspondientes.
- ∞ Crear las clases de ayuda y de conexión con los servicios.

Componentes

Una interfaz gráfica de usuario (GUI) presenta un mecanismo amigable al usuario para interactuar con una aplicación. Las GUIs se crean a partir de componentes de la GUI, a estos se les conoce también como controles o widgets (accesorios de ventana) en otros lenguajes. Un componente de la GUI es un objeto con el cual interactúa el usuario mediante el ratón, el teclado u otra forma de entrada como el reconocimiento de voz que genera un evento que debemos capturar a través de una callback para realizar una tarea. Ejemplos de los posibles controles de una interfaz gráfica.



Librerías

El número de librería gráficas existentes son muchas: GTK, QT, Nativas, WxWidget, etc. Podemos usar cualquiera de ellas y dependiendo de la finalidad de la aplicación son más apropiadas unas u otras. En nuestro caso vamos a ver QT pero también queremos hablar de Tkinter que es muy utilizada para aplicaciones simples.

Tanto Tkinter como PyQt son dos librerías útiles para diseñar interfaces gráficas de usuario, pero al mismo tiempo difieren en términos de adaptabilidad y funcionalidad. La mayoría de las veces, Tkinter se trata de escribir GUI directamente, programar su configuración o funcionalidad en el mismo script. Por otro lado, en PyQt, se separa la interfaz gráfica de usuario en un script y se utiliza el conocimiento de Python de otro script. En lugar de crear su propio código para la interfaz de usuario, puede simplemente adoptar las funciones Qt Designer para desarrollar su aplicación.

Dentro de las librerías dedicadas a QT, tenemos dos aproximaciones PyQt desarrollo independiente y PySide desarrollado por la propia creadora de la librería QT, ambas tienen conceptos similares y se puede utilizar cualquiera, pero no combinar.

Qt

Ventajas de usar PyQt - PySide

- ∞ Flexibilidad de codificación - La programación GUI con Qt está diseñada alrededor del concepto de señales y slots para establecer comunicación entre objetos. Esto permite flexibilidad cuando se trata de eventos GUI y resulta en una base de código más fluida.
- ∞ Más que un marco de trabajo - Qt utiliza una amplia gama de APIs de plataformas nativas con el propósito de *redes, creación de bases de datos y mucho más*. Ofrece acceso primario a ellos a través de una API única.
- ∞ Varios componentes de la interfaz de usuario - Qt ofrece varios widgets, como botones o menús, todos ellos diseñados con una apariencia básica en todas las plataformas soportadas.
- ∞ Varios recursos de aprendizaje: dado que Qt es uno de los marcos de trabajo de interfaz de usuario más utilizados para Python, puede acceder fácilmente a una amplia gama de documentación.

- ∞ Fácil de dominar - Viene con una funcionalidad API sencilla y fácil de usar, junto con clases específicas vinculadas a Qt C++. Esto permite al usuario utilizar conocimientos previos de Qt o C++, haciendo que PyQt sea fácil de entender.

Desventajas de usar Qt

- ∞ Requiere mucho tiempo para entender todos los detalles de Qt, lo que significa que es una curva de aprendizaje bastante elevada.

Tkinter

Ventajas de usar Tkinter

- ∞ Disponible sin cargo para uso comercial.
- ∞ Se presenta en la biblioteca Python subyacente.
- ∞ La creación de ejecutables para las aplicaciones de Tkinter es más accesible ya que Tkinter está incluido en Python, y, como consecuencia, no viene con ninguna otra dependencia.
- ∞ Simple de entender y dominar, ya que Tkinter es una librería limitada con una API simple, siendo la opción principal para crear interfaces gráficas de usuario rápidas para scripts Python.

Desventajas de usar Tkinter

- ∞ Tkinter no incluye widgets avanzados.
- ∞ No tiene una herramienta similar a Qt Designer para Tkinter.
- ∞ No tiene un aspecto nativo

¿Cuál elegir?

De todos modos, en la mayoría de los casos, la mejor solución es utilizar PyQt - PySide, teniendo en cuenta las ventajas y desventajas de ambos.

La programación GUI con Qt se crea alrededor de señales y ranuras para la comunicación entre objetos. De este modo, permite flexibilidad, mientras que permite al programador acceder a una amplia gama de herramientas.

Tkinter puede ser útil para aquellos que quieren diseñar una GUI fundamental y rápida para scripts Python, pero *para un resultado de programación más avanzado*, casi todos los programadores optan por las funcionalidades que vienen con PyQt - PySide.

Programación GUI Básica

Hola mundo

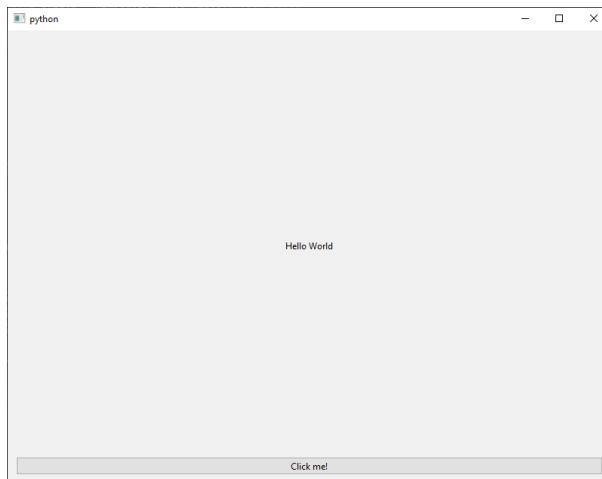
PySide

```
pip install pyside6
```

```
import PySide6.QtCore

# Prints PySide6 version
print(PySide6.__version__)

# Prints the Qt version used to compile PySide6
print(PySide6.QtCore.__version__)
```



```
import sys
import random
from PySide6 import QtCore, QtWidgets, QtGui

class MyWidget(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.hello = ["Hallo Welt", "Hei maailma", "Hola Mundo",
                     "Привет мир"]
        self.button = QtWidgets.QPushButton("Click me!")
        self.text = QtWidgets.QLabel("Hello World",
                                    alignment=QtCore.Qt.AlignCenter)
        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self.text)
        self.layout.addWidget(self.button)
        self.button.clicked.connect(self.magic)

    @QtCore.Slot()
    def magic(self):
        self.text.setText(random.choice(self.hello))

if __name__ == "__main__":
    app = QtWidgets.QApplication([])

    widget = MyWidget()
    widget.resize(800, 600)
    widget.show()

    sys.exit(app.exec_())
```

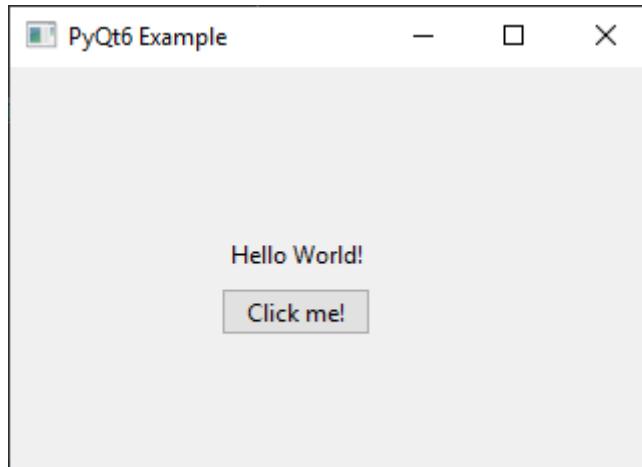
Primero se importan las librerías necesarias, y se pasa a definir una clase MyWidget que definirá la aplicación a ejecutar. En el constructor realizamos toda la tarea de construcción de la ventana y control de eventos de la siguiente manera.

1. Llamamos al constructor padre.
2. Creamos un atributo con las frases a contener (hello)
3. Creamos un botón con el texto “Clic me”.
4. Creamos una etiqueta con el texto “Hello Word” centrada
5. Creamos una disposición (layout) para contener a los controles.
6. Añadimos al layout los controles antes creados (addWidget)
7. Conectamos el evento clic del botón con la callback magic definida más adelante.

Para finalizar la clase definimos la callback, en este caso simplemente cambia el texto del propio control a uno de la propiedad hello de forma aleatoria.

PyQt

```
pip install pyqt6
```



```
import sys
from PyQt6.QtWidgets import QApplication, QWidget, QLabel,
QPushButton

def window():
    def button_text_clicked():
        textLabel.setText("click")

    app = QApplication(sys.argv)
    widget = QWidget()

    textLabel = QLabel(widget)
    textLabel.setText("Hello World!")
    textLabel.move(110, 85)

    button_text = QPushButton(widget)
    button_text.setText("Click me!")
    button_text.move(105, 110)
    button_text.clicked.connect(button_text_clicked)

    widget.setGeometry(50, 50, 320, 200)
    widget.setWindowTitle("PyQt6 Example")
    widget.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    window()
```

Primeros pasos con el GUI

PySide es una biblioteca de Python para crear interfaces gráficas de usuario multiplataforma. Es un enlace de Python al marco Qt. La biblioteca Qt es una de las bibliotecas GUI más poderosas. PySide se implementa como un conjunto de módulos de Python. Estos módulos proporcionan herramientas para trabajar con GUI, multimedia, documentos XML, redes o bases de datos. Trabajaremos con dos de estos módulos. Los módulos QtGui y QtCore.

El módulo QtCore contiene la funcionalidad principal no GUI. Este módulo se utiliza para trabajar con tiempo, archivos y directorios, varios tipos de datos, flujos, URL, tipos de mime, subprocessos o procesos. El módulo QtGui contiene los componentes gráficos y las clases relacionadas. Estos incluyen, por ejemplo, botones, ventanas, barras de estado, barras de herramientas, controles deslizantes, mapas de bits, colores, fuentes, etc. PySide es un conjunto de herramientas de alto nivel.

```
import sys
from PySide6 import QtWidgets, QtGui

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__() #Inicializar la clase base

        # Establecer la apariencia de la ventana
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Icon')
        self.setWindowIcon(QtGui.QIcon('web.png'))

def main():
    app = QtWidgets.QApplication(sys.argv)

    ex = Example()
    ex.show()

    sys.exit(app.exec_())

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

Podemos añadir una sugerencia de ayuda a cualquier caja de texto.

```
import sys
from PySide6 import QtWidgets, QtGui

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        btn = QtWidgets.QPushButton('Button', self)
        btn.setToolTip('This is a <b>QPushButton</b> widget')
        btn.resize(btn.sizeHint())
        btn.move(50, 50)
        btn.setToolTip('This is a <b>QWidget</b> widget')

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Tooltips')

def main():
    app = QtWidgets.QApplication(sys.argv)

    ex = Example()
    ex.show()
```

```
    sys.exit(app.exec_())
```

Cierre de la ventana.

```
import sys
from PySide6 import QtWidgets, QtCore

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        qbtn = QtWidgets.QPushButton('Quit', self)
        qbtn.clicked.connect(
            QtCore.QCoreApplication.instance().quit)
        qbtn.resize(qbtn.sizeHint())
        qbtn.move(50, 50)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Quit button')

def main():
    app = QtWidgets.QApplication(sys.argv)

    ex = Example()
    ex.show()

    sys.exit(app.exec_())

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

Con estos tres ejemplos queremos introducir los conceptos básicos de la programación GUI. En primer lugar, tendremos que crear nuestra ventana extendiendo de `QtWidgets.QWidget`. Dentro del constructor determinaremos el aspecto de la misma, creando todos los controles que deba tener, así como todas las uniones entre los eventos y las callback correspondientes. A continuación, en la clase, crearemos las callback correspondientes añadiendo el decorador `@QtCore.Slot()` o usando una acción por defecto del sistema `QtCore.QCoreApplication.instance().quit`.

Cada control tiene un conjunto de métodos y propiedades que podremos utilizar, también tendrá un conjunto de eventos que va a generar que podemos controlar, pero no es obligatorio implementarlos todos, solo aquellos que necesitemos para nuestra programación.

Controlando eventos de la ventana principal

```
import sys
from PySide6 import QtWidgets

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Message box')
```

```

def closeEvent(self, event):
    reply = QtWidgets.QMessageBox.question(self, 'Message',
                                           "Are you sure to quit?",
                                           QtWidgets.QMessageBox.Yes | 
                                           QtWidgets.QMessageBox.No,
                                           QtWidgets.QMessageBox.No)

    if reply == QtWidgets.QMessageBox.Yes:
        event.accept()
    else:
        event.ignore()

def main():
    app = QtWidgets.QApplication(sys.argv)

    ex = Example()
    ex.show()

    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

Al igual que los controles, la ventana principal que definimos tiene eventos que pueden ser capturados. Estos en vez de definir una callback y utilizar el decorador, debemos sobrescribir los correspondientes métodos. En el caso anterior se capture el evento de cierre, se lanza una pregunta al usuario mediante el QMessageBox y en función de la respuesta, se cierra o no la ventana al aceptar el evento (event.accept() o rechazarlo).

Eventos QT

El número de eventos presentes en un entorno GUI es muy numeroso, los siguientes son una pequeña muestra de los que se pueden capturar.

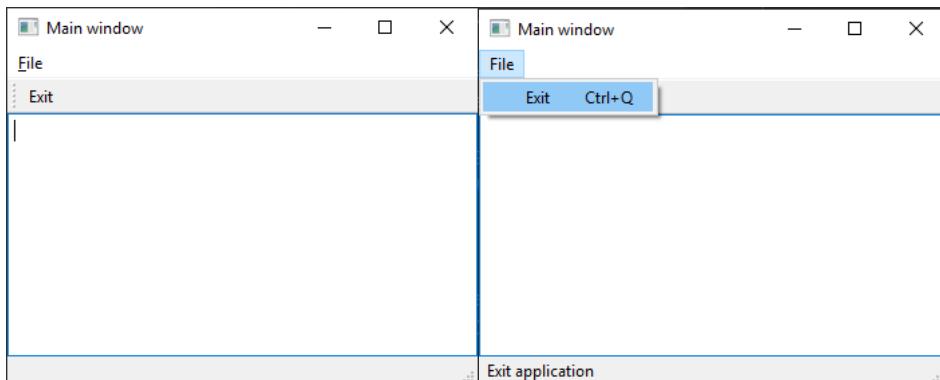
Evento	Descripción
QCloseEvent	Cierre de una ventana
QContextMenuEvent	Apertura del menú contextual de un control
QEnterEvent	Al entrar en un control
QFocusEvent	Cuando el control coge el foco de texto
QHideEvent	Cuando se oculta un control
QHoverEvent	Cuando se pasa el puntero del ratón por encima de un control
QInputEvent	Cuando el Usuario hace una entrada de texto
QKeyEvent	Cuando el usuario pulsa una tecla
QMoveEvent	Cuando se mueve el puntero del ratón
QShowEvent	Cuando se muestra un control
QClickEvent	Cuando se hace clic en un elemento

Elementos de la ventana principal

- ∞ La clase QtGui.QMainWindow proporciona una ventana principal de la aplicación. Esto permite crear el esqueleto de la aplicación clásica con una barra de estado, barras de herramientas y una barra de menú.
- ∞ La barra de estado es un widget que se utiliza para mostrar información de estado. Para obtener la barra de estado, llamamos al método statusBar() de la clase QtGui.QMainWindow. La primera llamada del método crea una barra de estado. Las llamadas posteriores devuelven el objeto de la barra de estado. ShowMessage () muestra un mensaje en la barra de estado.
- ∞ Una barra de menú es una parte común de una aplicación GUI. Es un grupo de comandos ubicados en varios menús. Mientras que en las aplicaciones de consola tenemos que recordar varios comandos

y sus opciones, aquí tenemos la mayoría de los comandos agrupados en partes lógicas. Estos son estándares aceptados que reducen aún más la cantidad de tiempo dedicado a aprender una nueva aplicación.

- ∞ Los menús agrupan todos los comandos que podemos usar en una aplicación. Las barras de herramientas proporcionan un acceso rápido a los comandos más utilizados.



```
import sys
from PySide6 import QtGui, QtWidgets

class Example(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        textEdit = QtWidgets.QTextEdit()
        self.setCentralWidget(textEdit)

        exitAction = QtGui.QAction('Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.setStatusTip('Exit application')
        exitAction.triggered.connect(self.close)

        self.statusBar()

        menubar = self.menuBar()
        fileMenu = menubar.addMenu('&File')
        fileMenu.addAction(exitAction)

        toolbar = self.addToolBar('Exit')
        toolbar.addAction(exitAction)

        self.setGeometry(300, 300, 350, 250)
        self.setWindowTitle('Main window')

    def main():
        app = QtWidgets.QApplication(sys.argv)
        ex = Example()
        ex.show()

        sys.exit(app.exec_())

if __name__ == '__main__':
```

```
main()
```

El código es muy simple, pero sirve para exemplificar todos los elementos explicados al comienzo del punto.

1. Se crea una Ventana de aplicación al heredar de QMainWindow, en su constructor se crean los elementos de la ventana llamando a initUI.
 - a. Una caja de texto que ocupa toda el área.
 - b. Se crea una acción del sistema (Exit) a la que se le asigna un acelerador de teclado. También cambiará el texto de la barra de estado para indicar el texto: Exit application.
 - c. Se habilita en la ventana la barra de estado.
 - d. Se crea un menú con un único elemento que se enlaza a la acción del sistema creada anteriormente, salir.
 - e. Se crea una barra de herramientas con un único botón para salir, enlazándolo también a la misma acción.
 - f. Se establece el título y el tamaño de la ventana.
2. Se crear la aplicación y se muestra.

Eventos y mensajes

Los eventos son una parte importante en cualquier programa de GUI. Los eventos los generan los usuarios o el sistema.

Todas las aplicaciones de GUI están controladas por eventos. Una aplicación reacciona a diferentes tipos de eventos que se generan durante su vida. Los eventos son generados principalmente por el usuario de una aplicación, pero también pueden generarse por otros medios. p.ej. Conexión a Internet, administrador de ventanas, temporizador. En el modelo de eventos, hay tres elementos:

- ∞ fuente del evento
- ∞ objeto de evento
- ∞ objetivo del evento

La fuente del evento es el objeto cuyo estado cambia, genera eventos. El objeto de evento (Evento) encapsula los cambios de estado en el origen del evento. El objetivo del evento es el componente que desea ser notificado. El objeto de origen del evento delega la tarea de manejar un evento al objetivo del evento y éste usa la callback para dicha gestión.

Cuando llamamos al método exec_() de la aplicación, la aplicación entra en el ciclo principal. El bucle principal busca eventos y los envía a los objetos. Las señales y las callback se utilizan para la comunicación entre objetos. Se emite una señal cuando ocurre un evento en particular, se llama a una callback cuando se emite una señal conectada a ella.

