

Informática I (2024-2025)

Grado en Ingeniería de Sistemas Audiovisuales y Multimedia (URJC)

Jesús M. González Barahona,
David Cortés Polo, Ángel Madridano Carrasco

GSyC, EIF, Universidad Rey Juan Carlos
<http://cursoprogram.gitlab.io>

18 de septiembre de 2024



- 1 Presentación
- 2 Motivación y fundamentos

3 Tema 1: Introducción

- Conceptos básicos
- Estructura y funcionamiento básicos de un equipo
- Lenguaje de alto nivel
- Trabajando con los datos
- Primeros programas en Python

4 Tema 2: Estructuras de control

- Introducción
- Estructuras básicas de control
- Bucles
- Herramientas útiles en los bucles
- Excepciones
- Ejercicios

5 Tema 3: Divide y vencerás

- Conceptos Básicos

- Funciones y Procedimientos
- Parámetros de las funciones y su ámbito
- Ejercicios

6 Tema 4: Estructuras de Datos

- Conceptos Básicos
- TADs y Estructuras de datos
- Tuplas
- Ejercicios Tuplas
- Listas
- Funciones avanzadas de los TADs listas y tuplas
- Tuplas vs Listas
- Ejercicios Listas

7 Tema 5: Estructuras de Datos II

- Conceptos Básicos
- Diccionarios
- Ejercicios Diccionarios
- Manejo de cadenas de caracteres

- Ejercicios de Strings
- Punteros y gestión de memoria
- Combinación de estructuras
- Ejercicios de combinación de estructuras

8 Tema 6: Gestión de Ficheros

- Manejo de ficheros
- Ejercicios de ficheros

9 Tema 7: Programación orientada a objetos

- Introducción
- La POO como metodología de programación
- Ejercicios

10 Algoritmos I: Algoritmos de Aproximación

- Introducción
- Teoría de Optimización

11 Tema 8: Eficiencia del código

- Introducción
- Tiempo de ejecución

- Complejidad algorítmica
- Ejercicios

Presentación

Objetivos

Aprender a programar:

“introducir los conceptos y técnicas básicas de programación de ordenadores”

Realizar programas:

“adquirir los conocimientos y habilidades básicos para realizar programas sencillos”

Objetivos (2)

Aproximación práctica:

“carácter fundamentalmente práctico, realizando programas sencillos desde las primeras clases, y proyectos completos en los que se deberán realizar programas que cumplan unas especificaciones dadas”

Herramientas y técnicas de apoyo:

“se introducirán herramientas de apoyo al desarrollo, técnicas de pruebas (testing) y depuración (debugging) y otras prácticas habituales en el desarrollo de software”

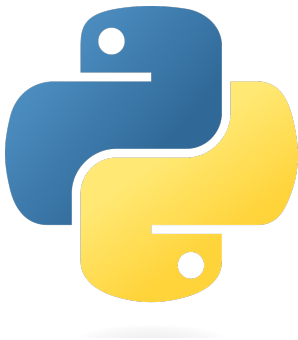
Profesores

- Jesús M. González Barahona, Gregorio Robles, David Moreno Lumbreras, Sergio Montes
- Grupo de Sistemas y Comunicaciones, EIF
- Despacho 110, 1ª planta, Departamental III
- Despacho 113, 1ª planta, Departamental III
- Despacho 103, 1ª planta, Biblioteca
- Horario de tutoría:
 - Jueves de 17:00 a 19:00 en la plataforma Teams
- Mejor forma de contactar con nosotros
 - Foro General en el aula virtual.
- Segunda mejor forma:
 - [jesus.gonzalez.barahona @ urjc.es](mailto:jesus.gonzalez.barahona@urjc.es)
 - [gregorio.robles @ urjc.es](mailto:gregorio.robles@urjc.es)
 - [david.morenolu @ urjc.es](mailto:david.morenolu@urjc.es)
 - [sergio.montes @ urjc.es](mailto:sergio.montes@urjc.es)

Recursos principales

- Aula virtual:
<https://aulavirtual.urjc.es>
- Grupo en GitLab de la EIF:
<https://gitlab.eif.urjc.es/cursoprogram>

Lenguaje de Programación: Python



Primer Mandamiento:
Amarás Python
por encima de (casi) todo.

<https://python.org>

<https://www.youtube.com/watch?v=UNSoPa-XQN0>

<https://madnight.github.io/github>

<https://spectrum.ieee.org/top-programming-languages-2022>

Entorno de Desarrollo: GitLab



Segundo Mandamiento:
GitLab será tu (humilde) morada.

<https://gitlab.eif.urjc.es>

Calendario

- Generalmente:
 - Martes de 9:00 a 11:00: “Teoría”
 - Jueves de 11:00 a 13:00: “Prácticas”
- Habrá excepciones (ver calendario)
- Labos en Laboratorios III:
 - martes: labo 210
 - jueves: labo 203

Calendario detallado (se irá actualizando):

<https://gitlab.eif.urjc.es/cursoprogram/materiales/-/blob/main/programa.md>

Evaluación

Prueba final de conceptos de programación (prueba escrita):

- Hasta 4 puntos, 5/10 para aprobar

Ejercicios prácticos para entrega:

- Hasta 2 puntos

Proyecto final:

- Hasta 2 puntos (requisitos obligatorios, necesario para aprobar)
- Hasta 2 puntos (parte opcional básica)
- Hasta 1 punto (parte opcional especial)

Evaluación (2)

Ejercicios prácticos y proyecto final: realización y entrega individual.

Evaluación de prácticas:

“La evaluación práctica será fundamentalmente continua, y tendrá en cuenta no solo las prácticas entregadas, sino también su desarrollo, su defensa (en su caso), y en general la evolución del alumno.”

Proyecto final:

- trabajo en grupo para resolver aspectos generales.
- defensa individual (todos los que sean llamados).

“El alumno deberá ser capaz de explicar satisfactoriamente cualquier aspecto del programa que entregue”

Evaluación (3)

Mínimo para aprobar:

- aprobado en prueba final de conceptos (2 puntos) y
- requisitos mínimos de proyecto final (2 puntos) y
- mínimo de 5 puntos en total

Convocatoria de junio (extraordinaria):

- Prueba de conceptos: sólo si no se aprobó
- Proyecto final: sólo si no se aprobó
- Ejercicios prácticos: nuevas fechas de entrega para los no aprobados

Calificación: igual que en ordinaria, pero:

- Calificación aprobada en ordinaria se mantiene
- Ejercicios prácticos entregados en junio: se tendrá en cuenta que no se entregaron en su momento

No se “guarda” nota de un curso para otro.

Estimación de dedicación

- Total: 6 créditos ECTS
- Cada ECTS: unas 25 horas de dedicación.
- Total: **150 horas**
- Sesiones en horario: unas 26
+ **52 horas**
- Fuera de horario:
 - Trabajo continuo (incl. ejercicios prácticos): 4 h/semana
+ **52 horas**
 - Proyecto final:
+ **30 horas**
 - Preparación prueba conceptos:
+ **8 horas**
 - Prueba de concepto:
+ **2 horas**
- Suma: **144 horas**

Motivación y fundamentos

Motivación



¿Cuáles han sido los motivos para elegir ISAM?

Motivación



¿Qué es una ingeniería?

Motivación

Definición de ingeniería:

- **RAE:** Conjunto de conocimientos orientados a la invención y utilización de técnicas para el aprovechamiento de los recursos naturales o para la actividad industrial
- **Significados.com:** El objetivo de la ingeniería es ofrecer soluciones a los problemas prácticos de las personas, tanto a nivel social como económico e industrial. De allí que la ingeniería sea una disciplina que transforme el conocimiento en algo práctico para beneficio de la humanidad.

Ejemplos

Veamos algunos ejemplos

Motivación



El desarrollo de los servicios de streaming

Motivación



**Desarrollo de nuevos códecs.
Nuevos requisitos y dispositivos**

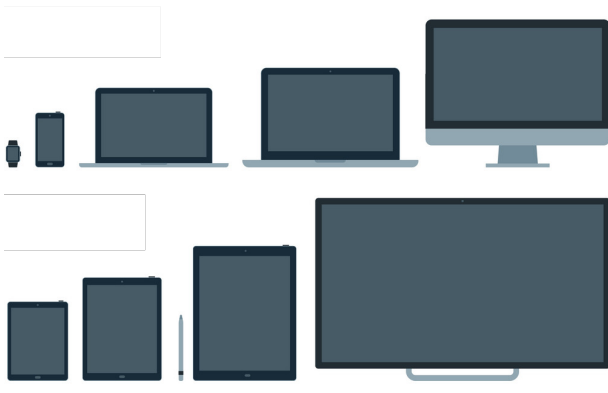
Motivación



Desarrollos disruptivos como el Metaverso

¿Qué elementos tienen en común estas innovaciones?

Fundamentos principales de Informática I



Todos los dispositivos usan software nativamente o han requerido software para reproducir un contenido

Fundamentos principales de Informática I



**Preguntas: ¿Has programado alguna vez?
¿En qué lenguaje?**

Fundamentos principales de Informática I

Qué habilidades vais a necesitar en el desempeño de vuestra profesión:

- Conocer las Tecnologías de la Información y las Comunicaciones (TIC)
- Aplicar fundamentos de Ingeniería para el desarrollo de soluciones
- Desarrollos innovadores usando una o varias tecnologías
- Conocimiento de múltiples tecnologías (hardware y software)

Fundamentos principales de Informática I

- Vamos a estudiar la base para el desarrollo de aplicaciones en entornos TIC
- Comenzar a pensar como Ingenieros a la hora de resolver problemas
- Aprenderemos cómo utilizar y reutilizar los recursos disponibles para nuestros propios desarrollos

Pero... ¿De dónde partimos?



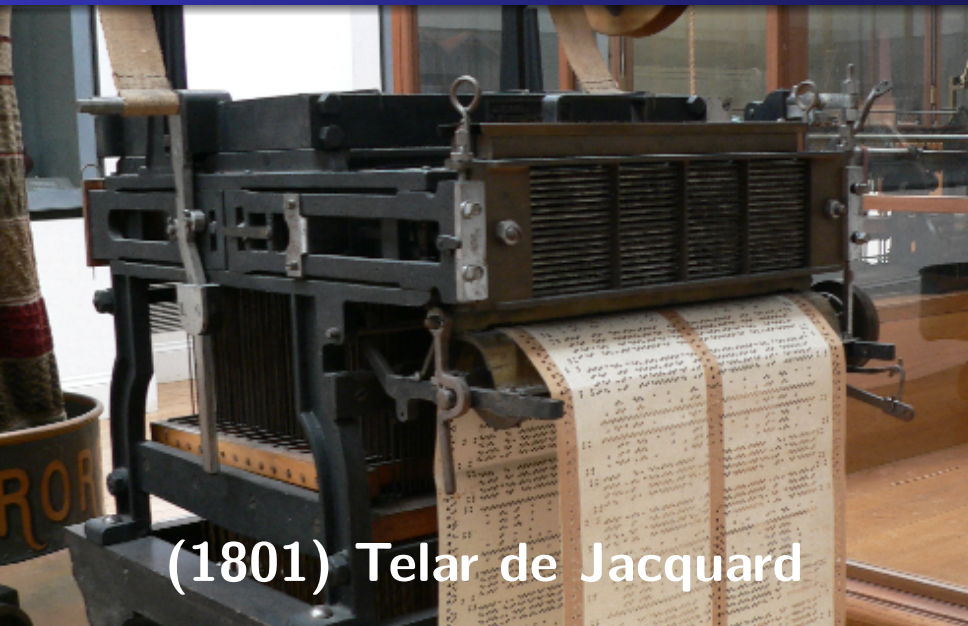
**Desarrollado hace 5000 años
para cálculos**

Pero... ¿De dónde partimos?