```
import sys
from PySide6 import QtCore, QtWidgets

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        lcd = QtWidgets.QLCDNumber(self)

        sld = QtWidgets.QSlider(QtCore.Qt.Horizontal, self)
        sld.valueChanged.connect(lcd.display)

        vbox = QtWidgets.QVBoxLayout()
        vbox.addWidget(lcd)
```

```

vbox.addWidget(sld)
self.setLayout(vbox)

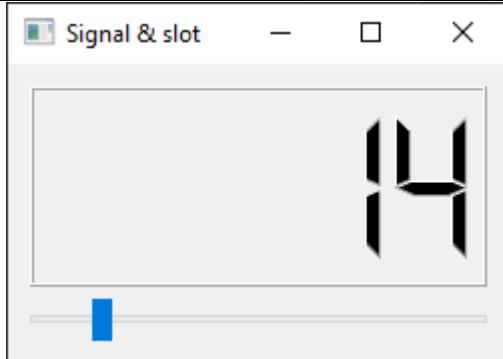
self.setGeometry(300, 300, 250, 150)
self.setWindowTitle('Signal & slot')

def main():
    app = QtWidgets.QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

1. Creamos un control lcd numérico.
2. Creamos un control slider o barra de desplazamiento y conectamos el evento de cambio de la barra (changed) con el de visualización del lcd (display).
3. Añadimos un layout, se verá en el siguiente punto y los widget a él.



Las callback puede recoger parámetros, en concreto el objeto evento que se está transmitiendo para poder utilizarlo en su beneficio.

```

import sys
from PySide6 import QtWidgets

class Example(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        btn1 = QtWidgets.QPushButton("Button 1", self)
        btn1.move(30, 50)

        btn2 = QtWidgets.QPushButton("Button 2", self)
        btn2.move(150, 50)

        btn1.clicked.connect(self.buttonClicked)
        btn2.clicked.connect(self.buttonClicked)

        self.statusBar()

        self.setGeometry(300, 300, 290, 150)
        self.setWindowTitle('Event sender')

    def buttonClicked(self):
        sender = self.sender()

```

```

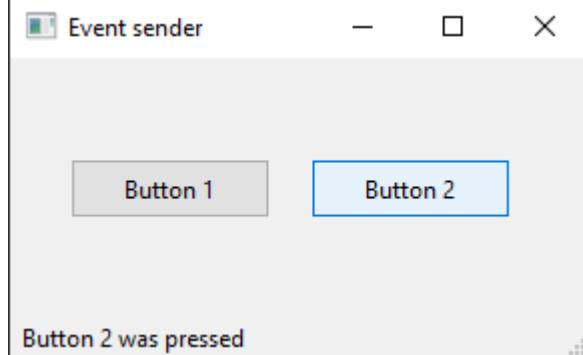
        self.statusBar().showMessage(sender.text() + ' was
pressed')

def main():
    app = QtWidgets.QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

1. Creamos dos botones y los posicionamos.
2. Conectamos el evento click con la callback buttonClicked en la que se utiliza el método sender() para acceder al control origen, en concreto se cambia el texto de la barra de estado.



Diálogos

Las ventanas de diálogo o los cuadros de diálogo son comunes en las aplicaciones GUI modernas. En una aplicación informática, un cuadro de diálogo es una ventana que se utiliza para "hablar" con la aplicación. Un cuadro de diálogo se utiliza para ingresar datos, modificar datos, cambiar la configuración de la aplicación, etc. Los cuadros de diálogo son medios importantes de comunicación entre un usuario y un programa.

- ∞ El QtGui.QInputDialog proporciona un diálogo de conveniencia simple para obtener un valor único de un usuario. El valor de entrada puede ser una cadena, un número o un elemento de una lista.
- ∞ The QtGui.QColorDialog es un diálogo para seleccionar un color.
- ∞ The QtGui.QFontDialog es un diálogo para seleccionar una fuente.
- ∞ The QtGui.QFileDialog es un diálogo para seleccionar un directorio o fichero tanto para escritura como lectura de un fichero.

```

import sys
from PySide6 import QtWidgets

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.btn = QtWidgets.QPushButton('Dialog', self)
        self.btn.move(20, 20)
        self.btn.clicked.connect(self.showDialog)

        self.le = QtWidgets.QLineEdit(self)
        self.le.move(130, 22)

        self.setGeometry(300, 300, 290, 150)
        self.setWindowTitle('Input dialog')

```

```

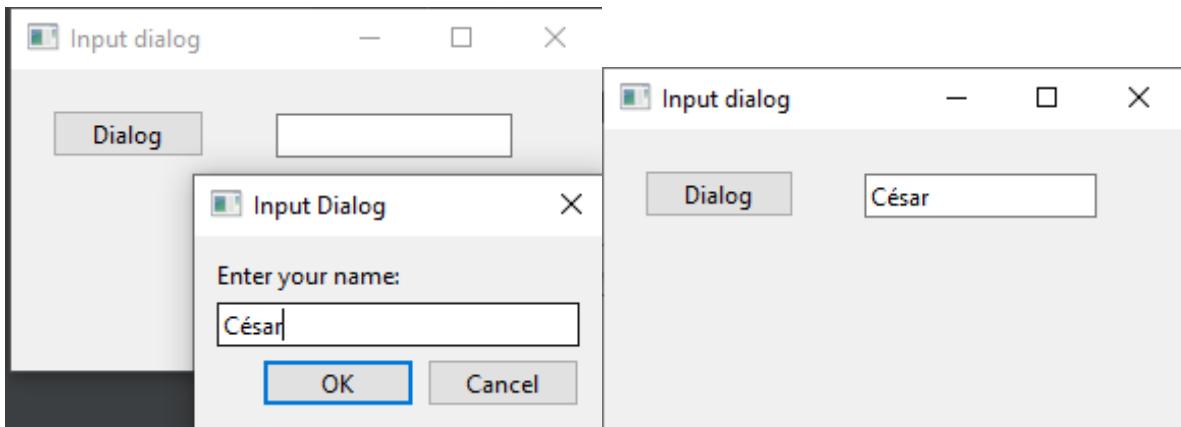
def showDialog(self):
    text, ok = QtWidgets.QInputDialog.getText(self,
        'Input Dialog',
        'Enter your name:')

    if ok:
        self.le.setText(str(text))

def main():
    app = QtWidgets.QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

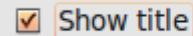


1. Creamos un botón para mostrar el diálogo, en el clic del botón se conecta con la callback .showDialog.
2. Creamos un control de texto para mostrar el texto introducido por el usuario en el diálogo.
3. El manejador del clic, abre el diálogo (QtWidgets.QInputDialog.getText) y recoge tanto el texto (text) como el botón que se ha pulsado (ok).
4. Si se ha pulsado el botón ok, se muestra el texto en la caja de texto.

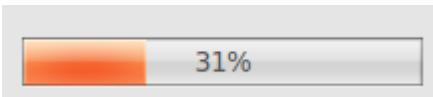
Controles

Los widgets son componentes básicos de una aplicación. El kit de herramientas de programación PySide tiene una amplia gama de controles: botones, casillas de verificación, controles deslizantes, cuadros de lista, etc. Todo lo que un programador necesita para su trabajo.

- ∞ **QtGui.QCheckBox.**
 - cb = QtGui.QCheckBox('Show title', self)
 - cb.toggle()
 - cb.stateChanged.connect(self.changeTitle)
- ∞ **ToggleButton.** PySide no tiene ningún widget para ToggleButton. Para crear un ToggleButton, usamos un QtGui.QPushButton en un modo especial. ToggleButton es un botón que tiene dos estados. Presionado y no presionado. Puede alternar entre estos dos estados haciendo clic en él.
 - greenb = QtGui.QPushButton('Green', self)
 - greenb.setCheckable(True)
 - greenb.clicked[bool].connect(self.setColor)
- ∞ **QtGui.QSlider.** QtGui.QSlider es un widget que tiene un identificador simple. Esta manija se puede tirar hacia adelante y hacia atrás. De esta forma estamos eligiendo un valor para una tarea específica. A veces, usar un control deslizante es más natural que simplemente proporcionar un número o usar un cuadro de giro. QtGui.QLabel muestra texto o imagen.



- `sld = QtGui.QSlider(QtCore.Qt.Horizontal, self)`
- `self.label = QtGui.QLabel(self)`
- `self.label.setPixmap(QtGui.QPixmap('mute.png'))`
- `sld.valueChanged[int].connect(self.changeValue)`
- ∞ **QtGui.QProgressBar.** Una barra de progreso es un widget que se utiliza cuando procesamos tareas largas. Está animado para que el usuario sepa que nuestra tarea está progresando. El widget QtGui.QProgressBar proporciona una barra de progreso horizontal o vertical. El programador puede establecer los valores mínimo y máximo para la barra de progreso. Los valores predeterminados son 0, 99.
 - `self.pbar = QtGui.QProgressBar(self)`
 - `self.timer = QtCore.QBasicTimer()`
 - `self.timer.start(100, self)`
- ∞ **QtGui.QCalendarWidget.** El QtGui.QCalendarWidget proporciona un widget de calendario mensual. Permite al usuario seleccionar una fecha de forma sencilla e intuitiva.
 - `self.cal = QtGui.QCalendarWidget(self)`
 - `cal.clicked[QtCore.QDate].connect(self.showDate)`
 - `def showDate(self, date): self.lbl.setText(date.toString())`



Controles avanzados

- ∞ **QtGui.QPixmap.** QtGui.QPixmap es uno de los widgets que se utilizan para trabajar con imágenes. Está optimizado para mostrar imágenes en pantalla.
 - `pixmap = QtGui.QPixmap("redrock.png")`
 - `lbl = QtGui.QLabel(self)`
 - `lbl.setPixmap(pixmap)`
- ∞ **QtGui.QLineEdit.** QtGui.QLineEdit es un widget que permite ingresar y editar una sola línea de texto plano. Hay funciones de deshacer / rehacer, cortar / pegar y arrastrar y soltar disponibles para el widget QtGui.QLineEdit.
 - `qle = QtGui.QLineEdit(self)`
 - `qle.textChanged[str].connect(self.onChanged)`
 - `def onChanged(self, text):`
 `self.lbl.setText(text)`
 `self.lbl.adjustSize()`
- ∞ **QtGui.QSplitter.** QtGui.QSplitter permite al usuario controlar el tamaño de los widgets secundarios arrastrando el límite entre los mismos.
 - `topleft = QtGui.QFrame(self)`
 - `topleft.setFrameShape(QtGui.QFrame.StyledPanel)`
 - `splitter1 = QtGui.QSplitter(QtCore.Qt.Horizontal)`
 - `splitter1.addWidget(topleft)`
 - `splitter1.addWidget(topright)`
 - `splitter2 = QtGui.QSplitter(QtCore.Qt.Vertical)`
 - `splitter2.addWidget(splitter1)`
 - `QtGui.QApplication.setStyle(QtGui.QStyleFactory.create('Cleanlooks'))`
- ∞ **QtGui.QComboBox.** El QtGui.QComboBox es un widget que permite al usuario elegir de una lista de opciones.
 - `combo = QtGui.QComboBox(self)`
 - `combo.addItem("Ubuntu")`
 - `combo.addItem("Mandriva")`
 - `combo.addItem("Fedora")`

- `combo.addItem("Red Hat")`
- `combo.addItem("Gentoo")`
- `combo.activated[str].connect(self.onActivated)`
- `def onActivated(self, text):`
 `self.lbl.setText(text)`
 `self.lbl.adjustSize()`

Layouts

Lo importante en la programación de GUI es la gestión del diseño. La gestión del diseño es la forma en que colocamos los widgets en la ventana. La gestión se puede realizar de dos formas. Podemos utilizar clases de diseño o posicionamiento absoluto.

Absolute

Con este mecanismo, el programador especifica la posición y el tamaño de cada widget en píxeles. Cuando utiliza el posicionamiento absoluto, debe comprender varias cosas.

- ∞ El tamaño y la posición de un widget no cambian si cambiamos el tamaño de una ventana
- ∞ Las aplicaciones pueden verse diferentes en varias plataformas
- ∞ Cambiar las fuentes en nuestra aplicación podría estropear el diseño
- ∞ Si decidimos cambiar nuestro diseño, debemos rehacer completamente nuestro diseño, lo cual es tedioso y requiere mucho tiempo.

Simplemente llamamos al método `move()` para posicionar nuestros widgets. Los posicionamos proporcionando las coordenadas x e y. El comienzo del sistema de coordenadas está en la esquina superior izquierda. Los valores de x crecen de izquierda a derecha. Los valores de y crecen de arriba a abajo.

Box Layout

La gestión del diseño con clases de diseño es mucho más flexible y práctica. Es la forma preferida de colocar widgets en una ventana. Las clases de diseño básicas son `QtGui.QHBoxLayout` y `QtGui.QVBoxLayout`. Alinean los widgets horizontal y verticalmente.

Imagina que quisiéramos colocar dos botones en la esquina inferior derecha. Para crear tal diseño, usaremos un cuadro horizontal y uno vertical. Para crear el espacio necesario, agregaremos un factor de estiramiento.

```
import sys
from PySide6 import QtWidgets

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        okButton = QtWidgets.QPushButton("OK")
        cancelButton = QtWidgets.QPushButton("Cancel")

        hbox = QtWidgets.QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(okButton)
        hbox.addWidget(cancelButton)

        vbox = QtWidgets.QVBoxLayout()
        vbox.addStretch(1)
        vbox.addLayout(hbox)

        self.setLayout(vbox)
```

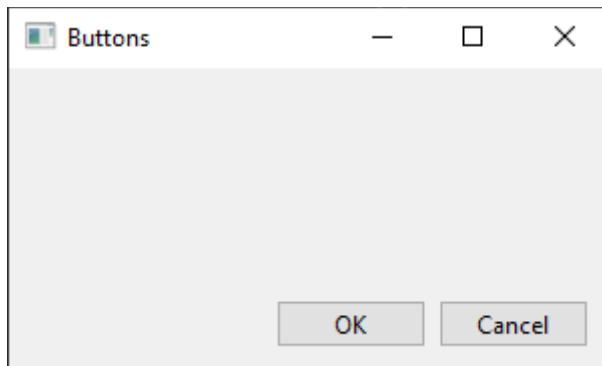
```

        self.setGeometry(300, 300, 300, 150)
        self.setWindowTitle('Buttons')

def main():
    app = QtWidgets.QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```



Grid layout

La clase de diseño más universal en PySide es el diseño de cuadrícula. Este diseño divide el espacio en filas y columnas. Para crear un diseño de cuadrícula, usamos la clase QtGui.QGridLayout.

```

import sys
from PySide6 import QtWidgets

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        names = ['Cls', 'Bck', '', 'Close', '7', '8', '9', '/',
                 '4', '5', '6', '*', '1', '2', '3', '-',
                 '0', '.', '=', '+']

        grid = QtWidgets.QGridLayout()

        j = 0
        pos = [(0, 0), (0, 1), (0, 2), (0, 3),
                (1, 0), (1, 1), (1, 2), (1, 3),
                (2, 0), (2, 1), (2, 2), (2, 3),
                (3, 0), (3, 1), (3, 2), (3, 3),
                (4, 0), (4, 1), (4, 2), (4, 3)]

        for i in names:
            button = QtWidgets.QPushButton(i)
            if j == 2:
                grid.addWidget(QtWidgets.QLabel(''), 0, 2)
            else:
                grid.addWidget(button, pos[j // 4][j % 4])
            j += 1

```

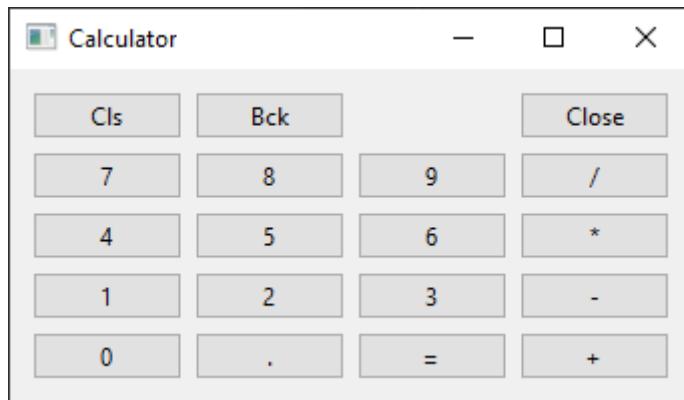
```
        else:
            grid.addWidget(button, pos[j][0], pos[j][1])
            j = j + 1

    self.setLayout(grid)

    self.move(300, 150)
    self.setWindowTitle('Calculator')

def main():
    app = QtWidgets.QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

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



Ejemplo

```
import sys
from PySide6 import QtWidgets

class Example(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        title = QtWidgets.QLabel('Title')
        author = QtWidgets.QLabel('Author')
        review = QtWidgets.QLabel('Review')

        titleEdit = QtWidgets.QLineEdit()
        authorEdit = QtWidgets.QLineEdit()
        reviewEdit = QtWidgets.QTextEdit()

        grid = QtWidgets.QGridLayout()
        grid.setSpacing(10)

        grid.addWidget(title, 1, 0)
        grid.addWidget(titleEdit, 1, 1)
```

```
grid.addWidget(author, 2, 0)
grid.addWidget(authorEdit, 2, 1)

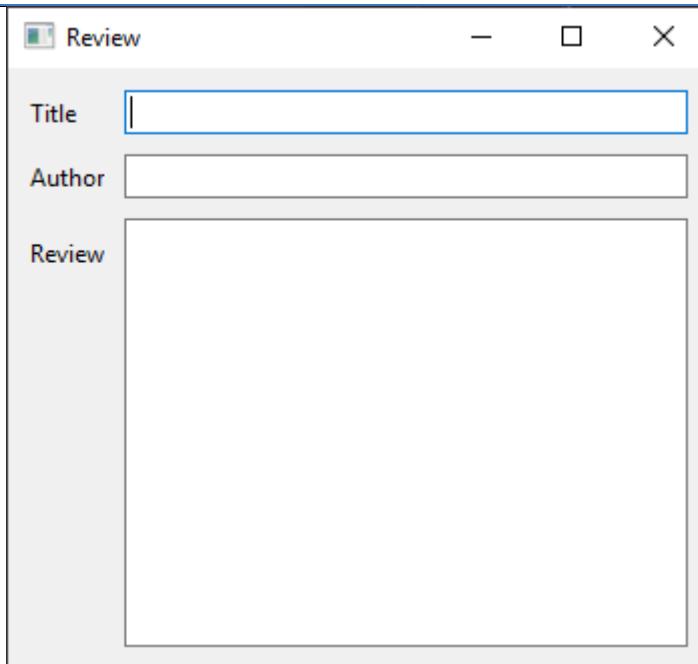
grid.addWidget(review, 3, 0)
grid.addWidget(reviewEdit, 3, 1, 5, 1)

self.setLayout(grid)

self.setGeometry(300, 300, 350, 300)
self.setWindowTitle('Review')

def main():
    app = QtWidgets.QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

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



Para finalizar

En este capítulo hemos dado una pequeña introducción al diseño GUI, no pretende ser exhaustiva ni completa, simplemente avanzar los conceptos principales de este tipo de programación, como son el diseño orientado a eventos, la creación de GIU a través de controles y las callbacks y su uso. Podemos acceder a la mayoría de ejemplos en la siguiente dirección.

<https://zetcode.com/gui/pysidetutorial/>

Ejercicios

Ejercicio 1 – ¿Par o impar?

Aplicación gráfica que permita introducir un número entero y luego saber si dicho número es par o impar. Utiliza un QLineEdit para introducir el valor, un QLabel para una etiqueta para anterior al QLineEdit, se determinará en el cambio del contenido de QLineEdit y un QLabel para mostrar “PAR” o “IMPAR” según el caso..

Ejercicio 2 – Mini calculadora I

Aplicación gráfica que permita introducir dos números reales y calcular el resultado de su suma, resta, multiplicación o división. Utiliza un QPushButton distinto para cada operación y un único QLabel para mostrar el resultado.

Ejercicio 3 – Mini calculadora II

Aplicación gráfica que permite introducir dos números enteros (A y B) y permita realizar tres cálculos distintos: sumatorio de A a B, productorio de A a B y exponencial A^B (A elevado a B). Utiliza tres QRadioButton (uno para cada cálculo) y un botón “¡Calcula!”.

Ejercicio 4 – Validar letra NIF

Aplicación gráfica que permita introducir un NIF (8 números y una letra utilizando un QLineEdit) e indique si la letra de dicho NIF es válida. Más información en este enlace:

<http://www.interior.gob.es/web/servicios-al-ciudadano/dni/calcu...o-de-control-del-nif-nie>

Ejercicio 5– Número aleatorio

Aplicación gráfica que permita obtener un número aleatorio cada vez que se pulse un botón. Deberá incluir un QSlider entre 1 y 100 que permita al usuario elegir el valor máximo del número aleatorio a generar. Así, por ejemplo, si el usuario pone el slider a 25, cada vez que le dé al botón se mostrará un número aleatorio entre 0 y 25.

Ejercicio 6 – Dados de Rol

Aplicación gráfica que permita al usuario simular que lanza un dado de juegos de rol. Podrá elegir entre dado de 6 caras (de 1 a 6), dado de 8 caras (de 1 a 8), dado de 10 caras (de 1 a 10), dado de 12 caras (de 1 a 12) y dado de 20 caras (de 1 a 20). Utiliza un botón distinto para cada tipo de dado. Muestra en cada botón una imagen de cada dado. Puedes encontrar las imágenes fácilmente haciendo una búsqueda en Internet.

Ejercicio 7 – Tablas de multiplicar

Aplicación gráfica que muestre las tablas de multiplicar del 1 al 10. Utiliza un QTable de 10 filas por 10 columnas y muestra los valores de cada tabla de multiplicar en una fila distinta.

Extra 1: Inserta los valores en la tabla en tiempo de ejecución, por ejemplo al presionar un botón.

Extra 2: Añade dos campos de texto A y B para que el usuario pueda introducir dos números enteros y se muestren todas las tablas de multiplicar desde A hasta B. Ten en cuenta que el número de filas es variable.

Ejercicio 8 – Inicio de sesión

Aplicación gráfica que simule una ventana de inicio de sesión y registro de usuarios. El usuario podrá introducir su nombre de usuario, contraseña y apretar un botón de “Iniciar sesión”. Muestra el resultado del intento de inicio de sesión en un MensageBox.

Los usuarios registrados y sus contraseñas estarán en el archivo ‘users.txt’. Crea unos pocos usuarios de ejemplo para probar la aplicación. No está permitido utilizar espacios ni en los nombres de usuario ni en las contraseñas.

Anexos

ANEXO I (Ahora qué)

Hemos terminado el curso y nos preguntamos ¿y ahora qué? El año que viene veremos muchos conceptos relacionados con la programación web

- Veremos javascript.
- Veremos un lenguaje de lado servidor: Php o Python – Django.
- Veremos cómo diseñar interfaces.

Y algunos otros relacionados con la instalación de servidores, empresas, etc.

Pero hasta llegar al curso que viene, este verano, podemos ir avanzando o, mejor dicho, no olvidando lo que hemos aprendido, por lo que os reto a hacer.

1. Diversas páginas web con todos los elementos, principalmente tablas y formularios.
2. Creación y consultas a una base de datos vuestra. No olvidar una gran ración de SELECT.
3. Escribir varias DTDs y los correspondientes ficheros XML basados en ellas.
4. Crear una pequeña aplicación en Python gráfica.
5. Terminar los ejercicios de Python en <https://exercism.io/>
6. Y por último descansar, que os lo habéis merecido, por lo menos tres semanas de desconexión, pero luego volver poco a poco a la actividad.



ANEXO II (Bibliografía)

- Python 3 Los fundamentos del lenguaje (3^a edición), Sébastien CHAZALLET, Ediciones ENI, ISBN 978-2-409-02478-8
- Curso de Programación Python (MANUALES IMPRESCINDIBLES), Arturo Montejo Ráez y Salud María Jiménez Zafra , Grupo Anaya Publicaciones Generales, ISBN 978-8441541160
- Learning Python, 5th Edition Fifth Edition, Mark Lutz, O'Reilly, ISBN 978-1449355739
- Fluent Python: Clear, Concise, and Effective Programming 1st Edition, Luciano Ramalho, O'Reilly, ISBN 978-1491946008
- Modern Python Cookbook - Second Edition, Steven F. Lott, O'Reilly, ISBN 9781800207455

Ejercicios

- <https://pythondiario.com/ejercicios-de-programacion-python>
- <http://progra.usm.cl/apunte/ejercicios/>
- <https://exercism.org/>

Documentación

- <https://www.python.org/>
- <https://python-docs-es.readthedocs.io/es/3.11/>

ANEXO III (Nuevas versiones)

Python 3.10

Patrón estructural

Hasta la versión 3.9 no existían una estructura sintáctica que reflejara el caso según (switch) de otros lenguajes, debíamos usar estructuras if-else anidadas. A partir de esta versión aparece este patrón que aumenta las capacidades de la sentencia switch (según) dando unas capacidades extendidas.

Caso implementado en otros lenguajes

```
match thing:
    case 1:
        print("thing is 1")  # Ver que no es necesario break
    case 2:
        print("thing is 2")
    case 3:
        print("thing is 3")
    case _:
        print("thing is not 1, 2 or 3")
```

```
match args.command:
    case 'push':
        print('pushing')
    case 'pull':
        print('pulling')
    case _:
        parser.error(f'{args.command!r} not yet implemented')
```

Caso con listas

```
for thing in [[1,2],[9,10],[1,2,3],[1],[0,0,0,0,0]]:
    match thing:
        case [x]:
            print(f"single value: {x}")
        case [x,y]:
            print(f"two values: {x} and {y}")
        case [x,y,z]:
            print(f"three values: {x}, {y} and {z}")
        case _:
            print("too many values")

for thing in [[1,2],[9,10],[3,4],[1,2,3],[1],[0,0,0,0,0]]:
    match thing:
        case [x]:
            print(f"single value: {x}")
        case [1,y]:
            print(f"two values: 1 and {y}")
        case [x,10]:
            print(f"two values: {x} and 10")
        case [x,y]:
            print(f"two values: {x} and {y}")
        case [x,y,z]:
```

```
        print(f"three values: {x}, {y} and {z}")
    case _:
        print("too many values")

for thing in [[1,2,3,4],['a','b','c'],"this won't be matched"]:
    match thing:
        case [*y]:
            for i in y:
                print(i)
        case _:
            print("unknown")
```

```
match event.get():
    case Click((x, y), button=Button.LEFT):
        handle_click_at(x, y)
    case Click():
        pass # ignore other clicks
    case KeyPress(key_name="Q") | Quit():
        game.quit()
    case KeyPress(key_name="up arrow"):
        game.go_north()
    ...
    case KeyPress():
        pass # Ignore other keystrokes
    case other_event:
        raise ValueError(f"Unrecognized event: {other_event}")
```

Python 3.11

Ubicaciones de error mejoradas

Al imprimir errores, el intérprete ahora señalará la expresión exacta que causó el error, en lugar de solo la línea.

Nueva anotación para clases: Self

La nueva anotación proporciona una forma simple e intuitiva de anotar métodos que devuelven una instancia de su clase:

```
class MyLock:
    def __enter__(self) -> Self:
        self.lock()
        return self
```

Velocidad mejorada

Se gana entre un 10% y un 60% de velocidad dependiendo el script ejecutado con respecto a las versiones anteriores, en media un 25%.

Excepciones a coste cero

La representación interna de las excepciones es diferente en Python 3.11. Los objetos de excepción son más livianos y el manejo de excepciones ha cambiado, por lo que hay poca sobrecarga en un bloque try ... except, siempre que la cláusula de excepción no se active.

Las llamadas excepciones de costo cero están inspiradas en otros lenguajes como C++ y Java. El objetivo es que el camino sin fallos, cuando no se plantee ninguna excepción, sea virtualmente libre de coste. Manejar una excepción aún llevará algún tiempo.

Python 3.12

Expresiones f"" mejoradas

Hasta ahora había algunas restricciones en el uso de f-strings. Los componentes de expresión dentro de las cadenas f pueden ser cualquier expresión válida de Python, incluyendo barras invertidas, secuencias escapadas unicode, expresiones multilínea, comentarios y cadenas que reutilicen la misma comilla que la cadena f que las contiene.

- **Reutilización de comillas:** en Python 3.11, la reutilización de las mismas comillas que la cadena f que la contiene genera un error de sintaxis, obligando al usuario a utilizar otras comillas disponibles (como comillas dobles o triples si la cadena f utiliza comillas simples). En Python 3.12, ahora se pueden hacer cosas como esta:

```
>>>
songs = ['Take me back to Eden', 'Alkaline', 'Ascensionism']
f"This is the playlist: {", ".join(songs)}"
'This is the playlist: Take me back to Eden, Alkaline,
Ascensionism'
```

Hay que tener en cuenta que antes de este cambio no había ningún límite explícito en cuanto a cómo se podían anidar las cadenas f, pero el hecho de que las comillas de cadena no se puedan reutilizar dentro del componente de expresión de las cadenas f hacía imposible anidarlas de forma arbitraria. De hecho, esta es la cadena f más anidada que se puede escribir:

```
>>>
f"""{f' {f'{f" {f" {1+1} "}}' }' }"""
'2'
```

Ahora las f-strings pueden contener cualquier expresión válida de Python dentro de componentes de expresión, incluso es posible anidar f-strings arbitrariamente:

```
>>>
f" {f" {f" {f" {f" {f" {1+1} "}" }" }" }" "
'2'
```

- **Expresiones multilínea y comentarios:** En Python 3.11, las expresiones f-strings debían definirse en una sola línea, incluso si las expresiones f-strings externas podían abarcar varias líneas (como las listas literales que se definen en varias líneas), lo que dificultaba su lectura. En Python 3.12 ahora puedes definir expresiones que abarquen varias líneas e incluir comentarios en ellas:

```
>>> f"This is the playlist: {}, ".join([
...     'Take me back to Eden',      # My, my, those eyes like fire
...     'Alkaline',                 # Not acid nor alkaline
...     'Ascensionism'             # Take to the broken skies at
last
... ])"
'This is the playlist: Take me back to Eden, Alkaline,
Ascensionism'
```

- **Barras invertidas y caracteres unicode:** antes de Python 3.12 las expresiones f-string no podían contener ningún carácter \. Esto también afectaba a las secuencias escapadas unicode (como \N{snowman}), ya que éstas contienen la parte \N que antes no podía formar parte de los componentes de expresión de las cadenas f. Ahora, puede definir expresiones como ésta:

```
>>>
print(f"This is the playlist: {"\n".join(songs)}")
This is the playlist: Take me back to Eden
Alkaline
Ascensionism
print(f"This is the playlist: {\N{BLACK HEART
SUIT}}.join(songs)}")
This is the playlist: Take me back to Eden♥Alkaline♥Ascensionism
```

Decorador @override

Se ha añadido un nuevo decorador `typing.override()` al módulo `typing`. Indica a las herramientas de programación que el método está destinado a sobrescribir un método de una superclase. Esto detecta errores cuando un método que pretende sobrescribir algo en una clase base no lo hace.

```
from typing import override

class Base:
    def get_color(self) -> str:
        return "blue"
```

```
class GoodChild(Base):
    @override # ok
    def get_color(self) -> str:
        return "yellow"

class BadChild(Base):
    @override # type checker error: colour no es color
    def get_colour(self) -> str:
        return "red"
```

Mejores mensajes de error

Los módulos de la biblioteca estándar se sugieren ahora como parte de los mensajes de error mostrados por el intérprete cuando se genera un `NameError`.

PEP 684: A Per-Interpreter GIL

Se pueden crear subintérpretes con un único GIL por intérprete. Esto permite a los programas Python aprovechar al máximo los múltiples núcleos de la CPU siendo por fin posible **la multitarea real bajo Python**. Este cambio se espera que genere grandes avances en el lenguaje.

<https://peps.python.org/pep-0684/>

ANEXO IV (F.A.Qs)

POO avanzado

Propiedades de clase

Una propiedad de clase es aquella definida en el ámbito de la clase y es común a todas las instancias que creamos. Para acceder utilizaremos el nombre de clase o el nombre del objeto o instancia.

Si accedemos a la variable de clase a través del nombre del objeto y la modificamos, crearemos una nueva variable de instancia con el mismo nombre en dicho objeto exclusivamente, no en los demás, quedando expuesta a los demás objetos como variable de clase.

```
class Coche:
    numero_instancias = 0 # variable de clase

    def __init__(self, matricula, color):
        self.matricula = matricula # Variables de instancia
        self.color = color # Variable de instancia
        Coche.numero_instancias += 1 # Acceso a variable clase

    def __str__(self):
        return f"{self.matricula}-{self.color}",
               "(Clase:{Coche.numero_instancias})"

    def __repr__(self):
        return self.__str__()

c1 = Coche("ABC-1900", "rojo")
print(f"c1={c1}")
c2 = Coche("XYZ-5600", "verde")
print(f"c1={c1}")
print(f"c2={c2}")

Coche.numero_instancias = 5 # Se accede a la de clase
print(f"c1={c1}")
print(f"c2={c2}")

c1.numero_instancias = 7 # se crea una de instancia en c1
print(f"c1={c1}")
print(f"c2={c2}")
print(f"c1.numero_instancias={c1.numero_instancias}")
print(f"c2.numero_instancias={c2.numero_instancias}") # Accedemos la la de clase
Coche.numero_instancias = 5
print(f"c1.numero_instancias={c1.numero_instancias}")
print(f"c2.numero_instancias={c2.numero_instancias}")
```

Resultado

```
C:/Python/Proyectos/pruebas/a.py
c1=ABC-1900 - rojo (Clase:1)
```

```
c1=ABC-1900 - rojo (Clase:2)
c2=XYZ-5600 - verde (Clase:2)
c1=ABC-1900 - rojo (Clase:5)
c2=XYZ-5600 - verde (Clase:5)
c1=ABC-1900 - rojo (Clase:5)
c2=XYZ-5600 - verde (Clase:5)
c1.numero_instancias=7
c2.numero_instancias=5
c1.numero_instancias=7
c2.numero_instancias=5
```

¿Cómo invoco a un método definido en una clase base desde una clase derivada que lo ha sobrescrito?

Usa la función incorporada super():

```
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

Polimorfismo

Por defecto Python no implementa el polimorfismo en los métodos, con la siguiente estructura es posible.

```
# person.py

from datetime import date
from functools import singledispatchmethod

class BirthInfo:
    @singledispatchmethod
    def __init__(self, birth_date):
        raise ValueError(f"No soportado: {birth_date}")

    @_init_.register(date)
    def _from_date(self, birth_date):
        self.date = birth_date

    @_init_.register(str)
    def _from_isoformat(self, birth_date):
        self.date = date.fromisoformat(birth_date)

    @_init_.register(int)
    @_init_.register(float)
    def _from_timestamp(self, birth_date):
        self.date = date.fromtimestamp(birth_date)

    def age(self):
        return date.today().year - self.date.year

class Person:
    def __init__(self, name, birth_date):
        self.name = name
        self._birth_info = BirthInfo(birth_date)

@property
```

```

def age(self):
    return self._birth_info.age()

@property
def birth_date(self):
    return self._birth_info.date

>>> from person import Person
>>> john = Person("John Doe", date(1998, 5, 15))
>>> john.age
>>> john.birth_date
>>> jane = Person("Jane Doe", "2000-11-29")
>>> jane.age
>>> jane.birth_date
>>> linda = Person("Linda Smith", 1011222000)
>>> linda.age
>>> linda.birth_date
>>> david = Person("David Smith", {"year": 2000, "month": 7,
"day": 25})

```

Iteradores

Implementación recomendada

Se presenta un problema potencial al crear un iterador sobre una estructura de datos si devolvemos la propia estructura de datos como el iterador en vez de una clase nueva que gestione los datos.