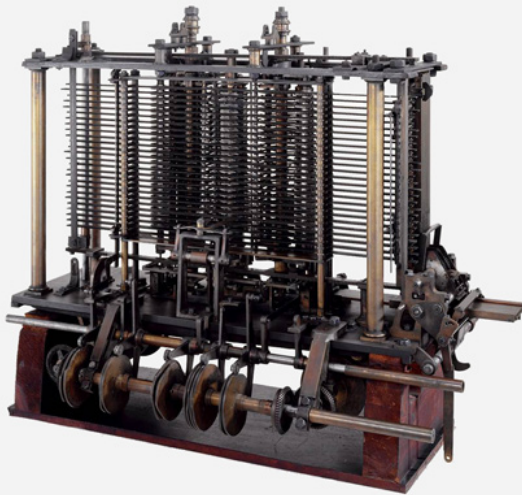
(1652) Primera calculadora

Pero... ¿De dónde partimos?



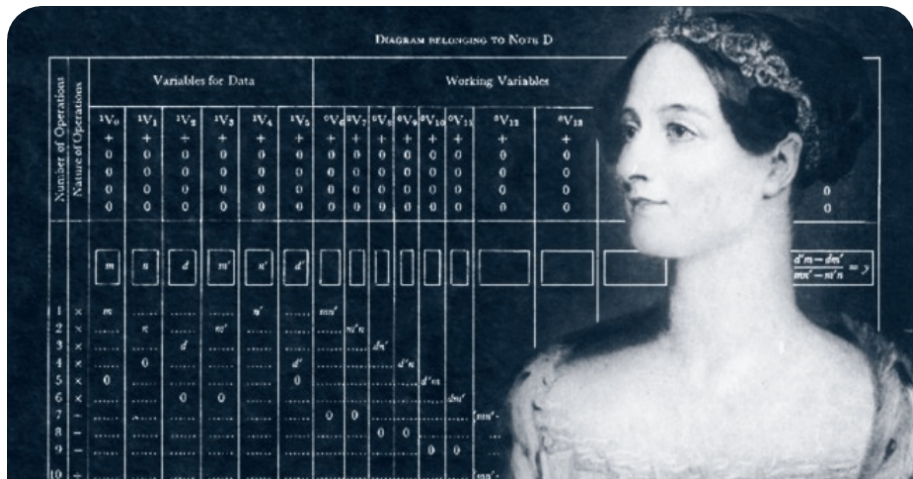
(1801) Telar de Jacquard

Pero... ¿De dónde partimos?



(1837) Máquina Analítica - Babbage

Pero... ¿De dónde partimos?



(1843) Primer programa sobre la Máquina Analítica
Ada Lovelace

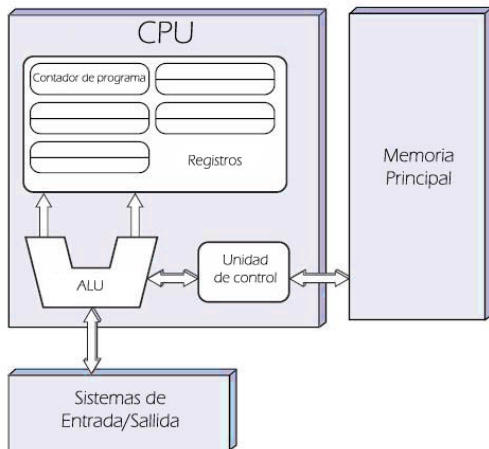
Pero... ¿De dónde partimos?



(1936) Máquina de Turing

Alan Turing

La arquitectura usada para programar



(1945) Arquitectura de Von-Neumann

Tema 1: Introducción

Conceptos Básicos

- **Procesador:** Entidad capaz de entender una serie de acciones y de realizar las tareas indicadas por éstas.
Ejemplos: un electrodoméstico, un ser humano, un ordenador
- **Entorno de trabajo:** Conjunto de elementos necesarios para la realización de dicho trabajo por un determinado procesador.
Las características del entorno varían mucho, según el tipo de trabajo y el tipo de procesador.
- **Acción:** Suceso de duración finita que modifica de alguna forma el entorno de un trabajo.

Conceptos Básicos

- **Acción primitiva para un procesador:** Acción básica que entiende y ejecuta directamente el procesador.
En programación lo conocemos como **Instrucción**. Las acciones primitivas dependen del procesador.
De hecho, una acción más compleja se puede descomponer en una secuencia de acciones más simples equivalente.

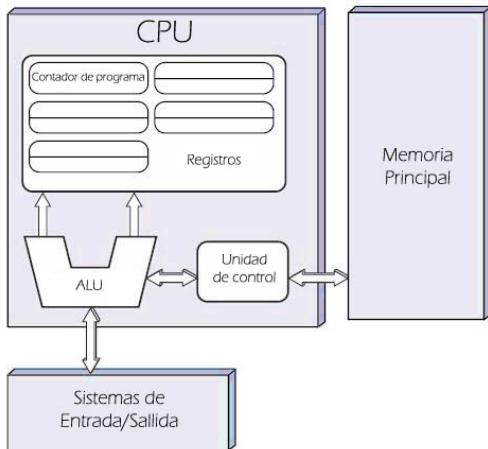
Conceptos Básicos

- **Algoritmo:** Secuencia de acciones primitivas que resuelven un problema en un procesador.
- **Características de un algoritmo:**
 - Debe realizar la tarea
 - Debe ser claro e inequívoco para el procesador
 - Debe definir la secuencia ordenada de pasos requeridos para llevar a cabo la tarea.
 - El número de pasos debe ser finito.

Conceptos Básicos

- **Programa:** Algoritmo que resuelve un problema mediante un procesador informático (ordenador), y que ha sido implementado (escrito) en un lenguaje de programación.
- **Lenguaje de programación:** Notación para describir procesos de cálculo o computacionales.
Lenguaje que entienden y utilizan las computadoras.
- **Proceso computacional:** Conjunto de pasos que deben realizarse en una máquina para cumplir una tarea.

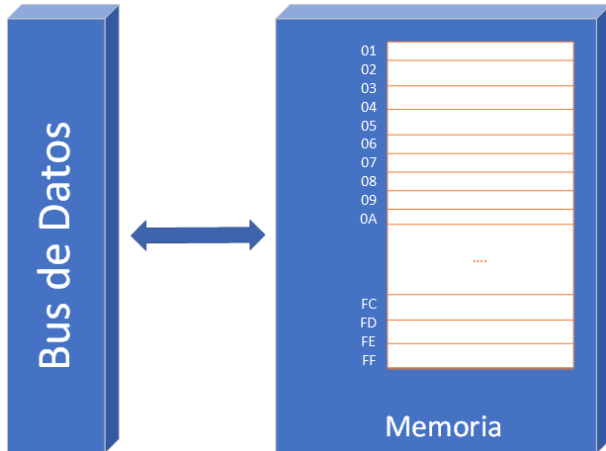
Recordatorio de la arquitectura de Von-Neumann



Estructura y funcionamiento básicos de un equipo

- La CPU es la encargada de ejecutar las instrucciones (programas) almacenadas en la memoria principal.
Ésta se puede dividir en:
 - Unidad de Control: Controla la ejecución de las instrucciones en el orden indicado
 - Registros: memoria rápida, almacena resultados intermedios e información de control: contador de programa, registro de instrucción.
 - Unidad lógico-aritmética o ALU: operaciones matemáticas y lógicas
- Dispositivos de Entrada/Salida: pantalla, teclado, ratón, altavoces, almacenamiento persistente, conexión a red, etc.
- Memoria: con programas, datos y cálculos intermedios.
 - Direcciones de las celdas de memoria: permite acceder a cada celda y leer/escribir los datos en ella.

Estructura y funcionamiento básicos de un equipo



**Cada celda está identificada por una dirección.
Las celdas de memoria pueden ser de 8, 16, 32 ó 64 bits.**

Estructura y funcionamiento básicos de un equipo

Un ordenador tiene un lenguaje muy simple, el binario (0/1). Aunque para trabajar con ellos se agrupan los bits en unidades mayores:

- Normalmente la unidad básica es el byte (8 bits).
- Para que sea más entendible por el ser humano, podemos transformar las cadenas de bits en hexadecimales. Por ejemplo:
 - El byte 0100 1111 se puede traducir en: 4F

Estructura y funcionamiento básicos de un equipo

- Los códigos hexadecimales pueden representar instrucciones, registros de la CPU, direcciones de memoria, interrupciones de entrada/salida, etc...
- Ejemplo:

40 01 EF	Cargar posición de memoria EF en registro 01
40 02 E0	Cargar posición de memoria E0 en registro 02
A0 01 02	Sumar los valores de los registros 01 y 02 y guardarlos en 01
50 FF 01	Guardar el registro 01 en la posición de memoria FF

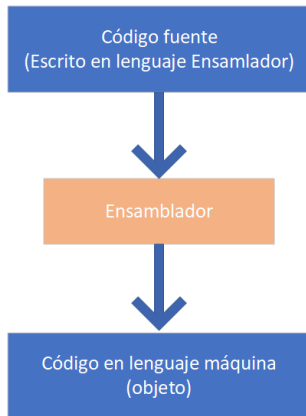
Estructura y funcionamiento básicos de un equipo

- El siguiente paso para incrementar la legibilidad del código es introducir nemotécnicos a las instrucciones y los operandos de forma que sea más "simple" escribir código. Ejemplo:
- 40 → *LOADR*, A0 → *ADD*, 50 → *SAVER*
- *Registro 01* → *R01*, *Registro 02* → *R02*

El código anterior quedaría:

```
LOADR R01 EF  
LOADR R02 E0  
ADD R01 R02  
SAVER FF R01
```


Estructura y funcionamiento básicos de un equipo



Introducción de los compiladores de alto nivel

- Los lenguajes ensambladores son todavía muy complejos
- A partir de los años 1950 se comenzaron a desarrollar lenguajes de alto nivel:
- Son independientes de la máquina.
- Primer lenguaje de alto nivel: Fortran (Formula Translating System). Especialmente diseñado para el cálculo científico.
- Un ejemplo de este tipo de lenguajes es: $x = 5 + 2$ o `print` (“Esto es una prueba”)
- Normalmente las primitivas del lenguaje de programación se construyen con **palabras en inglés, números, cadenas y operadores simples**.