Si entregamos la propia estructura y creamos varios iteradores, todos compartirán el estado siendo unos dependientes de otros.

```

class Dias:
    def __init__(self):
        self.dias = ["L", "M", "X", "J", "V"]

    def __iter__(self):
        self._index = 0
        return self

    def __next__(self):
        if self._index < 0 or self._index > len(self.dias):
            raise StopIteration
        else:
            ret = self.dias[self._index]
            self._index += 1
        return ret

dias = Dias()
iter_uno = iter(dias)
iter_dos = iter(dias)
print(next(iter_uno)) # L
print(next(iter_dos)) # M

```

En este ejemplo vemos como ambos iteradores comparten estado ya que el comportamiento esperado es que el segundo empiece también el L, no siga por la M del primero.

```

class Dias:
    def __init__(self):
        self.dias = ["L", "M", "X", "J", "V"]

    def __iter__(self):
        return DiasIterador(self.dias)

class DiasIterador:
    def __init__(self, values):
        self.dias = values
        self._index = 0

    def __next__(self):
        if self._index < 0 or self._index > len(self.dias):
            raise StopIteration
        else:
            ret = self.dias[self._index]
            self._index += 1
        return ret

dias = Dias()
iter_uno = iter(dias)
iter_dos = iter(dias)
print(next(iter_uno)) # L
print(next(iter_dos)) # L

```

Varios

¿Por qué los valores por defecto se comparten entre objetos?

Considera esta función:

```

def foo(key, value, mydict={}):
    # Danger: el diccionario se comparte entre llamadas
    mydict[key] = value
    return mydict

print(foo(5, 3))
print(foo(6, 3))

# {5: 3}
# {5: 3, 6: 3} Guarda el resultado anterior entre llamadas

```

La primera vez que llamas a esta función, `mydict` solamente contiene un único elemento. La segunda vez, `mydict` contiene dos elementos debido a que cuando comienza la ejecución de `foo(6, 3)`, `mydict` comienza conteniendo un elemento de partida.

A menudo se esperaría que una invocación a una función cree nuevos objetos para valores por defecto. Eso no es lo que realmente sucede. **Los valores por defecto se crean exactamente una sola vez, cuando se define la función.** Si se cambia el objeto, como el diccionario en este ejemplo, posteriores invocaciones a la función estarán referidas al objeto cambiado.

Por definición, los objetos inmutables como números, cadenas, tuplas y None están asegurados frente al cambio. Cambios en objetos mutables como diccionarios, listas e instancias de clase pueden llevar a confusión.

Debido a esta característica es una buena práctica de programación el no usar valores mutables como valores por defecto. En su lugar usa None como valor por defecto dentro de la función, comprueba si el parámetro es None y crea una nueva lista/un nuevo diccionario/cualquier otra cosa que necesites. Por ejemplo:

No usar:

```
def foo(mydict={}):
    ...
```

Pero sí:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {} # create a new dict for local namespace
```

¿Cuál es la forma más eficiente de concatenar muchas cadenas conjuntamente?

Los objetos str y bytes son inmutables, por tanto, concatenar muchas cadenas en una sola es ineficiente debido a que cada concatenación crea un nuevo objeto. En el caso más general, el coste total en tiempo de ejecución es cuadrático en relación a la longitud de la cadena final.

Para acumular muchos objetos str, la forma recomendada sería colocarlos en una lista y llamar al método str.join() al final:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

¿Cómo puedo crear una lista multidimensional?

Seguramente hayas intentado crear un array multidimensional de la siguiente forma:

```
A = [[None] * 2] * 3
```

Esto parece correcto si lo muestras en pantalla:

```
print(A)
# [[None, None], [None, None], [None, None]]
```

Pero cuando asignas un valor, se muestra en múltiples sitios:

```
A[0][0] = 5
print(A)
# [[5, None], [5, None], [5, None]]
```

La razón es que replicar una lista con * **no crea copias, solo crea referencias a los objetos existentes**. El *3 crea una lista conteniendo 3 referencias a la misma lista de longitud dos. Cambios a una fila se mostrarán en todas las filas, lo cual, seguramente, no es lo que deseas.

El enfoque recomendado sería crear, primero, una lista de la longitud deseada y, después, llenar cada elemento con una lista creada en ese momento:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

Esto genera una lista conteniendo 3 listas distintas de longitud dos. También puedes usar una comprensión de lista:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

¿Por qué list.sort() no devuelve la lista ordenada?

En situaciones donde el rendimiento es importante, hacer una copia de la lista solo para ordenarla sería un desperdicio. Por lo tanto, `list.sort()` ordena la lista en su lugar. Para recordarle ese hecho, no devuelve la lista ordenada. De esta manera, no se dejará engañar por sobrescribir accidentalmente una lista cuando necesite una copia ordenada, pero también deberá mantener la versión sin ordenar.

Si desea crear una nueva lista, use la función incorporada `sorted()` en su lugar. Esta función crea una nueva lista a partir de un iterativo proporcionado, la ordena y la retorna.

Utilizar el color ASCII

```
class bgcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKCYAN = '\033[96m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    RESTORE = '\033[0m'
    SELECT = '\33[7m'

class fgcolors:
    BLACK = '\033[0;30m' # Black
    RED = '\033[0;31m' # Red
    GREEN = '\033[0;32m' # Green
    YELLOW = '\033[0;33m' # Yellow
    BLUE = '\033[0;34m' # Blue
    PURPLE = '\033[0;35m' # Purple
    CYAN = '\033[0;36m' # Cyan
    WHITE = '\033[0;37m' # White
    RESTORE = '\033[0m'
    SELECT = '\33[7m'

print(fgcolors.RED + "Hola" + fgcolors.RESTORE)
print(bgcolors.SELECT + bgcolors.WARNING + "Hola" + \
bgcolors.RESTORE)
print("Hola")
```

Cambio de librerías recomendados

Pathlib en vez de os

El módulo `os` representa el path como una cadena de texto, mientras que `pathlib` usa un estilo orientado a objetos e independiente del SSOO.

```
from pathlib import Path
import os.path

# Old, Unreadable
two_dirs_up= \
    os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# New, Readable
two_dirs_up = Path(__file__).resolve().parent.parent
```

```
readme = Path("README.md").resolve()
print(f"Absolute path: {readme.absolute()}")
# Absolute path: /home/martin/some/path/README.md
print(f"File name: {readme.name}")
# File name: README.md
print(f"Path root: {readme.root}")
# Path root: /
print(f"Parent directory: {readme.parent}")
# Parent directory: /home/martin/some/path
print(f"File extension: {readme.suffix}")
# File extension: .md
print(f"Is it absolute: {readme.is_absolute()}")
# Is it absolute: True
```

Se puede usar el operador `/` para unir dos paths:

```
# Operators:
etc = Path('/etc')
joined = etc / "cron.d" / "anacron"
print(f"Exists? - {joined.exists()}") # Exists? - True
```

Se pueden usar las siguientes correspondencias entre el módulo `os` y el `pathlib`:

```
print(f"Working directory: {Path.cwd()}")
# same as os.getcwd(), Working directory: /home/martin/some/path
Path.mkdir(Path.cwd() / "new_dir", exist_ok=True)
# same as os.makedirs()
print(Path("README.md").resolve())
# same as os.path.abspath()
# /home/martin/some/path/README.md
print(Path.home())
# same as os.path.expanduser()
# /home/martin
```

<https://docs.python.org/es/3/library/pathlib.html>

Dataclasses en vez de namedtuple

El remplazo se basa sobre todo en las siguientes características no presentes en namedtuple:

- Se pueden modificar.
- Proporciona implementaciones por defecto para: `__repr__`, `__eq__`, `__init__`, `__hash__`.
- Permite valores por defecto.
- Soporta herencia.

ANEXO V (Diseño POO Robusto)

Qué es el software robusto

Para definir el software robusto, podríamos acudir a las definiciones técnicas encontradas en los libros de POO, a los clientes o a los programadores. Ninguna de estas definiciones completa el concepto de software robusto. Para nosotros, software robusto es aquel que:

Cumple con las expectativas del cliente
y
está bien diseñado, bien codificado, es fácil de mantener, reusar y extender

Creación del software en tres pasos

1. Asegurarse que el software hace lo que el cliente espera que haga.
2. Aplicar principios básicos de la POO.
3. Escribir código mantenable y reusable.

La máxima del software es que siempre Cambiará

Pequeños proyectos

Creación de requerimientos y casos de uso

Qué es un requerimiento: Es una necesidad que el sistema debe satisfacer para trabajar correctamente.

- Buenos requerimientos aseguran que el sistema hará lo que el cliente espera.
- Los requerimientos deben cubrir todos los casos de uso.
- Revisar los casos de uso para determinar lo que el cliente haya olvidado decírnos.
- Los casos de uso revelarán cualquier requerimiento que falte o esté incompleto que haya que añadir al sistema.
- Los requerimientos siempre van a cambiar.
- La mejor manera de crear los requerimientos es conociendo que se supone que hace el sistema.

Qué es un caso de uso: Describe qué hace el sistema para completar una necesidad del cliente. Es una técnica para capturar posibles requerimientos en el sistema. Cada caso de uso proporciona uno o más escenarios que tenemos que cubrir y determinar cómo el sistema debe interactuar con el usuario para solucionar una tarea específica.

- Escribirlos de forma que los entienda tanto el jefe de proyecto como el cliente.
- Deben reflejar el sistema en el contexto del mundo real.
- Tener especial atención a los nombres y verbos para la creación de las clases posteriormente.
- Un buen caso explica de forma clara y precisa lo que hace el sistema.

Principios de la POO

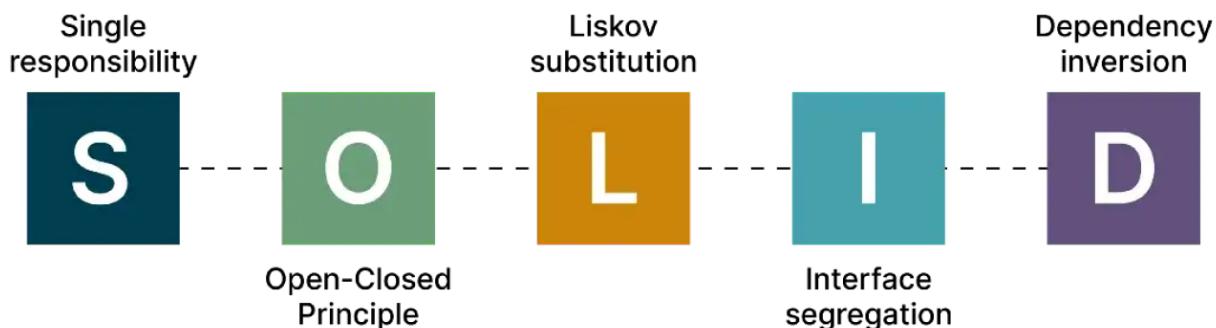
La aplicación de estos principios nos proporcionará flexibilidad en el diseño y dotará de facilidad a la hora de cambiar. Parecen lógicos tener en cuenta los siguientes principios:

- **Encapsular** lo que varía.
 - Cuando se tenga un conjunto de propiedades que cambian, lo más acertado es crear un diccionario de propiedades.
- **Delegación.** Desplazar la implementación de las partes que cambian en nuestros objetos a otros, de tal manera que agrupamos lo que cambia en un nuevo objeto.
- Codificar para el interfaz en vez de para la implementación.
- **Principio de causa única:** Cada clase debe tener un único motivo para ser cambiada.

Principios OPP revisados

Según ha avanzado la programación los principios anteriores se han quedado obsoletos y se han modernizado en los siguientes:

- **S - Principio de responsabilidad única (SRP).** Cada objeto en el sistema debería tener una única responsabilidad o razón para cambiar.
- **O - Principio de Abierto – Cerrado (OCP).** Las clases tienen que estar abiertas para la extensión y cerradas a la modificación. Proporciona flexibilidad. Ser capaz de añadir nuevo código sin modificar el existente.
- **L - Principio de sustitución de Liskov (LSP).** Las partes del código que usan clases deben poder usar objetos de clases derivadas sin necesidad de cambio.
- **I - Principio de segregación de interfaz.** “Un cliente no debe verse obligado a implementar una interfaz que no utiliza”. Es hacer que las interfaces (clases abstractas principales) sean más específicas, en lugar de genéricas.
- **D – Principio de inversión de dependencias.** El sistema “Depende de abstracciones, no de concreciones”. Es hacer que las clases dependan de clases abstractas en lugar de clases no abstractas.



- **Principio de no repetirse (DRY).** Evitar en todo momento código duplicado. Proporciona que cada requerimiento esté en un único sitio.
- **Principio de mantener el código simple (KISS).** La mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de complicarlos; por lo tanto, la simplicidad debe ser un objetivo clave en el diseño y la complejidad innecesaria debería ser evitado.
- **Principio Test Cases Before Your Feature (TDD) Crear los casos antes de prueba antes que la funcionalidad.** El software de calidad no existe sin pruebas. Cuanto antes descubramos problemas en el código, menos impacto tendrán, por lo que es aconsejable crear los casos de prueba antes de programar la característica.
- **Principio de no reinventar la rueda, usar patrones.** Si ya está hecho, no volver a crearlo nosotros, debemos aprender los patrones de programación existentes.

Cohesión. Mide el grado de conectividad entre los elementos de un módulo, clase u objeto. A mayor cohesión más fácil es que el software esté bien diseñado. Para estar seguros que tenemos la máxima cohesión nos preguntaremos sobre cada parte:

- Cuánto es de fácil cambiar un elemento.
- Está realmente bien diseñado.

Dependencia. Es la cantidad información necesaria de otras partes de la aplicación en nuestras funciones o métodos.

Acoplamiento. Es un concepto que mide la dependencia entre dos módulos distintos de software, como pueden ser por ejemplo las clases. El acoplamiento puede ser de dos tipos:

- Acoplamiento débil, que indica que no existe dependencia de un módulo con otros. Esto debería ser la meta de nuestro software.
- Acoplamiento fuerte, que por lo contrario indica que un módulo tiene dependencias internas con otros.

El término acoplamiento está muy relacionado con la cohesión, ya que acoplamiento débil suele ir ligado a cohesión fuerte. En general lo que buscamos en nuestro código es que tenga acoplamiento débil y cohesión fuerte, es decir, que no tenga dependencias con otros módulos y que las tareas que realiza estén relacionadas entre sí. Un código así es fácil de leer, de reusar, mantener y tiene que ser nuestra meta.

- **Principio de máxima cohesión – mínimo acoplamiento.** Si ya está hecho, no volver a crearlo nosotros, debemos aprender los patrones de programación existentes.

Principios de análisis y diseño

- El software bien diseñado es fácil de cambiar y extender.
- Se usan principios básicos de POO en la creación para ser más flexible.
- Si un diseño no es flexible, hay que cambiarlo, no dejarlo como está.
- Estar seguros que cada clase tiene máxima cohesión, es decir, solo hace una única cosa.
- Siempre buscar la máxima cohesión a lo largo del desarrollo del proyecto.

Grandes proyectos

Divide y vencerás

Los grandes problemas se pueden solucionar dividiéndolos en otros más pequeños (partes funcionales) y después centrándonos en cada una de las partes de forma individual

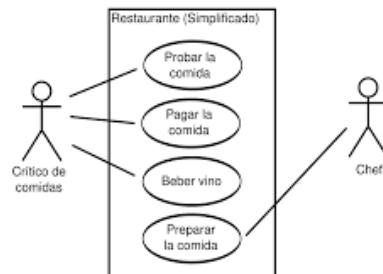
Creación de características

Una característica es una descripción de alto nivel sobre lo que el sistema tiene que hacer. Estas características se suelen conseguir en conversaciones con el cliente o los usuarios. De cada característica obtendremos los requerimientos y los casos de uso.



Diagramas de casos de uso

Los diagramas de casos de uso se crean una vez analizadas todas las características y creados todos los casos de uso. Son los planos que se van a utilizar para crear el sistema. Una vez creado el diagrama, se validará con la lista de características y requerimientos. Los diagramas de casos de uso son sobre lo que el sistema es sin mucho detalle.

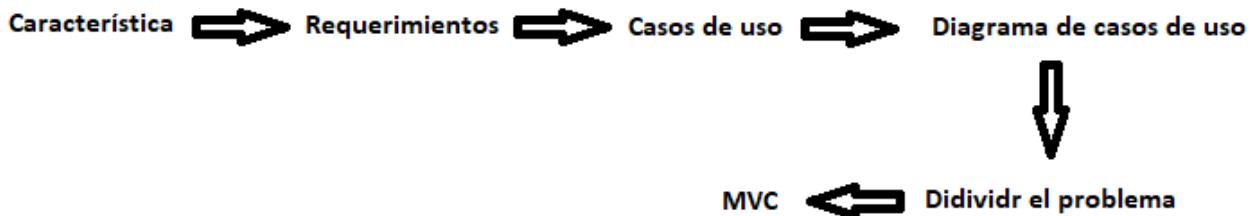


Análisis del dominio

Es el proceso por el que se identifica, recoge, organiza y representa toda la información relevante de un dominio basado en el estudio del sistema existente. Este estudio permite crear partes de nuestro sistema que NO tenemos que construir también.

Uso del patrón MVC

Uso del patrón Modelo – Vista – Controlador permitirá un mejor diseño.



Resumen

- Escuchar al cliente y entender lo que necesita.
- Escribir una lista de características en un lenguaje que se entienda por el cliente.
- Asegurarse que dicha lista es lo que quiere el cliente.
- Crear mapas del sistema usando el diagrama de casos de uso.
- Divide el problema en partes más pequeñas.
- Aplicar los patrones de diseño a las partes anteriores.
- Usar los principios básicos de la POO en cada elemento.

Arquitectura

La arquitectura de un sistema es la organización estructural del mismo, incluye la descomposición en partes, la unión entre las diferentes partes, los mecanismos de interacción, los principios de diseño y las decisiones sobre el sistema. La arquitectura nos sirve para unir toda la documentación anterior de una forma ordenada.

- Los elementos realmente importantes para nuestra aplicación son los que también son representativos para la arquitectura. Son en los primeros que tendríamos que fijarnos. Para determinar si forma parte o no de la arquitectura nos preguntaremos:
 - ¿Es parte esencial del sistema?
 - Si no sabemos lo que significa una característica, es probable que tengamos que centrarnos más hasta entenderla.
- Hay que centrarse en una característica a la vez para minimizar el riesgo en nuestro proyecto. Hay que dejar de lado todo aquello que no reduzca el riesgo del proyecto.
- Una de las partes más importantes de la arquitectura son las relaciones entre las distintas partes de la misma.

Relaciones entre los objetos

- Herencia.
- Delegación. Traspasar la responsabilidad de una tarea a otra clase o método. Se suele usar cuando queremos utilizar la funcionalidad de otra clase, pero no queremos que cambie dicha funcionalidad.
- Composición. Se usa cuando una clase necesita de varias otras para realizar toda su tarea. Si un objeto utiliza la composición, cuando sea destruido también dichos objetos que lo componen deberán serlo.
- Agregación. Similar a la anterior, pero el objeto que ahora es parte también existe fuera del objeto que lo utiliza.

Repetir para terminar

El software se escribe iterativamente, una vez finalizada una característica seguimos con la siguiente. Pero hay dos aproximaciones para el desarrollo: Dirigido a características o dirigido a casos. Cualquiera de las dos es factible usar. En la primera elegimos característica a característica hasta que está completa, en el segundo se elige un caso de uso (del diagrama) y se desarrolla de forma completa, aunque implique a varias características.

Pero qué elemento usamos para empezar o cuál sería la siguiente: el que reduzca el riesgo del proyecto.

Hay una tercera aproximación es el desarrollo dirigido por pruebas, en el que primero se realizan las pruebas que debe superar una parte, se programa y se realiza la prueba, en caso afirmativo se pasa al siguiente paso.

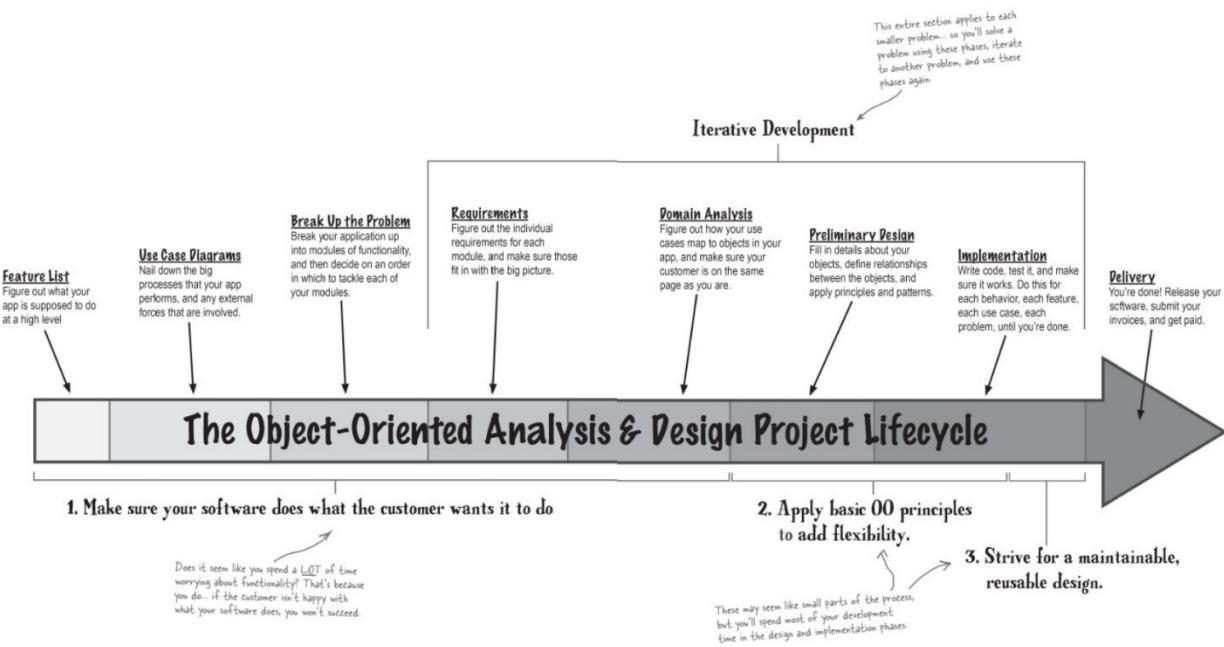
Los desarrollos reales usan una mezcla de las tres.

Tipos de programación

- **Por contrato:** Definimos el API y no se modifica, confiamos en que el programador lo use tal y como lo hemos definido, en caso que no sea así se genera un error de ejecución.
- **Programación defensiva:** Controlamos que el uso del API sea correcto en todo momento, no confiamos en que el programador lo use de forma debida siempre.

Ejemplo (Cap. 10 Head First Object-Oriented Analysis&Design – Brett D. McLaughlin)

Resumen de los puntos anteriores



Supuesto

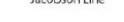
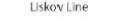
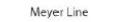
Queremos proporcionar a los turistas una manera fácil de ver las maravillosas atracciones que hacen la ciudad única. Necesitamos una aplicación (RouterFinder) que debería servir para almacenar todas las rutas de metro de la ciudad, a la vez que las paradas de cada línea. El metro en nuestra ciudad permite desplazarse en cualquier sentido entre dos estaciones por lo que no hay que preocuparse por la dirección de las líneas.

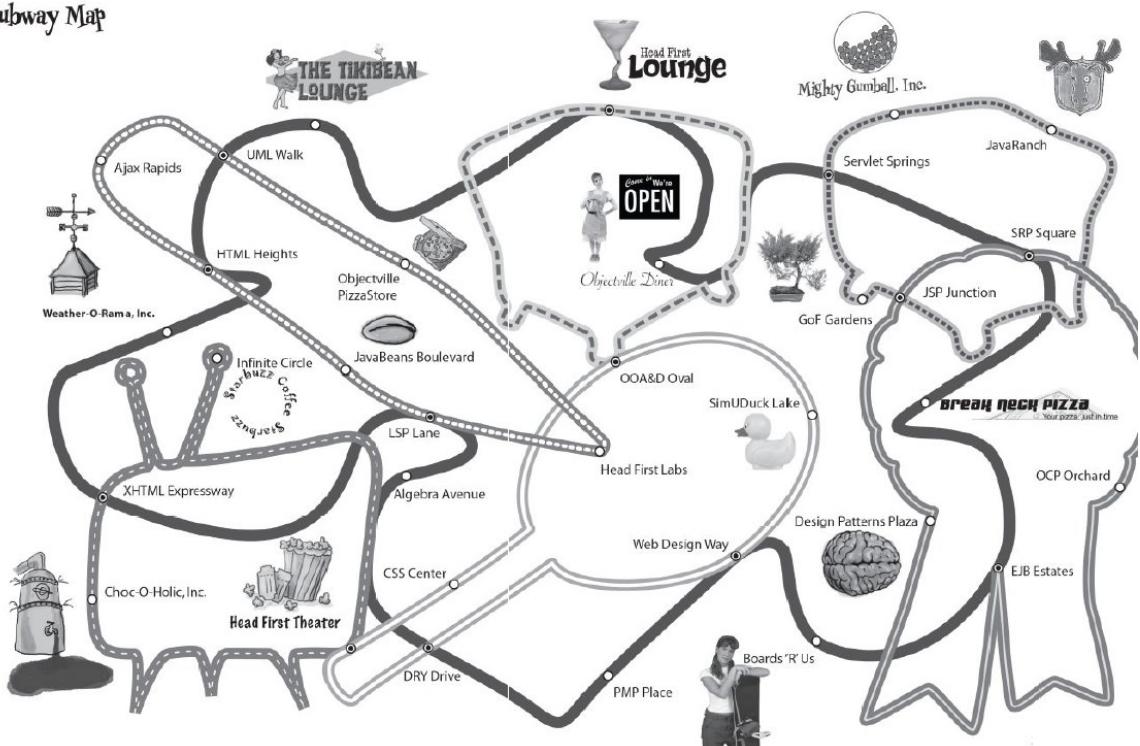
La App debe ser capaz de recoger una estación inicial y una final, encontrando la ruta entre ellas. Además, estas rutas se deben de imprimir, indicando qué líneas coger en qué estaciones cambiarse y qué estaciones recorre la ruta.

Deseamos que la App sea flexible y extensible, y que en un futuro aparecerán nuevas rutas y estaciones.
Añadimos un ejemplo de las rutas actuales:

Objectville Subway Map

Legend

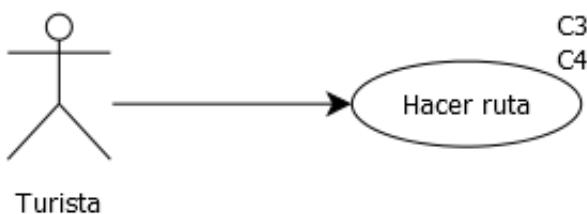
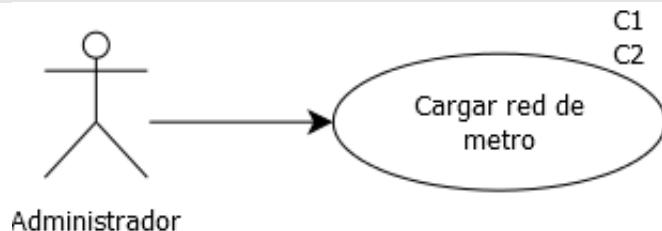
- Local train station
- Interchange with other lines
-  Bach Line
-  Gamma Line
-  Jacobson Line
-  Liskov Line
-  Meyer Line
-  Rumbaugh Line
-  Wirfs-Brock Line



Paso 1: Lista de características

- C1: Hay que representar una línea de metro y las estaciones que hay a lo largo de ella en el orden correcto.
- C2: El programa gestionará múltiples líneas de metro, incluyen el que se solapan unas con otras.
- C3: Tendremos que generar una ruta adecuada entre dos estaciones cualesquiera de cualquier línea.
- C4: Debe ser posible imprimir la ruta obtenida entre dos estaciones, así como la dirección.

Paso 2: Diagrama de casos de Uso



El primer caso de uso responde a las características 1 y 2, el segundo a las 3 y 4. Las características y los casos de uso no son lo mismo, aunque están relacionadas, tal y como vemos en el diagrama. En el supuesto que no encajemos una característica en un caso de uso, tal vez nos tendremos que replantear la necesidad real de dicha característica.

Paso 3: Divide y vencerás, haciendo partes más pequeñas.

En este paso se dividirá el problema en partes más pequeñas para poder hacer frente a cada una de ellas. No hay una manera “correcta” de hacerlo, en este caso queremos que la aplicación sea lo más modular posible, con lo que cada módulo implementará una única responsabilidad.

- Módulo Metro: representa el metro completo, las rutas, las estaciones y conoce cómo crear rutas entre dos puntos de la red.
- Módulo Cargador: Contendrá todos los mecanismos para cargar el módulo metro, desde una BBDD, desde un fichero, etc. Este módulo es diferente ya que la responsabilidad que implementa se podría reutilizar en otros proyectos, no solo en el metro.
- Módulo Imprimir: Al igual que el anterior al no formar parte del problema en sí, al poder ser compartido entre diferentes proyectos, al poder usar diferentes mecanismos de impresión debe estar separado.

Los módulos anteriores aparecen una vez estudiados los casos de uso en el diagrama y las características necesarias, pero no es suficiente, hay que añadir un módulo más:

- Módulo Pruebas: Este módulo no pertenece al sistema en sí, pertenece al proceso de Análisis – Diseño de la POO que nos garantizará la validez del proyecto ante nosotros y ante el cliente.

Paso 4: Iterar para encontrar los requerimientos

En este paso recorreremos todos los casos de uso uno por uno para encontrar los requerimientos de forma iterativa, empezaremos por uno u lo refinaremos hasta dar con todos los requerimientos antes de pasar al siguiente.

En caso que no tengamos un conocimiento completo del problema, tendremos que hacer un sub-paso “conocer el problema completamente” antes de seguir con el proceso.

Paso 4.1: Cargar red de metro

Paso 4.1.1: Conocer el problema

- Qué es una estación: Es un punto en el mapa con un nombre.
- Qué es una conexión: La unión entre dos estaciones.
- Qué es una línea: Un conjunto de conexiones que puede ser abierta o cerrada, pero en el mapa proporcionado todas las líneas son cerradas.

Estudio del fichero que nos van a mandar para cargar la red de metro:

Listado de todas las estaciones, una por línea

Espacio en blanco

Nombre línea

Listado de estaciones en orden

Espacio en blanco

Nombre línea

.....

Estacion_1

Estación_2

Estación_3

Línea 1

Estación_1

...

Estacion_1

Línea 2

Estación_3

....

Al comienzo del fichero se listarán todas las estaciones, a continuación, separadas por líneas en blanco aparecerá cada línea, en la cabecera el nombre de la línea y a continuación en orden cada estación, al ser circular la línea la primera y la última estación será la misma.

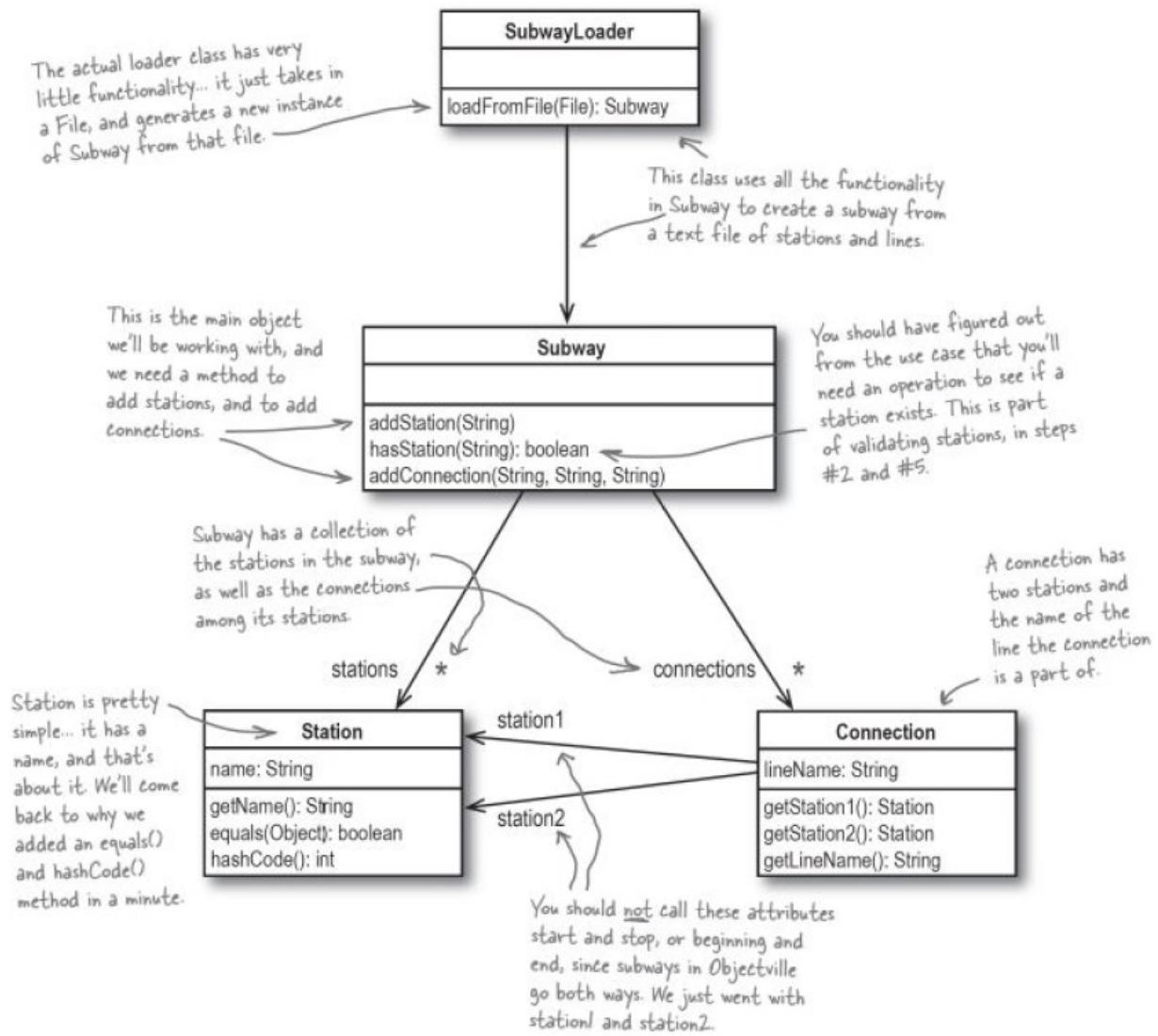
Paso 4.1.2: Requerimientos: casos de uso

1. El administrador proporciona el fichero a cargar con la estructura correcta.
2. El sistema lee el nombre de una estación.
3. El sistema valida que no existe ya la estación en el sistema.
4. El sistema añade la estación del metro al sistema.
5. Se repiten los pasos 2 al 4 hasta cargar todas las estaciones
6. El sistema lee el nombre de una línea.
7. El sistema lee dos estaciones conectadas de la línea.
8. El sistema valida que ambas estaciones existen en el sistema.
9. El sistema crea una nueva conexión entre las estaciones en la línea actual.
10. Se repiten los pasos 7 al 9 hasta finalizar la línea