Introducción de los compiladores de alto nivel

- Dos tipos los lenguajes de alto nivel:
 - **Compilados:** Se compilan y enlazan programas completos creando un binario para una arquitectura y un sistema operativo concretos. Ejemplos: C, C++, Pascal, Swift, Basic...
 - **Interpretados:** Compilan, enlazan y se ejecutan instrucción a instrucción. Ejemplos: Python, JavaScript, Ruby, Forth, Matlab.
- Compilador: herramienta que traduce automáticamente un programa en lenguaje fuente a uno equivalente en lenguaje objeto.
- En ese proceso de traducción, el compilador puede notificar errores en el programa origen.

Errores en la programación de alto nivel

- Errores **sintácticos**: sencillos de detectar y corregir, normalmente los detecta el IDE. Ejemplo: “inf” en vez de “int”.
- Errores **semánticos** estáticos: se pueden detectar en el momento de compilación. Ejemplo: Cómo se declaran las variables, si las funciones o métodos se llaman correctamente, las asignaciones son compatibles, etc...
- Errores **no semánticos**:
 - El programa se bloquea y deja de funcionar
 - El programa entra en un bucle infinito
 - El programa devuelve un resultado diferente al esperado

Entorno de trabajo en lenguajes de alto nivel

- **Entorno de un algoritmo:** Conjunto de elementos que usa un algoritmo para solucionar un problema.
- **Elementos:** Los elementos del entorno tienen tres características o atributos fundamentales:
 - Nombre
 - Tipo
 - Valor

Entorno de trabajo en lenguajes de alto nivel

Nombre o identificador de un elemento

- Secuencia de caracteres para identificar y a hacer referencia a un elemento.
- Cada elemento debe tener un nombre distinto.
- Conviene que reflejen su función.

Tipo de un elemento

- Indica y limita el conjunto de valores que puede tomar ese elemento.

Valor de un elemento

- Contenido del elemento.

(Tipo: Entero)

Edad

18

Entorno de trabajo en lenguajes de alto nivel

En esta asignatura usamos **Python**

- Lenguaje interpretado.
- Permite programación imperativa, orientada a objetos, orientada a eventos y funcional.
- Permite la programación de interfaces gráficas de usuario y servicios web.
- Muy usado en muchos campos, como IA, ingeniería de datos, aplicaciones web, etc.

Versiones de Python

- Versiones 2.x: ya no soportadaa aunque hay algunas aplicaciones y distribuciones antiguas de Linux que la siguiente incorporando.
<https://www.python.org/doc/sunset-python-2/>
- Versión 3.x: en el laboratorio usamos la 3.12

Tipos de datos

Tipos básicos de elementos (datos):

- **Variables:** Su valor puede cambiar a lo largo del algoritmo.
- **Constantes:** Su valor no varía durante la ejecución del algoritmo.
- **Constantes sin nombre** (valores)

Antes de empezar a ejecutar un algoritmo hay que tener claros los elementos que van a intervenir.

Para definir una variable:

- Lenguajes tipados: tipo y nombre
- Lenguajes no tipados: nombre
- En algunos lenguajes hay que dar un valor

Python: puede usarse como tipado o no tipado, hay que dar valor

Tipos de datos

Tipos de datos numéricos:

- Enteros (int): positivos y negativos
- Reales (float): positivos y negativos con parte decimal

Notación científica: Mantisa y exponente

Rango de valores: depende del procesador que se esté usando.

Ejemplo de enteros en Python y en C:

En python3. Representado por 64 bits.

```
>>> sys.maxsize
```

```
9223372036854775807
```

// En C. Representado por 32 bits.

```
-2147483648 a 2147483647
```

En Python3: enteros arbitrariamente grandes

Tipos de datos

Tipo carácter:

- Letras mayúsculas y minúsculas 'a' 'A' ...
- Dígitos decimales '0' '1' ... '9'
- Signos de puntuación ' . ' ' ; ' ' : ' ' _ ' ' ? ' ' (' ') '
- Caracteres especiales ' { ' , ' | ' , ' } '
- Códigos estándar: ASCII, UTF-8 (hasta 32 bits).
- En Python: todos los caracteres en Unicode, se convierten de/a códigos al leerlos/escribirlos

Cadena de caracteres: en Python se usa para expresar también caracteres individuales.

Tipos de datos

Tipo lógico (booleano): dos valores posibles

- Verdadero (True)
- Falso (Fals)

Tipos compuestos: Los datos simples se pue den combinar formando datos estructurados.

- Tuplas, listas
- Diccionarios
- ...

Tipos de operaciones y expresiones

- Operador: símbolo que representa una determinada operación.
 $+$ $*$ $/$ $-$ $<$ \geq
- Operando: dato con el que se va a realizar la operación indicada por el operador. $3x$
- Expresión: combinación de de **operadores** y **operandos** siguiendo unas reglas determinadas.
 - Ejemplos: $3 * x$, $-x$, $edad > 18$

Cada operador tiene una cardinalidad (número de operandos):

- Binarios: $+$ $-$ $*$ $/$ AND $<$ $>$ \leq
- Unarios: $-$ NOT

Las expresiones se pueden combinar formando expresiones compuestas.

Tipos de operaciones y expresiones

Operaciones aritméticas

- Suma $+$
- Cambio de signo $-$ (unario)
- Resta $-$
- División real $/$
- Multiplicación $*$
- División entera $//$ (sólo se tomará la parte entera del resultado)
- Resto $\%$ (Cuidado con problemas de redondeo. Ejemplo $10.2 \% 3$)

Se puede dar una situación que hay que tratar con mucho cuidado:

- $X/0$ no está definida (¡ERROR!).
- El programador debe evitar esta situación

Tipos de operaciones y expresiones

Errores de redondeo en los lenguajes de programación:

- El error de redondeo es algo que suele pasar en la mayoría de los lenguajes de programación.
- Este error se debe a que se almacena los números decimales en binario y pasar de decimal a binario provoca errores de redondeo
- Si necesitamos precisión absoluta, debemos utilizar bibliotecas específicas.

Tipos de operaciones y expresiones

Expresiones lógicas : Condiciones

- Operadores relacionales: $Op1 \text{ Opr } Op2$ $<$ $>$ \leq \geq $!=$
- $Op1$, $Op2$ pueden ser expresiones de números, de caracteres (orden ASCII), lógicos (*False* $<$ *True*).
- Operadores booleanos:

NOT Op (No lógico) \rightarrow Verdadero si Op es Falso
Falso si Op es Verdadero

$Op1$ AND $Op2$ Y lógico (Y) \rightarrow Verdadero si ambos son Verdaderos

$Op1$ OR $Op2$ O lógico (O) \rightarrow Verdadero si uno de los dos o los dos son Verdaderos

Tipos de operaciones y expresiones

Expresiones lógicas : Condiciones

Op1	Op2	Op1 Y Op2	Op1 O Op2
Verdadero	Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso	Verdadero
Falso	Verdadero	Falso	Verdadero
Falso	Falso	Falso	Falso

Tipos de operaciones y expresiones

- Prioridad de los operadores
 - Más alta - (unitario) NOT
 - Media * / // % AND OR
 - Más baja + -
- Al igual prioridad, de izquierda a derecha.
- Para cambiar la prioridad, se deben utilizar los paréntesis.

Ejercicio: Evaluar las siguientes expresiones:

$(a+b)/c+2$	con $a=6, b=8$ y $c=7$
$l*-4*T*P$	con $l = 7, T = 3$ y $P = 2$
$A-B+C$	con $A=6, B=2$ y $C=3$
$A*B / C$	"
$A // B * C$	"
$A*B // C$	"
$A+B // C$	"
$A // B // C$	"

Acciones elementales

Asignación:

- Es la operación básica para cambiar el valor de las variables
- Normalmente, la expresión y la variable deben tener tipos compatibles (iguales).
- Primero se evalúa la expresión, y el resultado se almacena en la variable.
- `VARIABLE ← EXPRESIÓN`

Detalles de Python

- Las variables se pueden declarar en cualquier lugar.
- Para acceder al contenido de una variable ha de estar declarada antes.
- No hay un carácter especial para marcar final de instrucción (sí lo tienen C, C++, Pascal)
- Tipado dinámico: Las variables no tienen tipo asociado, pero sí sus valores
 - `a = 29`
 - `print(type(a))`
 - `a = "prueba"`
 - `print(type(a))`

Detalles de Python

Tipos de datos simples (variable = valor)

- Tipos Básicos
 - Enteros (int): `entero1 = 1`, `entero2 = 38907`
 - Reales (float): `r1 = 29.6`
 - cadena de carácter (string): `cadena = "Hola"`
- Transformar de cadena a entero: `int("56")`
- Otras transformaciones en diferentes sistemas
 - Octal (base 8): `o1 = 0o1`, `o2 = 0o6645`, `o3 = oct(12)`
 - Hexadecimal (base 16): `h1 = 0x1`, `h2 = 0xDA5`, `h3 = hex(12)`
 - Binario (base 2): `b1 = 0b100111000`, `b2 = bin(12)`

Detalles de Python

Apreciación para estos primeros días de clase

- Para que vayáis pensando los tipos de las variables durante unas clases vamos a definir las variables con el tipo que se supone que debe tener.
- Desde Python 3.6 se pueden declarar tipos tanto en variables como en funciones.
- Ojo, los tipos siguen siendo dinámicos sólo sirven a modo de información para los desarrolladores o para información adicional a los IDEs.
- Para definir un tipo en una variable:
 - `nombreVariable: tipo`

Detalles de Python

- Operadores

- Los operadores básicos que hemos visto anteriormente (+, -, *, /)
- También se pueden usar los incrementadores decrementadores +=, -=
Ejemplo: Variable += 1 \leftarrow Variable = Variable + 1
- ¡Cuidado! porque no existe ++, -- como en otros lenguajes

- Exponenciación: 4**2

- División

- División entera: 6//7 #0
- División real: 6 / 7 #0.8571428571428571
- Resto: 12% 5 # 2

- Cadenas

- Las cadenas de caracteres se definen con los caracteres " " o ' '. Es importante no mezclar ambos tipos de caracteres
- Los caracteres \n introduce un salto de línea y \t introduce una tabulación, siempre dentro de las comillas
- Los operadores lógicos para comparar elementos son: ==, !=, <, ≤, >, ≥, and, or, not. Todos ellos devuelven los valores: True o False

Gestión de variables

- `dir()`: Lista de variables declaradas hast el momento
- `del(variable)`: Elimina la variable cuyo nombre va entre los paréntesis del espacio de nombres.
 - Ojo: las variables en Python funcionan como etiqueta
 - `del()` borra la “etiqueta”, no su contenido.