11. Se repiten los pasos 6 al 10 hasta finalizar todas las líneas.

Paso 4.1.3: Análisis del dominio

En la lista de requerimientos o casos de uso buscamos los nombres y verbos que nos van a denotar las características y las acciones.

- **Nombres:** Administrador, fichero, sistema, estación, metro, conexión. Línea
 - Administrador no es candidata al estar fuera del sistema, es un actor.
 - Fichero es una entrada al sistema.
 - Sistema es el propio sistema
 - El resto son válidos para tener en cuenta.
- **Verbos:** Proporciona, lee, valida estación, añade estación, repite, crea una conexión.
 - Lee es una primitiva de programación.
 - Repite es una primitiva de programación.

Paso 4.1.4: Diseño PreliminarPaso 4.1.5: Implementación

Ahora es el momento de trasladar el paso 3 y 4 a código.

Paso 4.2: Hacer rutaPaso 4.2.1: Conocer el problema

El diseño de rutas no es un problema sencillo que se debería estudiar en profundidad antes de seguir adelante. Cómo crear la ruta, cómo buscar la más eficiente, etc.

Paso 4.2.2: Requerimientos: casos de uso

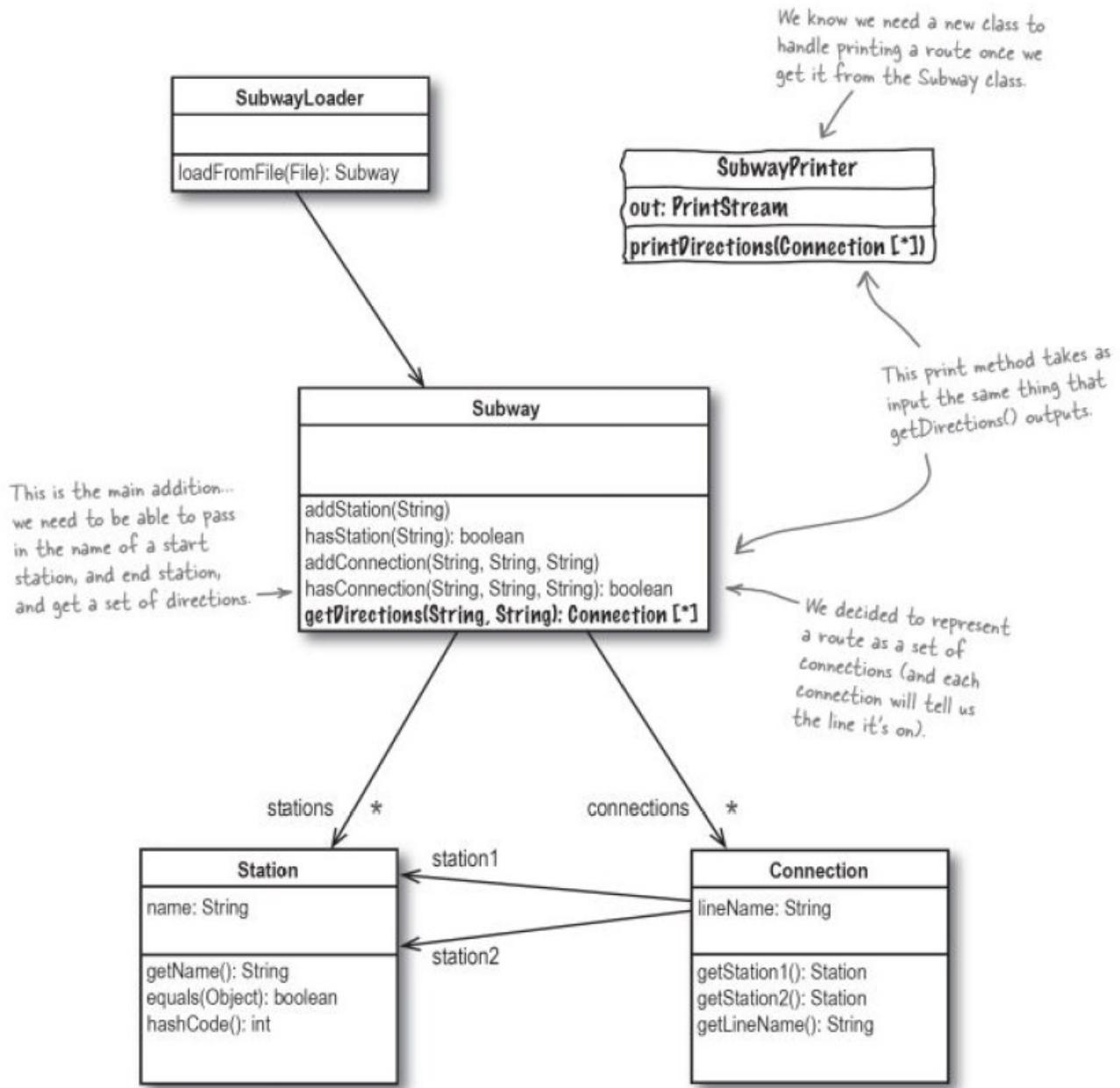
1. El turista pide al sistema una ruta, proporcionando la estación de origen y la de destino.
2. El sistema valida que ambas estaciones existen en el metro.
3. El sistema calcula la ruta desde el origen a destino.
4. El sistema imprime la ruta calculada.

Paso 4.2.3: Análisis del dominio

- **Nombres:** Turista, ruta, estación, sistema, metro, origen, destino.
 - Turista no es candidata al estar fuera del sistema, es un actor.
 - Sistema es el propio sistema

- El resto son válidos para tener en cuenta.
- **Verbos:** pide, proporciona, valida, calcula, imprime.

Paso 4.2.4: Diseño Preliminar



Paso 4.2.5: Implementación

Ahora es el momento de trasladar el paso 3 y 4 a código.

Paso 4.3: Implementar Test

En este caso la implementación de las pruebas se hace al final del proceso en este ejemplo, pero en mi opinión, este módulo se debe de ir implementando a la vez que se hacen los pasos 4.1.5 y 4.2.5 para dotar de mayor flexibilidad al diseño y encontrar antes los posibles errores del sistema y los algoritmos.

ANEXO VI (Escribir código limpio)

Por qué de la necesidad de escribir código limpio

Las nociones de diseño controlado únicamente por pruebas y el código, propias de finales de la década de 1990 ya no son válidas. El 80% del software o más de lo que hacemos se denomina mantenimiento, es decir, reparar el software. Por tanto, crear código legible es tan importante como crear código ejecutable (Fred Brooks postulaba que se debería rehacer el software cada siete años para eliminar los problemas latentes).

Definir calidad en el ámbito del software es difícil, pero podemos decir que es el resultado de un millón de acciones cuidadosas, no de un método magnífico caído del cielo, y el aprendizaje para crear código limpio es complicado, se requiere el uso disciplinado de técnicas aplicadas mediante un detallado sentido de la corrección.

Antes de introducirnos en la creación del código limpio, tendremos que poder identificar cuál no es código limpio, o lo que es lo mismo cuál es código incorrecto, para poder cambiarlo siguiendo unos principios que nos lleven hasta el código limpio. Como programadores, el primer principio que tendremos en cuenta será:

La ley de LeBlanc: Despues es igual a nunca.

Hoy en día la creación de Código limpio no es algo baldío, es rentable y es una cuestión de supervivencia profesional, aquel que no cree Código fácilmente mantenable posiblemente tendrá grandes problemas para mantener su trabajo.

Definir qué es código limpio es otra tarea ingente, pero grandes programadores han dado su visión de lo que es para ellos:

- Hace bien una cosa.
- Muestra gran atención al detalle.
- Se puede leer como un texto bien escrito.
- Se puede leer y mejorar por un programador que no es su autor original.
- Es sencillo y se ha prestado atención a los detalles.
- Evita las duplicidades, maximiza la expresividad y diseña abstracciones sencillas.

Hemos argumentado que el código debe ser fácil de mantener ya que el mundo actual tan cambiante (y las nuevas metodologías ágiles) van hacer que nuestro proyecto tenga que ser modificado después de ser creado. Y para que se pueda modificar por parte de un programador distinto al que lo ha creado, este tendrá que investigar y leer nuestro código para entenderlo, pasando a la modificación posteriormente. Como vemos, una tarea fundamental de nuestro programador será la lectura, por lo que si el código se puede entender fácilmente (código limpio) será más fácil de modificar. Si pensamos en las consecuencias de este párrafo, veremos que nosotros tendremos que modificar también código, por lo que agradeceremos que se pueda entender fácil y no tener que gastar horas en su comprensión. Por tanto, si nos encontramos en esta situación tendremos en cuenta el segundo principio:

Al modificar un Proyecto, dejar el Código un poco más limpio que como lo hemos encontrado.

Nombrado

El nombre que damos a una variable, función o método determinará más que lo que creemos su uso. Debe indicar pro qué existe, qué hace y cómo se usa, si tenemos que utilizar un comentario para explicar cualquiera de esas tres preguntas, significará que no es un buen nombre, con lo que deberemos replanteárnoslo. La elección de los nombres debe ser cuidadosa, y seguir las siguientes normas:

- En caso que utilice algún tipo de medida, incluirla en el nombre: tiempo_transcurrido_en_dias.

- Evitar usar nombres pobres o que dificulten el significado: hp sería un nombre muy mal elegido por ejemplo para la hipotenusa.
- Evitar usar variaciones mínimas en los nombres.
- Usar nombres iguales en significados similares y nombres diferentes para significados diferentes.
- No incluir nunca la palabra variable en el nombre de una variable o función o método en los correspondientes.
- Usar nombres que se puedan leer.
- Usar nombres que se puedan buscar, es más fácil buscar MAX_ALUMNOS que el número 7.
- No hay que tener miedo a usar nombres largos.
- En el caso de usar patrones, se debe usar el nombre de dicho patrón en el nombre de la clase que lo implementa.
- Los nombres de las variables y Clases deben ser nombres y el de los métodos y funciones verbos que expresen claramente su funcionalidad.
- Usar una palabra por cada concepto (se debe mantener a lo largo de todo el programa) evitando usar la misma palabra para dos fines distintos. Usar un léxico coherente es básico para los programadores que usen el código posteriormente.
- Siempre que se pueda hay que usar términos informáticos, algorítmicos, nombres de patrones, términos matemáticos, etc. ya que los programadores los entenderán más fácilmente.
- En caso de uso de nombres aislados, añadir al nombre algo del contexto en el que se usa: dirección_ciudad, dirección_calle, etc.

Diseño de funciones

Para el diseño de funciones tendremos en cuenta las siguientes reglas:

- Deben ser de tamaño reducido, unas 20 líneas estaría bien.
- El nivel de sangrado no debe ser mayor de dos o tres niveles, en caso de mayor número refactorizar.
- Las funciones deben seguir el principio de Responsabilidad única (SRP): Hacer una única cosa, hacerla bien.
- Las funciones no deben tener secciones diferenciadas, es una clara violación del SRP y se debería refactorizar.
- Todas las operaciones dentro de la función deben tener el mismo nivel de abstracción, no debe una instrucción crear una página desde una plantilla y utilizar un flujo para escribirla.
- El código se leerá siempre de arriba abajo.
- El uso de sentencias match dificulta el tamaño, se debería implementar el patrón factoría abstracta en este caso.
- Debe seguir las normas de nombre del punto anterior.
- El número de argumentos de una función será mínimo, no más de tres, en caso de necesidad debe estar muy justificado. En este caso, estudiar la posibilidad de crear una clase de datos con todos los argumentos relacionados y pasar el objeto en vez de los datos separados.
- No se utilizará un argumento como un interruptor de funcionalidad de la función, si es verdadero hace una cosa y si es falso hace otra, como mínimo incumple el SRP.
- El nombrado de los argumentos seguirá las recomendaciones de las variables.
- Las funciones siguen el principio de SRP, por lo que harán lo que exprese su nombre, nunca realizarán funciones ocultas.
- Con los nuevos mecanismos de gestión de errores, evitaremos devolver códigos de error en las funciones, en todo caso generarán excepciones.
- Las funciones seguirán el principio de no repetirse a uno mismo (DRY), o lo que es lo mismo, evitar duplicidad de código.
- Al crear funciones reducidas, podemos saltarnos el principio de Dijkstra de tener una única salida ya que será más significativo.

Comentarios

No hay nada más útil que un comentario bien colocado, pero por norma general dentro de la perspectiva del código limpio, los comentarios son siempre fallos, si los usamos es porque no hemos sabido expresar con código de forma correcta nuestra intención. Además, los programadores tendemos a no actualizar los comentarios, con lo que se vuelve en una fuente de información imprecisa y dañina, al contener datos no actualizados. Los comentarios que no deberemos de omitir serán aquellos comentarios que se insertan por motivos legales, deben de mantenerse y actualizarse junto con el resto del código.

De todas maneras, los comentarios son útiles y serán tenidos en cuenta para las siguientes situaciones:

- Clarificación del significado.
- Advertir de determinadas consecuencias.
- Determinar partes faltantes del código: TODO. Pero hay que tener en cuenta el primer principio de la introducción de este capítulo: Despues es igual que nunca.
- Amplificar la importancia de algo que resultaría irrelevante en caso contrario.

Formato del código

Cómo escribir el código es tan importante que el resto de apartados vistos hasta ahora. En caso de programación en Python se tendrá muy en cuenta la Pep 8, pero aun así se podrán seguir unas reglas:

- Archivos de longitud entre 200 y 500 líneas.
- En la parte superior se situarán elementos generales del módulo y según se va bajando se irán entrando en conceptos con más detalle. Es decir, las funciones generales en la parte superior y las más concretas en la inferior, lo que conlleva a escribir primero las funciones que usan otras funciones.
- Las sentencias relacionadas estarán generalmente juntas, separándolas de otras mediante espacios.
- Los conceptos relacionados estarán cerca unos de otros, a mayor distancia entre dos funciones, menor relación entre ambas.
- En contra de la Pep 8 y al ser esto un curso de programación: las variables definirán siempre en la parte superior de la misma.
- Se utilizará sangrado de forma efectiva, en caso de Python está implementado en el propio lenguaje.

Objetos y estructuras de datos

El nombrado de clases sigue las normas ya explicadas, debe ser un nombre que describa su responsabilidad y seguir una nomenclatura (Pep 8: ClaseNombre). Si no podemos derivar un nombre descriptivo, es muy probable que dicha clase incumpla en principio SRP y haya que dividirla. Además, incluiremos una breve descripción de la clase justo después de la definición (Docstring en Python) de no más de 25 palabras que no incluya las palabras: si, o, y, pero. Ya que son indicadores de que la clase incumple el principio SRP.

Recordamos el **principio SRP**: Una clase o módulo debe tener un solo motivo para cambiar, es decir, debe implementar un único concepto. Este principio es uno de los conceptos más importantes de la POO.

Para crear las clases, definiremos el menor número posible de variables de instancia y a la hora de diseñar nuestras clases hay una razón para que las variables sean privadas, no queremos que nadie dependa de ellas, queremos poder cambiar su tipo en el futuro si nos hace falta o modificar los algoritmos de acceso y control a las mismas si fuera necesario. Tampoco es necesario crear un método getter/setter por cada una de ellas, es mejor crear un API de acceso que permanezca invariable en el tiempo y sea abstracta.

Hay que diferenciar bien lo que son clases de los que son estructuras de datos o tipos, no son lo mismo, ni rigen las mismas normas, ni se utilizan para los mismos casos. No todo en el sistema será un objeto.

Por norma general, a la hora de programar un método tendremos en cuenta la ley de Demeter que dice que un método de una clase solo debe invocar métodos de:

- La propia clase.

- Cualquier objeto creado por el método.
- Cualquier objeto pasado como parámetro al método.
- Cualquier método de un objeto almacenado en una variable de instancia o clase.

Las clases deben tener cohesión, es decir los métodos de la clase manipularán cuantas más variables de instancia mejor. Pero es no indica que todos los métodos tengan que usar todas las propiedades ni que nos volvamos locos haciendo métodos que usen el mayor número posible de propiedades. Debemos tener un equilibrio entre la descripción que proporciona de API que estamos generando y las propiedades existentes.

Principio Abierto – Cerrado: Las clases deben ser cerradas para su propia modificación, pero abiertas para la ampliación. Es decir, la clase se modificará lo menos posible en nuestro programa y en caso de necesidad se podrá heredar de ella de forma fácil para crear nuevas responsabilidades.

Con el principio anterior, crearemos clases abstractas que encapsulan los conceptos y clases cliente que implementen dichos conceptos, una clase cliente que dependa de detalles concretos estará en peligro si dichos detalles cambian.

Control de errores

El control de errores es importante, pero si oscurece la lectura del código es incorrecto. Tradicionalmente se crearon códigos de error o devolvían **null** en las funciones que se debían tratar en bloques if-else. Esta estructura hoy en día está superada por las Excepciones. En caso de error una función generará una excepción del sistema o propia que tendrá que ser capturada y tratada dentro de un bloque try-catch superior.

Para crear estructuras try-catch-finally eficientes hay que tener en cuenta que, al salir del bloque, nuestro programa deberá estar en un estado coherente para su funcionamiento, es decir, dentro se debe dar solución al problema (no significa arreglar el problema, significar dejar el programa en un estado que pueda continuar ejecutándose).

El tratamiento de errores es uno de los puntos más difíciles de programar bien, por lo que atenderemos con sumo cuidado a la programación de estas situaciones. En concreto, las excepciones que se generen deben dar contexto al error para determinar el origen y la causa del mismo, los mensajes que generen tendrían que ser descriptivos y en un lenguaje fácilmente entendible.

Creación de pruebas

La creación de pruebas a evolucionado a la par que las metodologías y las formas de programar desde sus inicios, por lo que hay que prestar atención a las pruebas. Las pruebas no van a asegurar que no existen errores en el código, van a probar que cumple con las expectativas, por lo que tener pruebas incorrectas es igual que no tenerlas, o incluso peor al dar una falsa sensación de seguridad. Si tenemos un conjunto adecuado de pruebas, no tendremos miedo a cambiar cualquier parte del software ya que la realización de las pruebas nos asegurará que dicho cambio no afecta a otra parte de nuestro software.

Otro punto a tener en cuenta es la evolución de las pruebas. Si el código cambiar, porqué no lo van a hacer las pruebas, estas deben cambiar con los ciclos de desarrollo, **el código de pruebas es tan importante como el de producción.**

Para crear un caso de prueba de forma correcta atenderemos al patrón: generar-operar-comprobar en cada uno de ellos, de tal manera que el caso sea cerrado al resto de casos, es decir sea independiente completamente. En el primer paso, generamos los datos de pruebas, a continuación, realizamos las operaciones necesarias sobre dichos datos para crear la prueba y por último comprobamos los resultados generados y los esperados de la prueba.

Las pruebas siguen muchas de las reglas del código limpio, en concreto deben atender al principio de simple responsabilidad, es decir, deben comprobar un único concepto por caso de prueba. También siguen las siguientes reglas:

- Deben ser rápidas.
- No deben depender de las demás pruebas.
- Deben poder repetirse en cualquier entorno.
- Deben tener un resultado pasada – no pasada.
- Deben crearse antes del código de producción.

Reglas del diseño sencillo

- Ejecutar todas las pruebas. Los sistemas que no pueden probarse no se pueden verificar y no se debería implementar. La creación de pruebas conduce a obtener mejores resultados.
- Refactorizar. Dejar el código limpio, hay que refactorizar progresivamente.
 - Eliminar duplicados.
 - El código debe ser expresivo. Debe contar con claridad la intención de su autor.
 - Reducir el tamaño de clases y métodos. Esta última regla es la de menor rango de las tres anteriores y estará supeditada a ellas.

ANEXO VII (Patrones)

Los patrones son mecanismos de programación para resolver problemas comunes estandarizando la forma de programarlos.

<https://refactoring.guru/design-patterns/creational-patterns>

Singleton