Mostrar texto

- `print()`
- En el intérprete no hace falta:
la ejecución de las instrucciones muestra su evaluación
- En ficheros `.py` sí hace falta
- Ejemplos:
 - `print("Hola" + "Mundo")`
 - `print("Hola", "Mundo")`
 - `print("Hola", "Mundo", end="")`

Leer de teclado

- `input()`: Para obtener información del teclado
- Además, mensaje que será mostrado al usuario
`input("MENSAJE")`
- El usuario escribirá con el teclado y pulsará <INTRO>
- La salida de `input()` puede asignarse a una variable:
`variable = input ("MENSAJE")`
- Devuelve un **string**
Hay que convertir el tipo si se quiere un número

```
num = int (input ("Introduce un número"))  
print (5*num)
```

Ficheros py

- Se pueden crear estos ficheros con cualquier editor de texto.
 - En consola con el vim o el nano
 - En interfaz gráfica con cualquier editor de texto como GEdit o Kate
 - Con el IDE, por ejemplo PyCharm
- Ejecución:
`python3 programa.py`

Comentarios en Python

Documentar ayuda a que...

- ...otras personas entiendan el código
- ...tu yo del futuro entienda el código

Formas de añadir un comentario:

- Símbolo de numeral o hashtag (#) en una sólo línea.
- Tres comillas dobles (""") o simples (' ' ') al inicio y final del bloque.

Ejercicios

- Escribe un programa que lea un precio y un porcentaje de descuento, muestre ambas variables y aplique el descuento al precio final.
- Leer una cantidad de segundos y pasarlos a días, horas, minutos y segundos.
- Leer valores para dos variables, intercambiar los valores, y escribirlos.
- Lee la base y la altura de un triángulo y calcula su área.
- Escribe un programa que calcule si un valor dado está entre un valor superior e inferior.
- Leer el radio de una circunferencia y escribir su longitud y área.
- Leer el valor de un año y calcular si es bisiesto o no.

Tema 2: Estructuras de control

Conceptos Básicos

Una correcta **estructura** nos permite:

- Corregir errores fácilmente haciendo una depuración eficaz.
- Añadir nuevas funcionalidades sin un gran esfuerzo
- Modificar y mejorar funciones ya que estarán bien definidas
- Realizar un programa entre varias personas
- Revisar el programa en el futuro.
- Aumentar la **legibilidad** del programa.

Conceptos Básicos

Principios básicos de la programación estructurada:

- Diseño modular descendente → Abstracción a distintos niveles
 - Gracias a este diseño se puede afrontar un proyecto de programación empezando por lo más general e ir avanzando nivel a nivel hacia lo más particular.
- Estructuras de control que sustituyen a la bifurcación (GOTO es el enemigo de la estructuración). 60-70: Dijkstra, Wirth, ...

Estructuras de control básicas

Los programas pueden y deben construirse utilizando única y exclusivamente estas tres estructuras básicas de control de flujo:

- Secuencia
- Alternativa o decisión
- Iteración, bucle o ciclo

Böhm y Jacopini formularon este teorema en el año 1966 de forma que se deducía que sólo a través de estas estructuras se podrían diseñar los algoritmos.

Secuencia

Las instrucciones se ejecutan una detrás de otra, en el mismo orden en que aparecen. Ejemplo: Cálculo de la media de tres notas

Algoritmo Media

Inicio

~~~| Leer N1, N2, N3

~~~|  $\text{Media} \leftarrow (N1 + N2 + N3) / 3$

~~~| Escribir Media

Fin

# Secuencia

## Pseudocódigo

acción 1

acción 2

acción 3

...

acción n

## Organigrama



# Alternativa o decisión

Permite elegir entre la realización de una acción u otra según una condición (expresión lógica).

Ejemplo: Decir si un número entero es par o impar

# Alternativa o decisión

Pseudocódigo

si **CONDICIÓN** entonces

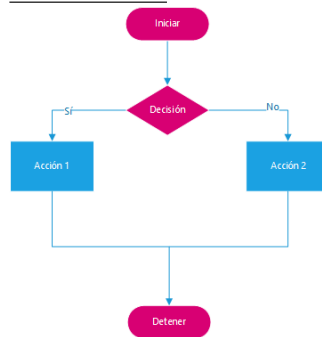
acción 1

si no

acción 2

fin si

## Organigrama



- La **CONDICIÓN** debe tener valor Verdadero o Falso
- Si se evalúa la **CONDICIÓN** como Verdadera, se ejecutará la Acción 1. Si es Falsa se ejecutará la Acción 2.

# Alternativa o decisión (simplificada)

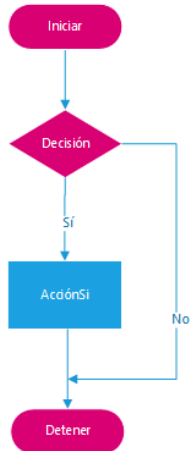
Pseudocódigo

si **CONDICIÓN** entonces  
    acciónSi

fin si

Ejemplo: Calcular el valor absoluto de un número real.

## Organigrama



# Alternativa o decisión en Python

```
if <condicion>:  
    ^^|<expresion>  
    ^^|<expresion>  
    ^^| ...
```

```
if <condicion>:  
    ^^|<expresion>  
    ^^|<expresion>  
else :  
    ^^|<expresion>  
    ^^|<expresion>
```

```
if <condicion>:  
    ^^|<expresion>  
    ^^|<expresion>  
elif <condicion>:  
    ^^|<expresion>  
    ^^|<expresion>  
else :  
    ^^|<expresion>  
    ^^|<expresion>
```

# Alternativa o decisión en Python: Indentación

- La indentación es mover un bloque de textos a la derecha insertando **espacios** o **tabuladores**.
- A través de este mecanismo se denotan los bloques de código.

```
x = float(input("Introduce un número para x: "))
y = float(input("Introduce un número para y: "))
if x == y:
    print("x e y son iguales")
elif y != 0:
    print("Ademas, x / y es", x/y)
elif x < y:
    print("x es mas pequeno")
else:
    print("y es mas pequeno")
print("Fin del programa")
```

# Iteración, bucle o ciclo

- Una acción o grupo de acciones se repite una serie de veces.
- El número de veces viene determinado por una **condición** (expresión lógica) que alguna vez debe hacerse **falsa**. (**¡Cuidado con los bucles infinitos!**)
- La acción se ejecuta 0 ó más veces.
- Ejemplo: Escribir los dobles de los números que se van leyendo hasta que se lea un 0.



# Iteración, bucle o ciclo

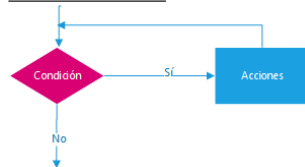
Pseudocódigo

mientras **CONDICIÓN** hacer

Acciones

finmientras

Organigrama



# Iteración, bucle o ciclo

- La condición de terminación del bucle debe usar alguna variable cuyo valor cambie al ejecutar las acciones internas del bucle.
- La variable debe estar inicializada antes de comprobar la condición.
- Ejemplo: Escribir la tabla de multiplicar de un número.

# Iteración, bucle o ciclo. Bucle while en Python

El bucle mientras se implementa en Python con while:

```
while <condicion>:
```

```
    ^|^<expresion>
```

```
    ^|^<expresion>
```

```
    ^|^...
```

## Recordad

- La <condicion> se evalúa como una expresión lógica (Boolean)
- Si la <condicion> es Verdadera, se ejecutan el resto de las expresiones dentro del bloque de código (identado)
- Se vuelve a comprobar la <condicion>
- Se repite hasta que la <condicion> sea Falsa

# Iteración, bucle o ciclo

para Variable de Explnicio a ExpFin incremento Explnc hacer  
    Acciones  
FinPara

- Incremento Explnc es opcional. Por omisión, su valor es 1.
- La variable contador ...generalmente es entera, aunque puede ser de otros tipos que veremos más adelante.
- Si  $\text{Explnicio} > \text{ExpFin}$ , las acciones no se ejecutarán ninguna vez.

# Iteración, bucle o ciclo

- **Dentro del bucle Para NO se puede cambiar el valor de la variable contador.**
- Al salir del bucle PARA, el valor de la variable queda indefinido ( o no; no importa)
- Se pueden hacer bucles con valores descendentes en la variable de control con incremento **negativo**.
- Ejemplo: Tabla de multiplicar de un número

# Iteración, bucle o ciclo. Pseudocódigo de Para implementado con un bucle Mientras

## SIMULACIÓN:

Variable  $\leftarrow$  ExpInicio

Mientras Variable  $\leq$  ExpFin hacer

    Acciones

    Variable  $\leftarrow$  Variable + ExpInc

FinMientras

Ejercicio 1: Implementarlo en Python.

Ejercicio 2: Haz la simulación descendente.

# Iteración, bucle o ciclo. Bucle For en Python

El bucle mientras se implementa en Python con while:

```
for <variable> in range (<cjto_numeros>):  
    ^|<expresion>  
    ^|<expresion>  
    ^|...
```

## Recordad

- Cada vez que se ejecute el bucle, la <variable> toma un nuevo valor
- La primera vez, <variable> comienza con el valor más pequeño (ExpInicio)
- La siguiente vez, <variable> será su valor actual más el incremento definido.
- Se repite hasta que la <variable> haya alcanzado ExpFin

# Herramienta - range

- Range permite generar un rango de valores.
- El formato del comando es:

**range** (inicio , fin , incremento)

- Los valores por defecto de range son inicio=0 e incremento=1. De hecho son opcionales.
- El bucle se ejecuta hasta fin - 1.

## Ejemplo:

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

**Ejercicio:** Escribid la misma aplicación anterior pero que comience el rango en 11, finalice en 21 y se incremente de 2 en 2.



# Herramienta - break

- Break permite salir de cualquier bucle que se esté ejecutando en ese momento.
- Al finalizar la ejecución, cualquier expresión que haya tras el break (indentada) no se ejecutará.
- Sólo finaliza el bucle donde se ejecuta. Si hay más bucles anidados seguirán siendo ejecutados.
- Ejemplo:

```
while <condicion_1>:  
    ~| while <condicion_2>:  
        ~| ~| <expresion_1>  
        ~| ~| break  
    ~| ~| <expresion_2>  
    ~| <condicion_3>:  
    ~| ...
```

# Comparativa de los bucles for y while

## Bucle for:

- Debes usarlo cuando **conoces** el número de iteraciones
- se puede **finalizar** usando la instrucción break
- Usa un **contador**
- Se **puede reescribir** un bucle for con un while.

## Bucle while:

- Debes usarlo cuando **no conoces** el número de iteraciones.
- se pueden **finalizar** usando la instrucción break.
- Puedes usar un **contador pero debe ser inicializado** antes del bucle e incrementarlo dentro de él.
- Normalmente **no es posible reescribir** un bucle while con un bucle for.

# Excepciones

- Python utiliza una estructura especial para la **gestión de errores** durante la ejecución del programa. Por ejemplo:
  - División de un número entre 0. Ej: 5/0.

```
>>> PRINT(5 / 0)
TRACEBACK (MOST RECENT CALL LAST):
FILE "<STDIN>", LINE 1, IN <MODULE>
ZERODIVISIONERROR: DIVISION BY ZERO
```

- Cuando ocurre un error durante la ejecución del programa Python crea una excepción. Esta detendrá la ejecución del programa y muestra el error (tracebak).
- Esto ocurrirá siempre que no controlemos esa excepción.

Los principales tipos de excepciones que nos podemos encontrar son:

- `TypeError` : Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.
- `ZeroDivisionError` : Ocurre cuando se intenta dividir por cero.
- `NameError` : Se referencia a una variable que no existe.

*Existen algunos más y los iremos viendo a lo largo del curso*

# Excepciones

Para que Python controle las excepciones, se utilizan unos **manejadores** que permiten gestionarlás.

Ejemplo:

```
try :  
    a = int(input("Introduce el 1º numero:"))  
    b = int(input("Introduce el 2º numero:"))  
    print(a/b)  
except :  
    print("Error en los numeros introducidos.")
```

Cuando se lanza la excepción por cualquier instrucción dentro del bloque **try**, se maneja en la parte del **except** y la ejecución continúa con las instrucciones que están dentro de este bloque.

# Excepciones

Se pueden introducir cláusulas **except** separadas para gestionar cada tipo de excepción.

Ejemplo:

**try :**

```
    a = int(input("Introduce el 1º número: "))
```

```
    b = int(input("Introduce el 2º número: "))
```

```
    print(a/b)
```

**except ValueError:**

```
    print("Error conversion a número.")
```

**except ZeroDivisionError:**

```
    print("No se puede dividir por cero.")
```

**except:**

```
    print("Algo fue muy muy mal.")
```

# Excepciones

Existen otros modificadores de excepciones.

- **else:** Se usa como manejador cuando el bloque del **try** se ejecuta **sin excepciones**.
- **finally:** Este bloque **siempre** se ejecuta después de un bloque **try**, **el se y except**. Tiene las siguientes peculiaridades:
  - Este bloque se ejecutará aunque se haya lanzado por otro error o se ejecute una instrucción **break**.
  - Normalmente es muy útil cuando se quiere ejecutar código que debe ejecutarse independientemente de lo que haya pasado en los bloques anteriores. (Ejemplo: cerrar un fichero).

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 1 Escribir un algoritmo que lea por teclado un número entero no negativo  $N$ , y escriba una lista con las  $N$  primeras potencias de 2.
- 2 Escribir un algoritmo que lea por teclado un entero no negativo  $N$  y escriba una lista con las potencias de 2 que sean menores o iguales que  $N$ .
- 3 Construir un algoritmo que realice la suma de todos los números impares comprendidos entre dos números enteros no negativos leídos
- 4 Escribir un algoritmo que lea por teclado calcule el factorial de un número entero no negativo.
- 5 Escribir un algoritmo que lea por teclado dos horas en el formato de horas, minutos y segundos, y calcule la diferencia en segundos entre ambas

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 6 Diseñar un algoritmo que lea por teclado dos números enteros no negativos y escriba el producto. No se puede utilizar el producto de enteros en este algoritmo, sólo la suma y la resta.
- 7 Escribir un algoritmo que lea por teclado una secuencia de números positivos acabada en 0 y calcule la media aritmética.
- 8 Escribir un algoritmo que lea por teclado una secuencia de números positivos acabada en 0 y encuentre el menor y el mayor de los números leídos.
- 9 Hacer un algoritmo que diga si un número entero positivo es primo o no.
- 10 Escribir un algoritmo que, dado un número entero positivo, encuentre el divisor más grande (que no sea él mismo).



# Tema 3: Divide y vencerás

# Principios básicos de la programación estructurada

- Primer intento de solución.
- Propiciada por la aparición de asociaciones como ACM (Association of Computer Machinery) e IEEE (Institute of Electronic and Electrical Engineering).
- Se introducen una serie de **estructuras de control básicas** en los lenguajes de programación y una **metodología** de uso para 'estandarizar el estilo de programación'.
- Propósito: construir programas más legibles y fáciles de mantener y modificar:
  - Estructuras de control: **secuencia, selección e iteración.**
  - Salto incondicional (**goto**) → **¡Prohibido!**
  - Elegir la estructura de datos adecuada.
- De esta manera, se construyen programas con un único inicio, un único final, y un único flujo de programa, de arriba a abajo.
- **Problema:** los programas grandes siguen siendo muy difíciles de mantener y caros.

# Principios básicos de la programación modular

- Cuando tenemos que enfrentarnos a un problema complejo, abarcarlo de primeras es casi imposible.
- Una de las principales metodologías que hay para el desarrollo de aplicaciones es **divide y vencerás**.
- Se agrupan trozos de programas que se encargan de hacer una misma tarea dentro de una unidad común → **MÓDULOS**
- Los programas grandes necesitan más de un programador para desarrollarlo. Descomponiéndolo, podemos dividirlo fácilmente entre varias personas (**Principio de descomposición**).
- Criterio a la hora de dividir un programa en módulos → **Experiencia del analista**.

# Principios básicos de la programación modular

- Ventajas de la programación modular:
  - Un programa puede escribirse entre varias personas
  - Librerías de rutinas
  - Verificación de módulos por separado
  - Simplifica el diseño del programa → más sencillo de mantener y modificar
- Características de un módulo:
  - Posee un nombre que identifica las acciones que desarrolla
  - Puede llamar a otros módulos, devolviéndose el control al que realiza la llamada cuando terminan su ejecución
  - Única entrada y única salida
  - Tamaño reducido: fácil de modificar, verificar, reutilizar
  - Oculta lo que lleva en su interior: **CAJA NEGRA**: se conoce cómo llamarlo, qué devuelve, pero **no** cómo lo hace (**Principio de abstracción**).

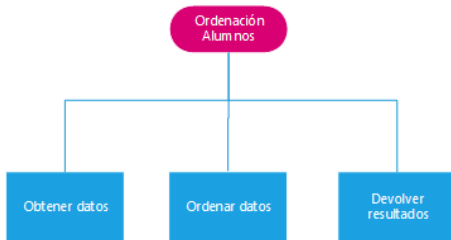
# Principios básicos de la programación modular

- ¿Cómo se divide una aplicación en módulos? **REFINAMIENTOS SUCESIVOS**

Proceso iterativo en el cual los detalles se relegan cuanto sea posible. En cada paso, algunos detalles se tienen en cuenta y el resto se 'refinarán' posteriormente.

- Así, obtenemos una descomposición modular de la aplicación; con módulos dependientes unos de otros → **DISEÑO DESCENDENTE**

Ejemplo: ordenación alfabética de los alumnos de una clase



# Definición de módulos

- Definición de un módulo (o subprograma)
  - La cabecera del módulo debe especificar el nombre, los parámetros del mismo y, dependiendo del lenguaje, el valor devuelto.
  - Dentro del módulo se especificarán las diferentes sentencias que resuelven el subproblema.
- Usando el **principio de abstracción**, allí donde sea necesaria la ejecución del módulo se realizará la invocación especificando el nombre del módulo y pasando las variables requeridas para la ejecución del mismo.
- El módulo deberá ser definido **siempre** antes de la invocación.
- Los módulos permiten definir los modos de procesar y operar los parámetros de entrada:
  - **Funciones** : calculan las diferentes salidas del módulo a partir de la información de entrada.
  - **Procedimientos**: procesan la información de entrada.

# Definición de módulos

- Los **procedimientos** permiten el procesamiento de la información en general.
  - La cabecera de un procedimiento no especifica que se devuelva ningún valor (ya que no lo hace), el nombre del módulo y los parámetros de entrada de la función.
  - El cuerpo estará formado por las diferentes sentencias que ejecutan las diferentes funcionalidades para la que fue concebido el procedimiento.
  - No se deberá devolver ningún valor como resultado del procedimiento a través de algún mecanismo especial como en el siguiente caso.
- Las **funciones** calculan los valores de retorno a partir de las variables de entrada
  - Normalmente la cabecera de una función especifica el valor devuelto, el nombre del módulo y los parámetros de entrada de la función.
  - El cuerpo estará formado por las diferentes sentencias que permitan el cálculo del resultado que deberá ser devuelto por la función.
  - La última sentencia del cuerpo de la función será normalmente una palabra reservada que devuelva el valor de resultado. Por ejemplo: **return**. Aunque haya otras sentencias no se ejecutarán.

# Funciones en Python

Las funciones en Python se definen: Ejemplo:

```
def nombre (parametro1:tipo , parametro2:tipo):  
    """
```

*Definicion de la funcionalidad del modulo*

*def → Palabra reservada*

*nombre → nombre modulo (debe ser descriptivo)*

*parametro1, ... → argumentos de la funcion*

*return → devuelve un valor (cuidado con los tipos).*

```
    """
```

```
        instruccion1
```

```
        instruccion2
```

```
        .
```

```
        .
```

```
    return valor
```

```
varRetorno = nombre (var1 , var2)#Llamada funcion
```



# Funciones en Python

**Posibles errores que nos podemos encontrar al diseñar un módulo con return.** Ejemplo: Define un módulo que calcule el cuadrado de un número par

```
def cuadradoPar (numero:int):  
    if not numero % 2 ==0:  
        return #podemos usar pass.  
    else:  
        print (numero **2)
```

```
cuadradoPar (16)  
cuadradoPar (5)  
print ("Prueba de funcion")  
print (cuadradoPar(5))
```

# Funciones en Python

- Python, a diferencia de otros lenguajes de programación, **no tiene procedimientos**. Un procedimiento sería como una función pero que no devuelve ningún valor.
- **Pero**, por la naturaleza de Python, aunque no se devuelva ningún valor (no se use return o return no devuelva nada como el ejemplo anterior), **la función devolverá el valor None**.
- Hay que tener en cuenta de cubrir **todas las opciones de retorno**.
- Las funciones en python pueden devolver más de un valor con return.  
Ejemplo:

```
def cuadradoMitad (numero: int):  
    return numero**2, numero/2
```

```
num = 5  
cuadrado, mitad = cuadradoMitad(num)  
print(f"El cuadrado de {num} es {cuadrado} y \\  
su mitad {mitad}")
```

# Parámetros de las funciones

- Cuando se realiza la invocación de un subprograma se debe llamar con el **número de parámetros** que marque la cabecera de la misma.
- En la mayor parte de los lenguajes de programación se establecen dos formas de comunicación entre el programa / subprograma que llama y la función/procedimiento llamado:
  - A través de la copia de valores usando el mecanismo de **paso por valor**. Es decir, se copia el valor de la variable que hace la llamada a la función/procedimiento (subprograma). Cualquier modificación del valor del parámetro dentro del subprograma no afecta a la variable con la que se llamó.
  - A través de la vinculación de las variables usando el mecanismo de **paso por referencia**. A través de este método lo que estamos copiando es la dirección de la memoria de las variables. De esta forma se estará usando la misma variable fuera del subprograma como en la llamada.

# Parámetros de las funciones en Python

- Python como muchos otros lenguajes modernos tienen su base en el paradigma de Programación Orientado a Objetos (los veremos al final del curso).
- Todo en Python es un objeto (Un bloque de código completo formado por datos y las funciones que manejan a esos datos).
- Lo que se pasa en Python al hacer una llamada a función es una referencia a ese objeto (paso por referencia). Su capacidad de modificación dependerá de un parámetro que marque si son objetos inmutables o mutables:

## No mutable

- Números
- Strings
- Tuplas

## Mutable

- Listas
- Conjuntos
- Diccionarios

# Parámetros de las funciones en python