```
class Singleton:
    """
    Este patrón es para mantener un único objeto pero
    con varias referencias.
    """

    __instance = None
    __value = 0

    def __new__(cls, *args, **kwargs):
        if cls.__instance is None:
            cls.__instance = object.__new__(cls)
        return cls.__instance

    def __str__(self):
        return "Mi Objeto: " + str(self.__value)

    def setvalue(self, val: int):
        self.__value = val

    @classmethod
    def me(cls):
        return cls.__instance

miObjeto1 = Singleton()
print(miObjeto1)          # 0
miObjeto2 = Singleton()
miObjeto2.setvalue(5)
print(miObjeto1)          # 5
print(Singleton.me())     # 5
```

Fábrica

```
import abc

class Fabrica:
    """
        Creación de una clase abstracta padre para
        gestionar varias hijas.
        Las hijas se definen para qué tipo gestionan y la
        madre realiza la creación de forma adecuada
        Las variables que empiezan con __ se renombran en
        las clases hijas, por eso no se puede usar __valor
    """
    valor = None

    def __new__(cls, objeto):
        for clase in cls.__subclasses__():
            if clase.estaDiseñadaPara(objeto):
                o = object.__new__(clase)
                o.__init__(objeto)
                return o

    def __init__(self, objeto):
        self.valor = objeto

    @classmethod
    def estaDiseñadaPara(cls, objeto):
        if cls.tipo in str(type(objeto)):
            return True

    @abc.abstractmethod
    def __str__(self):
        pass

class GestionStr(Fabrica):
    tipo = "str"

    def __str__(self):
        return "Cadena: " + str(self.valor)

class GestionInt(Fabrica):
    tipo = "int"

    def __str__(self):
        return "Entero: " + str(self.valor)

miObjeto1 = Fabrica("Hola")
print(miObjeto1)                      # Hola
miObjeto1 = Fabrica(23)
print(miObjeto1)                      # 23
miObjeto1 = Fabrica(True)
print(miObjeto1)                       # None
```

Director

```
import abc

class FiguraGeometrica:
    """
    Sería mucho mejor usar propiedades y setter
    """
    numero_lados = 0
    longitud = 0
    radio = 0

    def __str__(self):
        return "Numero de lados: %s, la longitud %s u el radio: %s" % (self.numero_lados, self.longitud, self.radio)

class Constructor:
    def __init__(self):
        self._producto = None

    @property
    def producto(self):
        return self._producto

    @producto.setter
    def producto(self, prod: FiguraGeometrica):
        self._producto = prod

    def crearProducto(self):
        self._producto = FiguraGeometrica()

    @abc.abstractmethod
    def configurarNumeroLados(self):
        pass

    @abc.abstractmethod
    def configurarLongitud(self):
        pass

    @abc.abstractmethod
    def configurarRadio(self):
        pass

class Cubo(Constructor):
    def configurarNumeroLados(self):
        self.producto.numero_lados = 4

    def configurarLongitud(self):
        self.producto.longitud = 20

    def configurarRadio(self):
        pass

class Triangulo(Constructor):
```

```
def configurarNumeroLados(self):
    self.producto.numero_lados = 3

def configurarLongitud(self):
    self.producto.longitud = 15

def configurarRadio(self):
    pass

class Circunferencia(Constructor):
    def configurarNumeroLados(self):
        pass

    def configurarLongitud(self):
        pass

    def configurarRadio(self):
        self.producto.radio = 10

class Director:
    """
    Cuando una clase es muy compleja de inicializar, en vez de proporcionar métodos diferentes para cada tipo de inicialización, se crean constructores diferentes para cada tipo
    """
    def __init__(self):
        self._constructor = None
    @property
    def constructor(self):
        return self._constructor
    @constructor.setter
    def constructor(self, construct: Constructor):
        self._constructor = construct
    def crearProducto(self):
        self.constructor.crearProducto()
        self.constructor.configurarLongitud()
        self.constructor.configurarNumeroLados()
        self.constructor.configurarRadio()
        return self.constructor.producto

direcc = Director()
direcc.constructor = Cubo()
producto = direcc.crearProducto()
print(producto)
direcc.constructor = Triangulo()
producto = direcc.crearProducto()
print(producto)
direcc.constructor = Circunferencia()
producto = direcc.crearProducto()
print(producto)
```

Prototipo

```
from copy import deepcopy

class Prototipo:
    """
        Cuando una instanciación es compleja es mejor
        hacer una copia de una ya existente
    """
    _instancia_primer = None

    def __new__(cls):
        result = object.__new__(cls)
        if cls._instancia_primer is not None:
            result.__dict__ =
                deepcopy(cls._instancia_primer.__dict__)
        else:
            cls._instancia_primer = result
        return result

    def __init__(self):
        if self._instancia_primer is not None:
            self._instancia_primer = None
            self.x = 0
            self.y = 0

    def __str__(self):
        return "(%s,%s)" % (self.x, self.y)

#miObjeto1 será el referente para todos los demás
miObjeto1 = Prototipo()
miObjeto1.x = 1
print(miObjeto1)          # (1,0)
miObjeto2 = Prototipo()
miObjeto2.y = 3
print(miObjeto2)          # (1,3)
miObjeto1.x = 2           # cambio el referente
miObjeto3 = Prototipo()
print(miObjeto3)          # (2,0)
```

Adaptador

```
import abc
"""

El uso de objetos no generados por nosotros, los tenemos que
adaptar a nuestro uso mediante
una clase que hace dicha adaptación
"""

# clase generadas por otros
class Perro:
    def ladrar(self):
        return "Ladrando..."

class Gato:
    def maullar(self):
        return "Maullando..."

"""

Queremos hacer uso de las clases anteriores en nuestra aplicación
pero de una forma similar ambas
"""

class Animal(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def hacer_ruido(self):
        pass

class AdaptadorPerro(Animal, Perro):
    def hacer_ruido(self):
        return self.ladrar()

class AdaptadorGato(Animal, Gato):
    def hacer_ruido(self):
        return self.maullar()

for inx in [AdaptadorPerro(), AdaptadorGato()]:
    print(inx.hacer_ruido())

"""

La solución anterior funciona, pero no tiene el espíritu del
patrón en el que debe guardarse la referencia al objeto original
"""

class AdaptadorPerro(Animal):
    def __init__(self, objeto: Perro):
        self._obj = objeto

    def hacer_ruido(self):
        return self._obj.ladrar()
```

```
def __getattr__(self, item):
    return self._obj.__getattribute__(item)

class AdaptadorGato(Animal):
    def __init__(self, objeto: Gato):
        self._obj = objeto

    def hacer_ruido(self):
        return self.maullar()

    def __getattr__(self, item):
        return self._obj.__getattribute__(item)

for inx in [AdaptadorPerro(Perro()), AdaptadorGato(Gato())]:
    print(inx.hacer_ruido())

"""
Mejor todavía si usamos una factoría
"""

class Factoria:
    @classmethod
    def getAnimal(cls, obj):
        if isinstance(obj, Perro):
            return AdaptadorPerro(obj)
        if isinstance(obj, Gato):
            return AdaptadorGato(obj)
        return None

for inx in [Perro(), Perro(), Gato()]:
    print(Factoria.getAnimal(inx).hacer_ruido())
```

Cadena

```
"""
Patrón cadena    c1(m1) -> c2 -> c3
"""

import abc

class Interfaz:
    __metaclass__ = abc.ABCMeta

    _next = None

    def next(self, n):
        if isinstance(n, Interfaz):
            self._next = n

    @abc.abstractmethod
    def run(self, msg): pass

class MiClase1(Interfaz):
    def run(self, msg):
        print("1.-" + msg)
        if self._next is not None:
            self._next.run(msg)

class MiClase2(Interfaz):
    def run(self, msg):
        print("2.-" + msg)
        if self._next is not None:
            self._next.run(msg)

c1 = MiClase1()
c2 = MiClase2()
c3 = MiClase2()
c1.next(c2)
c2.next(c3)
c1.run("Hola")
```

Estado

```
"""
Guardar un estado anterior al él y revertirlo si fuera necesario
"""

class ClaseGuardaEstado:
    estado = None
    estado_anterior = None

    def __init__(self, estado: int):
        self.estado = estado
        self.estado_anterior = estado

    def setEstado(self, estado: int):
        self.estado_anterior, self.estado = self.estado, estado

    def getEstado(self):
        return self.estado

    def revertirEstado(self):
        self.estado = self.estado_anterior

    def __str__(self):
        return "El estado actual es: %s, y el anterior: %s" % (self.estado, self.estado_anterior)

c = ClaseGuardaEstado(5)
print(c)
c.setEstado(6)
print(c)
c.setEstado(7)
print(c)
c.revertirEstado()
print(c)
```

Observador / Observado

```
"""
Un objeto es observado por varios, y son notificados cuando
cambia
"""
import abc

class Observador(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def run(self, msg):
        pass

    @abc.abstractmethod
    def update(self, msg):
        pass

class Observado:
    _lista_observadores = []
    _valor = None

    def __init__(self, valor: int):
        self._valor = valor

    def setObservador(self, obs: Observador):
        self._lista_observadores.append(obs)

    def removeObservador(self, obs: Observador):
        self._lista_observadores.remove(obs)

    def notify(self):
        for ele in self._lista_observadores:
            ele.update(self._valor)

    def setValor(self, valor: int):
        self._valor = valor
        self.notify()

class Punto(Observador):
    def __init__(self, msg):
        self.msg = msg

    def run(self, msg):
        print("Punto %s: %s" % (self.msg, msg))

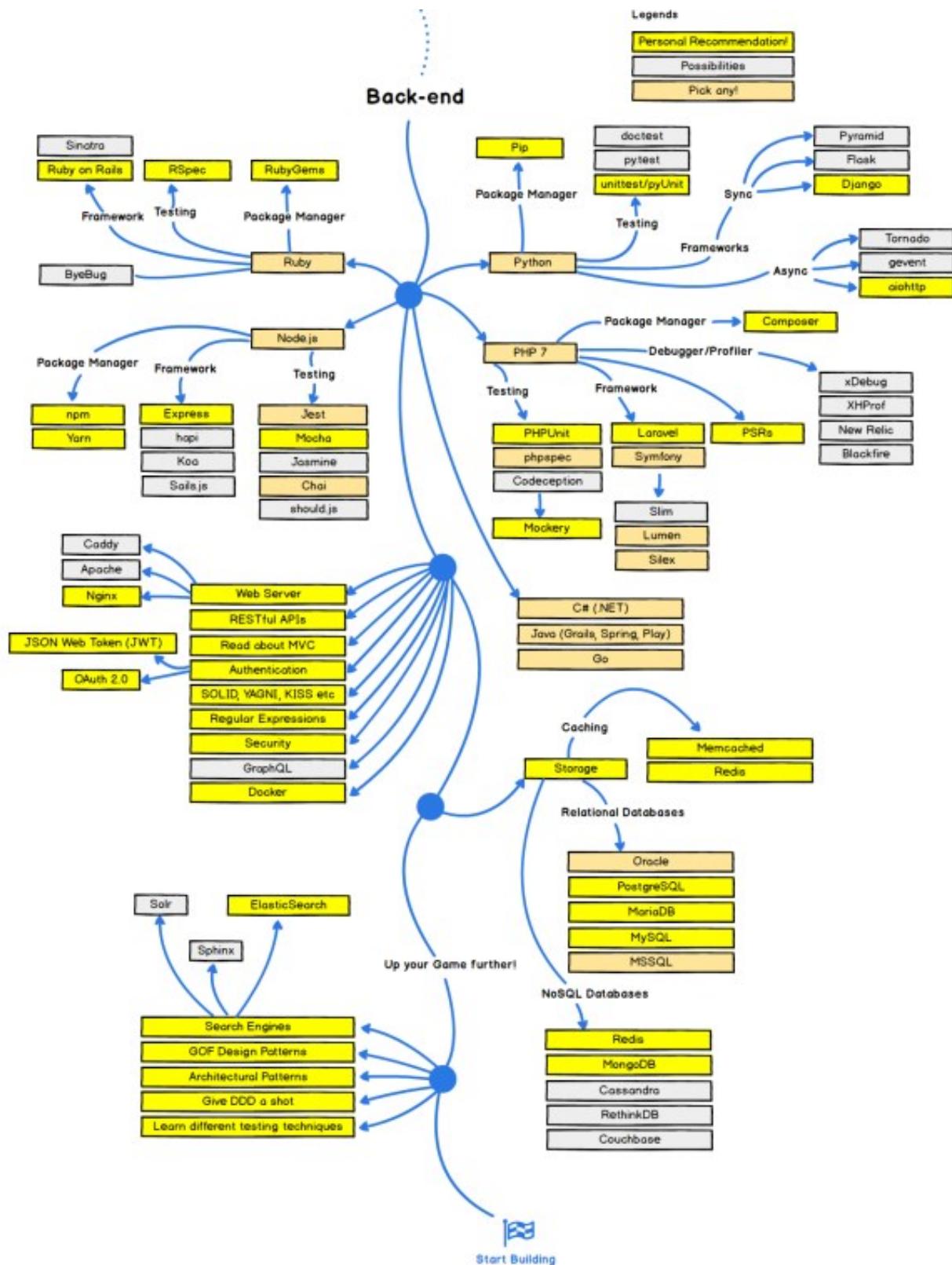
    def update(self, msg):
        super().update(msg)
        self.run(msg)

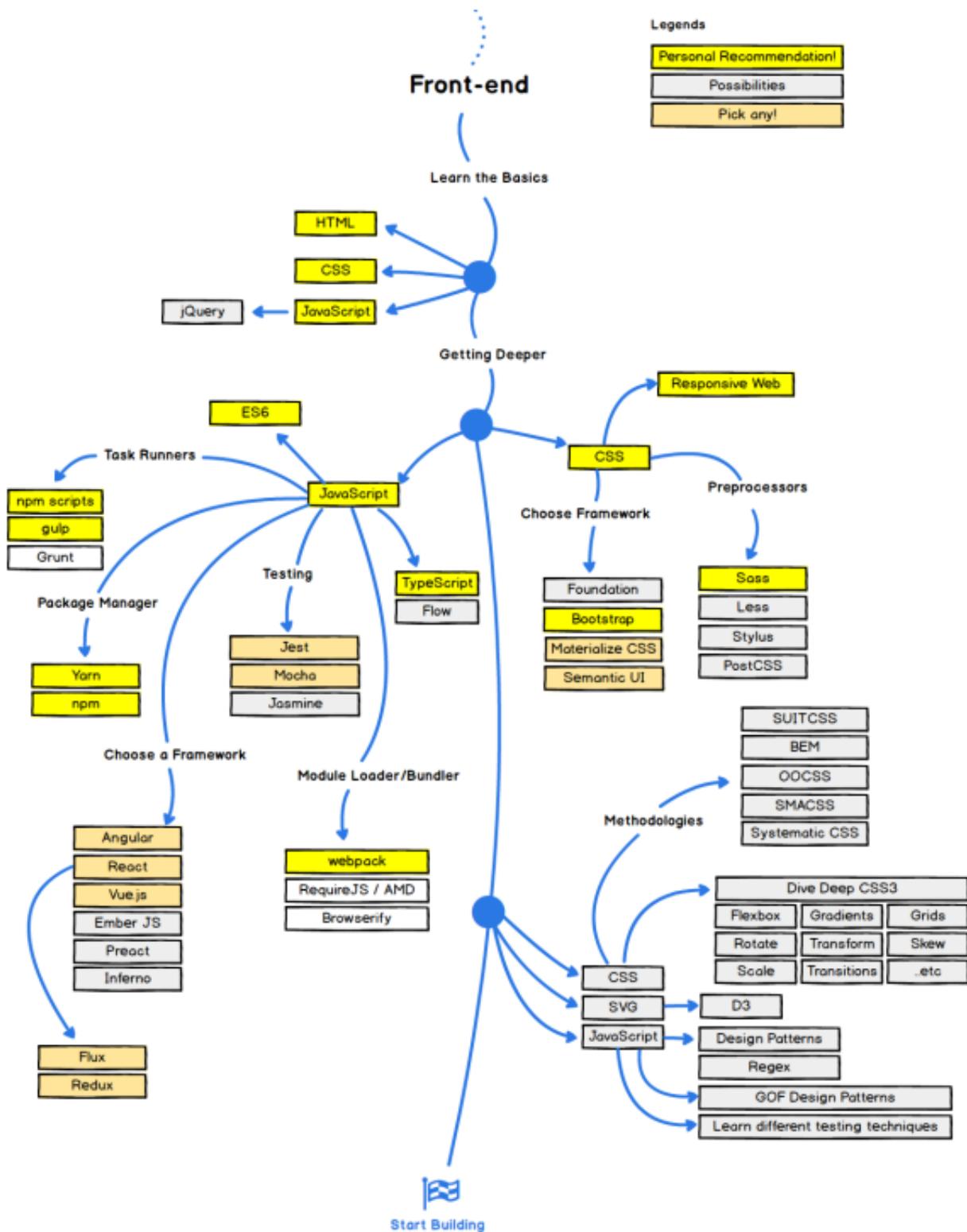
obs = Observado(4)
obs1 = Punto("Uno")
obs2 = Punto("Dos")
```