```
def noMutable(x: int):  
    x = x+1  
    return x
```

```
def mutable (l: list):  
    l[1]= 'O'
```

```
valorNoMutable = 5  
returnNoMutable = noMutable(valorNoMutable)  
#Creamos una lista  
valorMutable = ['L', 'A', 'U', 'R', 'A']  
returnMutable = mutable(valorMutable)  
  
print (f"Prueba de parametros en una funcion \  
No mutable {valorNoMutable} {returnNoMutable}, \  
Mutable {valorMutable} {returnMutable}")
```

# Ámbito de las variables

- El ámbito de las variables es el **contexto en el que existe la variable**. El ámbito será desde que se crea en memoria la variable hasta que deja de existir en ella.
- Cuando se define el programa principal se comienzan a definir las variables que se usarán. El contexto de estas variables será desde que comience el programa hasta que finalice. Estas variables serán accesibles desde cualquier punto del programa, incluso desde los módulos (funciones/procedimientos). **A estas variables se les conoce como globales.**
- **OJO! debemos evitar el acceso a las variables globales desde otros puntos que no sean el programa principal.** Ejemplo:

```
def hola():  
    print (f"Hola_{nombre}")
```

```
nombre = "David"  
hola()
```

# Ámbito de las variables globales en Python

- De hecho para evitar problemas que puedan surgir de la modificación incontrolada de variables globales, Python por defecto no deja modificar su valor. Ejemplo:

```
def incremento():  
    print (x+1)  
    x += 1
```

```
x = 5
```

```
incremento()
```

- Para que deje usar `x` dentro de la función hay que definirla como global poniendo la directiva **global nombreVariable** antes de usarla.
- Para esta asignatura está totalmente prohibido tanto el acceso a las variables globales como la modificación de las mismas, a menos que lo digamos explícitamente.**
- ¿Cómo resolverías el problema de usar las variables globales con funciones?*

# Ámbito de las variables

- Cuando el ámbito de la variable está localizada a una función específica se conoce que la variable es de **ámbito local**.
- Cuando se hace una llamada a un módulo el control de la aplicación pasa a ejecutar la primera instrucción del cuerpo del módulo.
- En ese momento las variables locales se crean a medida que se vayan definiendo en el módulo.
- Cuando finaliza la ejecución del módulo, los parámetros y variables locales creadas se destruyen, retornando el control de flujo por la siguiente instrucción a la que invocó el módulo. Ejemplo:

```
def hola():  
    nombre = "David"  
    print (f"Hola_{nombre}")
```

```
hola()  
print (nombre)
```



# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 1 Escribir un módulo que, dado un número  $X$  y un número entero  $N$ , dibuje una línea que incluya  $N$  veces el número  $X$ .
- 2 Escribir un programa que lea un número e imprima un triángulo de números de la forma siguiente: Si el número leído es 4, se imprimirá:  
1  
22  
333  
4444
- 3 Escribir una función que lea una secuencia de enteros acabada con -1 y determine si es estrictamente creciente.
- 4 Construir una rutina que acepte como parámetro un número entero y devuelva el número de cifras de dicho número.

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 5 Diseñar un módulo que, dado un número entero  $N$ , y una posición  $i$ , indique cuál es la cifra decimal que ocupa la posición  $i$ -ésima dentro del número  $N$ .
- 6 Diseñar una rutina que acepte un número entero positivo y lo devuelva al revés. Por ejemplo, si el número de entrada es 1234, la rutina debería devolver el 4321.
- 7 Construir una función que, dado un número natural, calcule cuántos 1 aparecen en su representación.
- 8 Escribir una función que calcule el término  $n$ -ésimo de la serie de Fibonacci, definida de la siguiente manera:

$$\text{FIB}(1) = 1$$

$$\text{FIB}(2) = 1$$

$$\text{FIB}(n) = \text{FIB}(n-1) + \text{FIB}(n-2) \text{ si } n > 2$$

# Tema 4: Estructuras de Datos

# Introducción

- Una definición que se hizo muy famosa en los años 70 en el ámbito de la programación fue realizada por Wirth:
  - Algoritmos + Estructuras de datos = Programas
- Podemos hacer una clasificación de los datos de la siguiente manera:
  - **Datos simples:** real, carácter, entero, lógico, etc.
  - **Datos compuestos:** grupos de datos relacionados. Ejemplos: fechas, datos bibliográficos o personales, etc.
- Una **estructura de datos** es un conjunto de datos relacionados entre sí de alguna manera determinada.
- Por tanto, una **estructura de datos** se caracteriza por la relación que existe entre sus elementos y por las funciones de acceso a ellos.

# Introducción

## Podemos clasificar las Estructuras de Datos en:

- **Estructuras de datos estáticas:** tienen un tamaño determinado de antemano, y no puede modificarse durante la ejecución del programa. Ejemplo: una fecha.
- **Estructuras de datos dinámicas:** pueden variar su tamaño durante la ejecución según las necesidades. Ejemplo: fichero de una biblioteca.

Todos los lenguajes de programación nos permitirán construir estructuras dinámicas y estáticas. El tipo de problema será determinante para seleccionar un tipo u otro.

# Tipos abstractos de datos

- Para el manejo de las estructuras de datos en un programa normalmente se utilizan un modelo que se conoce como **tipo abstracto de datos** (TAD).
- Este modelo matemático es definido por unos datos y una serie de operaciones que actúan sobre ellos de una manera determinada.
- **Ejemplo - Los números enteros:** tenemos unos datos ( el cero y el resto de los números [ sucesores y antecesores]) y una serie de operaciones ( suma, resta, \*, son\_iguales?, etc.).
- Un TAD es una herramienta formal que nos permite generalizar el concepto de tipo de datos.
- Un lenguaje de programación ya nos ofrece una serie de TADs implementados para que trabajemos con ellos sin necesidad de definirlos ( enteros, reales, booleanos, caracteres). **Son los tipos de datos simples.**

# Tipos abstractos de datos

- El TAD es una herramienta **sintáctica** : definimos operaciones, pero debemos decidir qué significan.
- Al desarrollar aplicaciones debemos hacer programas estructurados y modulares, la idea es diseñar algoritmos que trabajen con los TADs sin necesidad de conocer su implementación detallada (principio de abstracción).
- Si un TAD está bien diseñado, su funcionamiento siempre será igual, independiente del resto del programa.

# Especificación de un TAD

- El funcionamiento de un TAD no debe depender de su implementación concreta, sino **sólo de la especificación que define su comportamiento**.
- Hay que estudiar el comportamiento deseado: constantes, operaciones necesarias (básicas o a partir de las otras).

Ejemplo de un TAD:

TAD

Natural es Cero, Sucesor, Igual, Suma

OPERACIONES

Cero() **retorna** Natural

*efecto* devuelve el numero natural 0

Sucesor( $N : \text{Natural}$ ) **retorna** Natural

*efecto* devuelve el numero natural que sucede a N

Igual( $N, M : \text{Natural}$ ) **retorna** Booleano

*efecto* si  $N=M$  devuelve Verdadero sino devuelve Falso

Suma( $N, M : \text{Natural}$ ) **retorna** Natural

*efecto* retorna  $N+M$



# Representación de TADs en Python

- Para representar los TAD y las librerías de funciones y procedimientos, en Python se pueden usar los **módulos**.
- Un módulo en Python es **un fichero .py** que alberga las funciones, variables y objetos (del paradigma de Programación Orientado a Objetos).
- La forma de definición es sencilla, se crea el fichero y se incluyen tanto las estructuras de datos como las funciones necesarias para su manejo.
- Ejemplo:

# Representación de TADs en Python

*#TAD Numero Natural*

```
def cero():  
    return 0
```

```
def sucesor (N:int):  
    return N+1
```

```
def igual (N:int , M:int):  
    return N==M
```

```
def suma(N:int , M:int):  
    return N+M
```

# Representación de TADs en Python

```
#Usando import únicamente  
import natural
```

```
print (natural.sucesor(5))  
print (natural.suma (4,5))  
print (natural.igual (3,3))
```

```
#Importando las funciones que nos interesen  
from natural import suma, sucesor
```

```
print (sucesor(5))  
print (suma (4,5))
```

# Definición de tupla

- Una tupla es una secuencia **inmutable e indexable** de elementos.
- Normalmente se usa para agrupar valores que deben ir juntos por la lógica del programa.
- Sintaxis:

```
tupla = (elemt1 , elemt2 , ... , elemtn)
```

```
tupla1 = () # tupla vacia
```

```
tupla2 = (1,2,3,4,5) # tupla de numeros
```

```
tupla3 = ('David', 'Cortés') # tupla de strings
```

```
tupla4 = ('David', 'Cortés', 18) # tupla mixta
```

```
tupla5 = (tupla2 , tupla3 , tupla4) # tupla de tuplas
```

# Operaciones con tuplas

- Propiedad de **indexación**: Los elementos se obtienen por su **índice**

```
print(tupla5[0][2]) # >>> 3
print(tupla5[1][-1]) # >>> Cortés
print(tupla5[2][3]) # >>> 18
```

- Propiedad **inmutable**: Los elementos **no** pueden ser modificados

```
tupla2[0] = 2 # >>> Error
tupla2[0] = tupla2[0]+2 # >>> Error
```

- Para iterar una tupla se puede utilizar un bucle for:

```
for i in tupla2:
    print(i) # >>> 1, 2, 3, 4, 5
```

# Operaciones con tuplas

- Se pueden usar los operadores: **+**, **\***, **==**, **in**, **not in**

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)
print (t1 + t2) # >>> (1, 2, 3, 4, 5, 6)
print (t1 * 2) # >>> (1, 2, 3, 1, 2, 3)
print (2 in t1) # >>> True
print (4 not in t2) # >>> False
print (t1 == t2) # >>> False
```

# Operaciones con tuplas

- También se puede obtener una sub-tupla mediante la siguiente estructura

```
t3 = (1, 2, 3, 4, 5, 6)
print (t3[:3]) # >>> (1, 2, 3)
print (t3[4:]) # >>> (5, 6)
print (t3[::-1]) # >>> (6, 5, 4, 3, 2, 1)
```

- También hay funciones para operar las tuplas y acceder a los contenidos de los mismos: **len()**, **min()** y **max()**.

```
t3 = (1, 2, 3, 4, 5, 6)
print (min(t3)) # >>> 1
print (len(t3)) # >>> 6
print (max(t3)) # >>> 6
```

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- ❶ Cree una función que reciba un conjunto de datos de usuarios de usuarios (nombre, apellido, correo-electrónico, edad) y muestre:
  - El número de usuarios.
  - El nombre del usuario más mayor.
  - El nombre del usuario más joven.
  - El nombre del usuario con apellido más largo.
  - La edad promedio de los usuarios.
- ❷ Crea una función que reciba una tupla, una posición y un elemento. Esta función deberá devolver una tupla igual que la recibida por parámetros que incorpore en la posición dada el elemento dado por parámetros. En caso que la posición dada sea superior a la longitud de la tupla, se insertará al final de la tupla.
- ❸ Crea una procedimiento que reciba dos tuplas y muestre el número de elementos distintos entre ellas.



# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 4 Escribir un programa que almacene en una tupla las asignaturas de un curso de GISAM y en otra tupla las notas de la asignatura. El programa deberá mostrar por pantalla el nombre de la asignatura así como la nota obtenida.
- 5 Crea un programa donde se tenga una estructura de datos de tipo tupla que contenga los meses del año. El programa deberá pedir reiterativamente un número por pantalla y mostrar el equivalente al mes. En caso que el número no sea un valor válido finalizará la ejecución del programa.
- 6 Crea un programa que haga uso de una tupla con números ya definidos (tan larga como se quiera). Se deberá pedir al usuario un número y se retornará el número de veces que aparece en la tupla.

# Definición de lista

- Una lista es una secuencia **mutable** e **indexable** de elementos.
- Sintaxis:

```
lista1 = [] # lista vacía
lista2 = [1,2,3,4,5,6] # lista de números
lista3 = ["a","b","c","d"] # lista de strings
lista4 = ["a",2.9,False, 8] # lista mixta
lista5 = [[1,2,3],[4,5,6],[7,8,9]] # lista de listas
```

# Operaciones con listas

- Propiedad de **indexación**: Los elementos se obtienen por su **índice**

```
print (lista5 [0][2]) # >>> 3
```

```
print (lista4 [-3])# >>> 2.9
```

- Propiedad **mutable**: Los elementos **pueden** ser modificados

```
lista2 [0] = 2 # >>> [2, 2, 3, 4, 5, 6]
```

```
lista2 [2] = lista2 [0]+2 # >>> [2, 2, 4, 4, 5, 6]
```

- Para iterar una lista se puede utilizar un bucle for:

```
listaCompra = [ 'agua', 'tomates', 'peras', 'cafe' ]
```

```
print (f"□No□olvides□comprar:□", end="")
```

```
for i in listaCompra:
```

```
    print (i, end="□")
```

# Operaciones con listas

- Se pueden usar los operadores: **+**, **\***, **==**, **in**, **not in**

```
l1 = [1, 2, 3]
l2 = [4, 5, 6]
print (l1 + l2) # >>> [1, 2, 3, 4, 5, 6]
print (l1 * 2) # >>> [1, 2, 3, 1, 2, 3]
print (2 in l1) # >>> True
print (4 not in l2) # >>> False
print (l1 == l2) # >>> False
```

# Operaciones con listas

- También se puede obtener una sub-lista usando `l[i:j:k]`. Donde `i` y `j` son los límites de los índices de la lista y el parámetro `k` es el salto.

```
l3 = [1, 2, 3, 4, 5, 6]
print (l3[:3]) # >>> [1, 2, 3]
print (l3[4:]) # >>> [5, 6]
print (l3[::-1]) # >>> [6, 5, 4, 3, 2, 1]
print (l3[1:3]) # >>> [2, 3]
print (l3[1::2]) # >>> [2, 4, 6]
```

- También hay funciones para operar las listas y ver el valor máximo, mínimo, su suma y la longitud: `len()`, `sum()`, `min()` y `max()`.

```
l3 = [1, 2, 3, 4, 5, 6]
print (min(l3)) # >>> 1
print (len(l3)) # >>> 6
print (max(l3)) # >>> 6
print (sum(l3)) # >>> 21
```

# Funciones avanzadas de listas y tuplas

- Como hemos visto anteriormente, un TAD permite el acceso a ciertas funciones que hacen uso de una estructura de datos. Existen algunas funciones que hacen uso de las estructuras de datos y permiten modificarlas
- Estas funciones son:
  - **count(x)**: Esta función permite saber cuántas veces está contenido el elemento x en la lista / tupla.
  - **index(x)**: Retorna el identificador de la primera ocurrencia de x en la lista / tupla.

```
t1 = (1, 2, 3, 4, 2, 6)
l1 = [1, 2, 3, 4, 2, 6]
print (t1.count(2)) # >>> 2
print (l1.count(2)) # >>> 2
print (t1.index(4)) # >>> 3
print (l1.index(6)) # >>> 5
```

# Funciones avanzadas de listas

- Las listas al ser estructuras de datos mutables, se pueden modificar con las siguientes funciones:
  - **lista.append(x)**: Esta función agrega *x* al final de la lista.
  - **lista.extend(lista2)**: Agrega los elementos de *lista2* en *lista1*.
- Hay que tener cuidado con usar `lista.append(lista2)` ya que añadiríamos la lista2 completa a lista.

```
l1 = [1, 2, 3]
```

```
l2 = [4, 5, 6]
```

```
l1.append(4)
```

```
print (l1) # >>> [1, 2, 3, 4]
```

```
l1.extend(l2)
```

```
print (l1) # >>> [1, 2, 3, 4, 4, 5, 6]
```

```
l1.append(l2)
```

```
print (l1) # >>> [1, 2, 3, 4, 4, 5, 6, [4, 5, 6]]
```

# Funciones avanzadas de listas

- También existen funciones para inserción y eliminación de la lista:
  - **lista.insert(i,x)**: Esta función agrega  $x$  en  $l[i]$ . Si  $i$  es mayor o igual a la longitud de la lista, añade el elemento  $x$  al final de la lista.
  - **lista.remove(x)**: Elimina de la lista la primera ocurrencia de  $x$ . Si no existe  $x$  se produce error.
  - **del lista[i]**: Elimina el elemento  $i$  – *esimo* de la lista. Si no existe ese elemento, se devuelve error.

```
l1 = [1, 2, 3]
```

```
l1.insert(4, 2)
print (l1) # >>> [1, 2, 3, 2]
```

```
l1.remove(2)
print (l1) # >>> [1, 3, 2]
```

```
del l1[2]
print (l1) # >>> [1, 3]
```



# Funciones avanzadas de listas

- También existen funciones para la ordenación y colocación de los elementos en la lista:
  - **lista.sort()**: Ordena los valores de la lista. Por defecto la ordenación es ascendente aunque se puede definir descendente como `sort(reverse=True)`
  - **lista.reverse()**: Invierte los elementos de una lista

```
l = [2.3, 5.3, 5, 2, 3]
l.sort()
print (l) # >>> [2, 2.3, 3, 5, 5.3]
```

```
marcas = ['Ford', 'Mitsubishi', 'BMW', 'VW']
marcas.sort(reverse=True)
print (marcas)
# >>> ['VW', 'Mitsubishi', 'Ford', 'BMW']
```

```
l.reverse()
print (l) # >>> [5.3, 5, 3, 2.3, 2]
```

# Funciones avanzadas de tuplas

## Packing y Unpacking

- **Packing:** Esta técnica permite hacer uso de las funciones de las tuplas para agrupar variables (o valores) en una tupla
- **Unpacking:** Permite descomponer una tupla en diferentes variables.

*#Packing*

```
nombre = "David"
```

```
apellido = "Cortes"
```

```
edad = 18
```

```
usuario = nombre, apellido, edad
```

# Funciones avanzadas de tuplas

## Packing y Unpacking

*#Definición de las coordenadas*

`Coordenadas = 3, 5, 8`

*#Unpacking*

`x, y, z = Coordenadas`

**¿Recordáis en el tema anterior que una función podía devolver múltiples valores? → En realidad no es así, sólo se puede devolver un único valor pero se utiliza esta técnica. Ejemplo:**

```
def cuadradoMitad (numero:int):  
    return numero**2, numero/2 #Packing
```

```
cuadrado, mitad = cuadradoMitad(num)#Unpacking
```

# ¿Qué debo usar Tuplas o Listas?

- Como habéis visto, las listas y las tuplas tienen una estructura muy similar y una forma de trabajar también parecida. La **principal diferencia es la mutabilidad de la estructura de datos**. Mientras que la tupla es inmutable, es decir no se puede modificar los valores, la lista puede ser modificada.
- El principal uso de las tuplas es para **agrupar datos que de por sí deberían ir juntos**. Por ejemplo: Datos personales, posiciones de los puntos de una figura geométrica, fechas de un calendario, etc...
- De forma que podemos decir que la información almacenada en una tupla, son **datos heterogéneos pero con semántica**.

# ¿Qué debo usar Tuplas o Listas?

- El principal uso de las listas es para agrupar una **secuencia homogénea datos** cualesquiera. Por ejemplo: Lista de la compra, lista de puntos definidos como tuplas, etc...
- De hecho el ser mutable conlleva que las listas se deban almacenar en dos bloques de memoria, mientras que las tuplas sólo se utiliza un bloque para almacenarse.
- Esto también conlleva que las tuplas sean más rápidas de manejarse que las listas.

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 1 Escribir una función que pida un conjunto de números al usuario y los inserte en una lista hasta que el usuario introduzca un 0.
- 2 Escribir una función que la lista del ejercicio anterior la ordene de mayor a menor.
- 3 Crear una función que introduzca los números del 1 al 1000 en una lista.
- 4 Crear una función que pida 10 números por teclado, los almacene en una lista y calcule su media y su suma.

# Ejercicios

## **Ejercicios - Implementa los ejercicios en un script de Python.**

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 5 Realiza un programa para mantener una lista de compras del supermercado. Deberá mostrar por pantalla un menú en el que se permitan las diferentes opciones del programa. Las opciones serán: agregar productos a la lista, eliminar productos, ver la lista y ordenarla alfabéticamente y salir del programa.

# Tema 5: Estructuras de Datos II



# Introducción

- Hasta ahora hemos creado estructuras de datos simples (Listas y tuplas) donde se almacena un dato/elemento por cada una de las posiciones de la estructura.
- Pregunta: Si quisiéramos almacenar los datos relacionados de una persona de la clase con su nota final, ¿cómo lo haríais?

# Introducción

- Hasta ahora hemos creado estructuras de datos simples (Listas y tuplas) donde se almacena un dato/elemento por cada una de las posiciones de la estructura.
- Pregunta: Si quisiéramos almacenar los datos relacionados de una persona de la clase con su nota final, ¿cómo lo haríais?
- Quizás una de las ideas que podáis tener es usar una **lista**:
  - Una de las formas instintivas que tenemos es utilizar listas separadas para cada elemento.
  - Todas las listas deben tener la misma longitud.
  - Para que las listas tengan información coherente, deben almacenar en la misma posición del índice el valor relacionado para una persona.