```
obs.setObservador(obs1)
obs.setObservador(obs2)
obs.setValor(5)
obs.removeObservador(obs1)
obs.setValor(6)
```

ANEXO VIII (Tecnologías Web)

Visitar: <https://roadmap.sh/>





ANEXO IX (Índice completo)

Índice completo

U. T. 1 Introducción	5
Origen de la Informática.....	6
Conceptos Informáticos Fundamentales.....	6
Clasificación del Software.....	6
Qué es la programación.....	7
Concepto.....	7
Paradigmas de la programación	7
Qué es un algoritmo	8
Representación de los algoritmos	8
Elementos de los algoritmos	8
Diagramas de flujo (ordinogramas).....	9
Símbolos de los diagramas de flujo	9
Pseudocódigo	12
Sintaxis del pseudocódigo	13
Rendimiento de los algoritmos.....	14
Órdenes de complejidad	14
Codificación y ejecución	15
Clasificación de los lenguajes	15
Documentación de los programas.....	16
Ciclo de Vida del Software.....	17
Ejercicios.....	18
Entregar	20
Ampliación.....	21
U. T. 2 Elementos de un programa informático.	22
Introducción	23
Historia de Python	23
Versiones	23
Versión 1.0.....	23
Versión 2.0.....	24
Versión 3.0.....	24
Características de Python	24
Un vistazo rápido al interior de Python.....	25
Instalación de Python	26
Entornos IDE	26

Elementos de un programa	27
Estructura General de un programa.....	27
Conceptos básicos a recordar.....	27
Funciones introductorias.....	27
Ejemplo: Ejecutar un programa.....	28
Explicación del código	29
Líneas y espacios en blanco	30
Comentarios	31
Identificadores.....	31
Palabras reservadas.....	31
Operadores y expresiones.....	32
Operador de concatenación de cadenas de caracteres	32
Operadores lógicos o booleanos	32
Operadores de comparación	32
Consideraciones sobre los operadores de comparación.....	33
Operadores aritméticos en Python	33
Operadores a nivel de bits.....	34
Operadores de asignación.....	35
Operador morsa	35
Operadores de pertenencia.....	35
Operadores de identidad.....	36
Prioridad de los operadores en Python.....	36
Expresiones.....	36
Representación de los datos (Literales)	36
Literales	36
Literales numéricos	37
Literales de bytes.....	37
Literales Cadena	37
Tipos de datos.....	38
Booleanos	38
Nulo	39
Numéricos.....	39
Cadenas	39
Forzado de tipos	41
Variables y Constantes	42
Bibliotecas	42
Ejercicios.....	44
U. T. 3 Estructuras de control.....	46
Introducción	47

Condiciones	47
Estructuras de selección o alternativas.....	47
Estructuras de repetición	48
Introducción	48
Función range	48
Bucles.....	49
Estructuras de salto	49
Break y continue.....	50
Prueba y depuración de programas	50
Entrono de depuración.....	50
Ejemplo de depuración.....	52
Punto de ruptura	52
Inspección.....	52
Prácticas de depuración	52
Creación de casos de prueba.....	53
Documentación del código del programa	53
Introducción	53
Cadenas de documentación	53
Generando documentación en páginas de HTML.....	55
Docstrings vs. comentarios.....	55
Conclusiones.....	55
Cuestiones de Estilo.....	56
Programación modular	57
Ejemplo.....	57
Funciones en la programación modular	57
Diferencia entre procedimientos y funciones	57
Definición.....	58
Ámbito de las variables	58
Parámetros	58
Parámetros de Entrada – Salida	59
Juego del ahorcado.....	59
Primer nivel.....	59
Segundo nivel	59
Tercer nivel	59
Ámbito de las variables	60
Recomendaciones.....	61
Ejercicios.....	62
U. T. 4 P.O.O.	67
Introducción	68

Cómo se piensa en clases y objetos	68
Orígenes.....	68
Características de la POO	69
Ventajas y desventajas de la POO	70
Conceptos Fundamentales POO.....	70
Clases	71
Propiedades o atributos	71
Métodos	71
Objetos	72
Encapsulación y Ocultación	72
Abstracción	73
Herencia. Tipos de Herencia.....	74
Soluciones.....	76
Actividad N°1: Definir los siguientes términos:	76
Actividad N°2: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.	77
Actividad N°3: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.	77
Actividad N°4: Para la siguiente situación debes listar los objetos detectados, realizar la clasificación de clases de los objetos y determinar sus atributos.	78
Relaciones entre las clases	79
Sobrecarga	88
Polimorfismo y Enlace Dinámico	88
Mensajes.....	88
Lenguajes.....	89
Ejercicios.....	90
Ejercicio (Repaso)	91
Introducción	91
Descripción del proyecto	91
U. T. 5 POO (I).	92
Clases	93
Declaración de una clase	93
Nombres de la clase.....	94
Estructura y miembros de una clase	95
Creación y utilización de atributos.	95
Creación y utilización de métodos.....	95
Definición.....	96
Parámetros	98
Parámetros por defecto	99
Paso de parámetros por palabras clave	99

Número indeterminado de parámetros	100
Desempaquetado de parámetros.....	100
Sugerencias de tipos en los parámetros.....	100
Devolución de valores	100
Funciones lambda o anónimas	101
Paso de métodos a funciones.....	101
Creación y utilización de constructores y destructores	101
Propiedades.....	102
Ejemplos	103
Objetos	103
Creación y uso de objetos	104
Visibilidad	104
Ejercicio	106
Paquetes.....	107
Módulo	107
Paquete.....	107
Ejemplo.....	107
Uso de import.....	108
Destrucción de objetos y Liberación de memoria.....	109
La gestión del recolector de basuras	110
Librería estándar.....	110
Cadenas, clase str	110
Cadenas de caracteres.....	111
Expresiones regulares.....	114
Patrones.....	114
Metacaracteres.....	114
Elementos de una expresión	115
Ejemplos:	116
Módulo re	116
Math	118
Pandas	119
NumPy	119
Mathplotlib.....	119
Repaso	120
Ejercicios.....	124
Apartado A.....	124
Ejercicio A1 – Punto.....	124
Ejercicio A2 – Persona	124
Ejercicio A3 – Rectángulo	124

Ejercicio A4 – Artículo.....	124
Apartado B	125
Ejercicio B1 – Punto.....	125
Ejercicio B2 – Persona	125
Ejercicio B3 – Rectángulo	125
Ejercicio B4 – Artículo	125
Ejercicios de cadenas.....	125
Caso Práctico Dawbank	127
U. T. 6 POO (II).....	128
Tipos avanzados: Secuencias.....	129
Acceso a las secuencias	130
Tipos avanzados: Listas.....	130
Operaciones.....	130
Borrado de elementos.....	131
Ordenación	131
Pasar Listas como Parámetros.....	132
Listas multidimensionales	132
Otros tipos avanzados	133
Tuplas.....	133
Diccionarios	133
Operaciones.....	134
Conjuntos.....	134
Comprensión de listas y diccionarios	135
Recursividad	136
Introducción	136
Hacer un buen uso de la recursividad	137
Ventajas y Desventajas	137
Ventajas	137
Desventajas.....	137
Ejemplos	137
Herencia.....	138
Introducción	138
Sintaxis.....	138
Ejemplo	141
Sobreescritura de métodos heredados.....	141
Sobrecarga de Métodos	142
Métodos especiales o mágicos.....	142
Métodos géttters and setters	144
Clase object	145

Polimorfismo	145
Determinación del tipo de un objeto	146
Atributos de clase	146
Métodos de clase.....	147
Clases y métodos abstractos	147
Clases y métodos estáticos.....	148
Clases y métodos finales.....	148
Ejemplos	148
Interfaces.....	150
Decoradores	151
Generadores	152
Excepciones	153
Definición.....	153
Tipos	153
Jerarquías.....	153
Captura	155
Generación de Excepciones.....	156
Definición de Excepciones propias	157
Estructuras de datos.....	158
Qué es una lista	158
Usando listas como pilas	159
Usando listas como colas.....	159
Árboles.....	160
Grafos.	161
Creación de casos de prueba.....	163
Cobertura de las pruebas	165
Ejercicios.....	166
Apartado A.....	166
Ejercicio A1 – Punto.....	166
Ejercicio A2 – Persona	166
Ejercicio A3 – Rectángulo	166
Ejercicio A4 – Articulo.....	166
Apartado B - ASTROS	167
Apartado C - MASCOTAS	167
Apartado D – BANCO	168
Apartado E – EMPRESA Y EMPLEADOS.....	169
Apartado F - VEHÍCULOS.....	170
Apartado G - FIGURAS	170
Apartado H – Ejercicios de listas.....	171

Apartado I – Ejercicios de diccionarios.....	172
Ejercicio 1.....	172
Ejercicio 2.....	172
Ejercicio 3.....	172
Ejercicio 4.....	172
Ejercicio 5.....	173
Ejercicio 6.....	173
Ejercicio 7.....	173
Ejercicio 8.....	173
Ejercicio 9.....	173
Ejercicio 10.....	173
Ejercicio 11.....	174
Ejercicio 12.....	174
Apartado J – exercism.io	174
U. T. 7 Proyecto.	175
Objetivo	176
Descripción del videojuego	176
Descripción de las etapas	176
Inicio	176
Nivel 1	176
Nivel 2	177
Nivel 3	177
Nivel 4	177
Nivel 5	177
Ganar	178
Perder	178
Fin	178
Aleatoriedad de las preguntas.....	178
U. T. 8 Gestión de datos.	179
Gestión de ficheros.....	180
Qué es un fichero	180
Tipos de acceso.....	180
Ficheros binarios y ficheros de texto.....	180
Modo de acceso a los ficheros	180
Procesamiento de un fichero	180
Qué son los flujos	181
Acceso a ficheros de Texto y Binarios	181
Apertura de ficheros.....	181
Modos de acceso.....	181

Cierre de ficheros	182
Uso de with con ficheros	183
Escritura y lectura de información en ficheros.....	183
Ejemplo de acceso	183
Resumen.....	184
Serialización	184
Utilización de los sistemas de ficheros	185
Ejemplo.....	185
Ejemplo	185
Bases de Datos.....	186
Clases para el acceso a BD relacionales.....	186
Establecimiento de conexiones con diferentes SGBD.....	186
Instalación del driver Python.....	187
Conexión con la BBDD	187
Gestión de errores de conexión	187
Inserción de datos	187
Borrado de registros	188
Sentencias de selección.....	188
Ejemplos de consultas sobre la base de datos	189
XML.....	189
Librerías Relacionadas con XML	190
SAX.....	190
DOM.....	190
Creación de un documento XML	190
SAX.....	191
DOM.....	191
Ejercicios.....	193
1º Ejercicio	193
2º Ejercicio	193
3º Ejercicio	193
4º Ejercicio	193
5º Ejercicio	193
Ejercicio - Máximo y mínimo	193
Ejercicio - Notas de alumnos	193
Ejercicio - Ordenando archivos.....	193
Ejercicio - Nombre y apellidos	193
Ejercicio - Búsqueda en PI.....	193
Ejercicio - Estadísticas.....	194
6º Ejercicio	194

7º Ejercicio	194
8º Ejercicios de ampliación varios	194
Uno	194
Dos	194
Tres	194
Cuatro	194
Cinco	194
Seis	194
U. T. 9 Interfaces gráficos	195
Introducción al diseño GUI	196
Qué es la interfaz de usuario	196
Qué es el diseño GUI	197
Diseño dirigido a eventos	198
Creación de un GUI	198
Componentes	199
Librerías	199
Qt	199
Ventajas de usar PyQt - PySide	199
Desventajas de usar Qt	200
Tkinter	200
Ventajas de usar Tkinter	200
Desventajas de usar Tkinter	200
¿Cuál elegir?	200
Programación GUI Básica	200
Hola mundo	200
PySide	200
PyQt	202
Primeros pasos con el GUI	202
Eventos QT	205
Elementos de la ventana principal	205
Eventos y mensajes	207
Diálogos	209
Controles	210
Controles avanzados	211
Layouts	212
Absoluto	212
Box Layout	212
Grid layout	213
Ejemplo	214

Para finalizar	215
Ejercicios	216
Ejercicio 1 – ¿Par o impar?	216
Ejercicio 2 – Mini calculadora I	216
Ejercicio 3 – Mini calculadora II	216
Ejercicio 4 – Validar letra NIF.....	216
Ejercicio 5– Número aleatorio.....	216
Ejercicio 6 – Dados de Rol.....	216
Ejercicio 7 – Tablas de multiplicar	216
Ejercicio 8 – Inicio de sesión	216
ANEXO I (Ahora qué)	218
ANEXO II (Bibliografía).....	219
ANEXO III (Nuevas versiones)	220
Python 3.10.....	220
Patrón estructural.....	220
Caso implementado en otros lenguajes	220
Python 3.11.....	222
Ubicaciones de error mejoradas	222
Nueva anotación para clases: Self	222
Velocidad mejorada.....	222
Excepciones a coste cero.....	222
Python 3.12.....	222
Expresiones f''' mejoradas	222
Decorador @override.....	223
Mejores mensajes de error	224
PEP 684: A Per-Interpreter GIL	224
ANEXO IV (F.A.Qs).....	225
POO avanzado	225
Propiedades de clase	225
¿Cómo invoco a un método definido en una clase base desde una clase derivada que lo ha sobrescrito?	226
Polimorfismo	226
Iteradores	227
Implementación recomendada	227
Varios	228
¿Por qué los valores por defecto se comparten entre objetos?	228
¿Cuál es la forma más eficiente de concatenar muchas cadenas conjuntamente?	229
¿Cómo puedo crear una lista multidimensional?.....	229
¿Por qué list.sort() no devuelve la lista ordenada?	230

Utilizar el color ASCII	230
Cambio de librerías recomendados.....	231
Pathlib en vez de os	231
Dataclasses en vez de namedtuple	232
ANEXO V (Diseño POO Robusto)	233
Qué es el software robusto	233
Creación del software en tres pasos	233
Pequeños proyectos	233
Creación de requerimientos y casos de uso	233
Principios de la POO	233
Principios OPP revisados	234
Principios de análisis y diseño	235
Grandes proyectos.....	235
Divide y vencerás.....	235
Creación de características.....	235
Diagramas de casos de uso.....	235
Análisis del dominio.....	235
Uso del patrón MVC	236
Resumen	236
Arquitectura.....	236
Relaciones entre los objetos.....	236
Repetir para terminar	236
Tipos de programación.....	237
Ejemplo (Cap. 10 Head First Object-Oriented Analysis&Design – Brett D. McLaughlin)	238
Supuesto	238
Paso 1: Lista de características	239
Paso 2: Diagrama de casos de Uso	239
Paso 3: Divide y vencerás, haciendo partes más pequeñas.	240
Paso 4: Iterar para encontrar los requerimientos	240
Paso 4.1: Cargar red de metro.....	240
Paso 4.1.1: Conocer el problema.....	240
Paso 4.1.2: Requerimientos: casos de uso	241
Paso 4.1.3: Análisis del dominio	241
Paso 4.1.4: Diseño Preliminar.....	242
Paso 4.1.5: Implementación	242
Paso 4.2: Hacer ruta	242
Paso 4.2.1: Conocer el problema.....	242
Paso 4.2.2: Requerimientos: casos de uso	242
Paso 4.2.3: Análisis del dominio	242

Paso 4.2.4: Diseño Preliminar.....	243
Paso 4.2.5: Implementación	243
Paso 4.3: Implementar Test.....	243
ANEXO VI (Escribir código limpio)	244
Por qué de la necesidad de escribir código limpio	244
Nombrado.....	244
Diseño de funciones	245
Comentarios	246
Formato del código.....	246
Objetos y estructuras de datos.....	246
Control de errores	247
Creación de pruebas.....	247
Reglas del diseño sencillo	248
ANEXO VII (Patrones).....	249
Singleton.....	249
Fábrica	250
Director.....	251
Prototipo.....	253
Adaptador.....	254
Cadena.....	256
Estado	257
Observador / Observado	258
ANEXO VIII (Tecnologías Web)	260
ANEXO IX (Índice completo)	262
ANEXO X (Licencia)	275

ANEXO X (Licencia)

Estos apuntes se distribuyen según la licencia: **Atribución-NoComercial 4.0 Internacional (CC BY-NC 4.0)**



Attribution-NonCommercial 4.0 International

<https://creativecommons.org/licenses/by-nc/4.0/legalcode>

Usted es libre de:

- ∞ Compartir — copiar y redistribuir el material en cualquier medio o formato.
- ∞ Adaptar — remezclar, transformar y construir a partir del material.

El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

- ∞ Atribución — Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo del licenciante.
- ∞ No Comercial — Usted no puede hacer uso del material con propósitos comerciales.
- ∞ No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia.

Avisos:

No tiene que cumplir con la licencia para elementos del material en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable.

No se dan garantías. La licencia podría no darle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.



Programación

1º CFGS DAW

I.E.S. Arcipreste de Hita

César San Juan Pastor