# Introducción

## Ejemplo:

```
dni=[ '49121395L ', '94091571C ', '06960661X ',\
      '92289689G ' ]
nombre=[ 'David ', 'Dunia ', 'Pedro ', 'Gabriela ' ]
dni=[ '0.5 ', '10 ', '8 ', '7 ' ]
```

**Ejercicio:** Partiendo de una estructura de listas, crear una función que busque y devuelva la información de uno de los alumnos. **La búsqueda la tendréis que realizar por el valor que consideréis más apropiado.**

# Introducción

## Desarrollo:

- Es bastante complicado gestionar toda la información ya que hay que mantener múltiples estructuras con diferentes tipos.
- Se deben tener tantas listas como datos y pasarlas como argumentos.
- Se debe indexar siempre con enteros las listas.
- Si se produce algún cambio, se debe recordar cambiar varias listas

**Por lo tanto: Necesitamos otra estructura que nos permita mantener una estructura que tengamos un índice general que no sea siempre entero y un valor que se almacene.**

# Introducción

## Diccionarios vs Listas

### Listas

|     |        |
|-----|--------|
| 0   | Elem 1 |
| 1   | Elem 2 |
| 2   | Elem 3 |
| 3   | Elem 4 |
| ... | ...    |

Índice

Dato

### Diccionarios

|         |         |
|---------|---------|
| Clave 1 | Valor 1 |
| Clave 2 | Valor 2 |
| Clave 3 | Valor 3 |
| Clave 4 | Valor 4 |
| ...     | ...     |

Índice generado por  
una etiqueta

Dato

# Definición de diccionario

- Un diccionario es una estructura **mutable** de datos sin orden entre los elementos, el acceso a los mismos se realiza a través de la clave.
- Normalmente la clave será una cadena de texto, aunque puede ser otro tipo de datos inmutable de Python.
- Sintaxis:

```
dict1 = {} # diccionario vacío
dict2 = {'49121395L': 'David', '94091571C': 'Dunia', \
        '06960661X': 'Pedro', '92289689G': 'Gabriela'}
#Ejemplo anterior
dict3 = {'nombre': 'David', 'despacho': 101, \
        'email': 'david.cortes.polo@urjc.es'}
#otro ejemplo
dict4 = {'nombre_completo': \
        {'nombre': 'David', 'apellidos': 'Cortés_Polo'}}
# diccionarios anidados
```

# Operaciones con diccionarios

- Propiedad de **indexación**: Los elementos se obtienen por su **clave**

```
print(dict3['despacho'])# >>> 101
print(dict4['nombre_completo']['nombre'])
# >>> David
```

- Propiedad **mutable**: Los elementos **pueden** ser modificados

```
dict3['despacho'] += 1
# >>> {'nombre': 'David', 'despacho': 102, \
# 'email': 'david.cortes.polo@urjc.es'}
dict3['despacho'] = "Departamental_3_Desp._101"
# >>> {'nombre': 'David',
# \ 'despacho': 'Departamental 3 Desp. 101', \
# 'email': 'david.cortes.polo@urjc.es'}
```

# Operaciones con diccionarios

- Hay ciertas operaciones que no modifican un diccionario
  - **len(d)**: Devuelve el número de elementos del diccionario.
  - **min(d)**: Devuelve la clave mínima del diccionario.
  - **max(d)**: Devuelve la clave máxima del diccionario.
  - **sum(d)**: Devuelve la suma de las claves del diccionario siempre que se puedan sumar.
  - **d.get(clave)**: Devuelve el valor del diccionario asociado a la clave.

```
d1 = {1: 'David ', 2: 'Felipe ', 3: 'Ana '}  
print (min(d1)) # >>> 1  
print (max(d1)) # >>> 3  
print (len(d1)) # >>> 3  
print (sum(d1)) # >>> 6  
print (d1.get(2)) # >>> Felipe
```



# Operaciones con diccionarios

- También podemos gestionar la información de las claves y los valores a través de los siguientes:
  - **clave in d**: Devuelve True si la clave está en el diccionario o False si no se encuentra
  - **d.keys()**: Devuelve un iterador sobre las claves del diccionario.
  - **d.values()**: Devuelve un iterador sobre los elementos del diccionario.
  - **d.items()**: Devuelve un iterador sobre los pares clave-valor del diccionario. **¡OJO - Devuelve una tupla!**

# Operaciones con diccionarios

```
'despacho' in dict3 # >>> True
```

*#Para iterar sobre todas las claves*

```
for k in dict3.keys():  
    print (k, end = "□")
```

*#Para iterar sobre todos los elementos*

```
for v in dict3.values():  
    print (v, end = "□")
```

*#Para iterar sobre los pares clave-valor*

```
for i in dict3.items():  
    print (i)
```

# Operaciones con diccionarios

- Hay ciertas operaciones que sí modifican un diccionario
  - **d[clave]=nuevoValor**: Modifica la información relacionada con la clave a nuevoValor.
  - **d.update(dict2)**: Añade los pares de dict2 al diccionario d.
  - **d.pop(clave, valorAlternativo)**: Devuelve la clave buscada del diccionario d y lo **elimina** del mismo. En caso de no encontrar esa clave devuelve valorAlternativo .
  - **d.popitem()**: Devuelve el último par clave-valor añadido al diccionario d y lo **elimina** del mismo.
  - **del d[clave]**: Elimina el par con la clave buscada del diccionario d.
  - **d.clear()**: Elimina todos los pares del diccionario d.

# Cuándo elegir entre listas, tuplas y diccionarios

- Elegiremos una **lista** cuando:
  - Los datos necesiten tener un orden.
  - Es necesario actualizar o alterar los datos durante la ejecución del programa.
  - Vamos a usar esta estructura de datos para iterar sobre ella.
- Elegiremos una **tupla** cuando:
  - Los datos no requieren mantener un orden
  - No va a ser necesario actualizar o alterar los datos durante la ejecución del programa.
  - Vamos a usar esta estructura de datos para iterar sobre ella.
- Elegiremos un **diccionario** cuando:
  - Los datos no están ordenados o el orden no importa.
  - Puede ser necesario actualizar o alterar los datos durante el programa.
  - Vamos a usar esta estructura principalmente para organizar y estructurar datos y buscar valores.

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 1 Escribir una función que pida un conjunto de números al usuario, los inserte como clave de un diccionario y como valor inserte el cuadrado del número hasta que el usuario introduzca un 0. Por último deberá mostrar todos y dejar el diccionario vacío.
- 2 Crear un programa que se ejecutará en las balanzas de nuestro comercio de confianza. Una vez ejecutada la aplicación el dependiente guardará un conjunto de frutas y su precio por cada kilo de producto hasta que se escriba fin. Una vez insertada toda la lista de precios, la aplicación pedirá al cliente cada una de las frutas que lleva en su cesta y su cantidad en kilos y el programa devolverá el precio que deberá pagar en caja por cada fruta.

# Ejercicios

- 3 Vamos a crear un programa que sea un traductor automático de palabras en inglés. Para ello deberéis crear una función que se encargue de rellenar un diccionario que contenga diferentes palabras en español y su traducción en inglés hasta que se escriba **FIN**. Una vez hecho esto, comenzaréis a escribir palabras en español y la aplicación deberá traducirlas al inglés. En caso que no se encuentre deberá mostrar un mensaje de error. El programa finalizará cuando el usuario escriba **END**.
- 4 Escribir un programa que vaya recibiendo una palabra cada vez y cuente el número de veces que ha ido apareciendo cada una de las palabras. Una vez se introduzca la palabra **FIN**, se deberá calcular el promedio de aparición de cada una de las palabras en la ejecución del programa.

# Ejercicios

- 5 Escribir un programa que implemente una agenda de contactos. El programa tendrá el siguiente menú:
- Añadir - Modificar: El programa pedirá el nombre de la persona. Si el nombre está en la agenda, debe mostrar el teléfono y pedir al usuario si quiere elegir modificar su número.
  - Borrar: Nos pide un nombre y si existe en la agenda, tras una confirmación lo borrará de la misma.
  - Listar: Nos mostrará todos los contactos de la agenda.
  - Salir: Finalizará el programa.

# Definición de un string

- Un String es el tipo que define en Python una cadena de caracteres.
- Es un tipo básico y, al igual que en los enteros y los float, es no mutable.
- Como ya hemos visto en clase, se definen entre comillas simples o dobles.
- Sintaxis:

```
s1 = "String_1"
```

```
s2 = 'String_2'
```

**Qué pasará en la siguiente frase ¿funcionará o no cuando se compile el código?:**

```
s = "He_visto_cosas_que_vosotros_no_creeriais.\nAtacar_naves_en_llamas_mas_alla_de_Orion.\n    "He visto Rayos-C brillar en la oscuridad",\n    cerca_de_la_puerta_de_Tannhauser."
```



# Definición de un string

## Solución:

*# Usar comillas dobles para delimitar el String  
# y simples dentro del texto como carácter.*

*# O viceversa*

```
s = "He visto cosas que 'vosotros' no creíais.\nAtacar naves en llamas mas allá de Orión.\n'He visto Rayos-C brillar en la oscuridad',\n    cerca de la puerta de Tannhauser."
```

*# Usar comillas simples para delimitar el String  
# y dobles con el caracter de escape \*

*# dentro del texto como carácter. O viceversa*

```
s = 'He visto cosas que \"vosotros\" no creíais.\nAtacar naves en llamas mas allá de Orion.\n\"He visto Rayos-C brillar en la oscuridad\", \n    cerca de la puerta de Tannhauser.'
```

# Definición de un string

**La barra invertida no sólo permite poner comillas. También permite los saltos de línea, tabuladores, etc.. en un String:**

| Secuencia | Significado     |
|-----------|-----------------|
| \"        | Comilla doble   |
| \'        | Comilla simple  |
| \t        | Tabulador       |
| \n        | Salto de línea  |
| \\        | Barra invertida |

# Uso de los string

- Se puede pedir al usuario usando la función **input**("Texto").
- Un string puede ser transformado en otros formatos: Enteros **int**( variable ), Reales **float**( variable ), Complejos **complex**(variable), Booleanos **bool**( variable )
- Una característica de los String es que se puede obtener el valor entero (código ASCII) de cada uno de los caracteres que conforman el String, y transformar un valor ASCII en un carácter:
  - **ord**( variable ): Recibe un carácter y devuelve el valor ASCII.
  - **chr**( variable ): Recibe un valor ASCII y retorna un carácter.
- El recorrido de un String se realizará de la misma forma que se hace en **las listas**.

**Ejercicio:** Realiza una aplicación que pida una cadena de caracteres por teclado y muestre uno a uno los caracteres del String y su valor en decimal.

# Uso de los string

**En Python se permite formatear un string a diferentes formatos:**

`"%formato_%variable"`

| Secuencia | Formato                              |
|-----------|--------------------------------------|
| %d        | Decimal                              |
| %o        | Octal                                |
| %X        | Hexadecimal                          |
| %E        | Notación Científica                  |
| %f        | Coma Flotante                        |
| 0.n%f     | Coma Flotante con <i>n</i> decimales |

**Ejercicio:** Escribir el número **PI:3.141526535** como: decimal, número científico, float y float con 3 decimales.

# Operadores de los string

- Se pueden usar los operadores: **+**, **\***, **!=**, **==**, **in**, **not in**, **≤**, **<**, **>**, **≥**

```
c1 = "Cadena_1"
c2 = "_Cadena_2"
print (c1 + c2) # >>> "Cadena 1 Cadena 2"
print (c1 *2) # >>> "Cadena 1 Cadena 1"
print ('2' in c1) # >>> False
print ('n' not in c2) # >>> False
print (c1 == c2) # >>> False
print ("E" < "a") # >>> True
# El orden es primero mayusculas y
# luego minusculas siguiendo el codigo ASCII.
```

# Operaciones con los string

- Para obtener una sub-cadena de un String, se puede usar la siguiente estructura:

```
cadena = "Hola_que_tal"
print (cadena[:3]) # >>> Hol
print (cadena[4:]) # >>> "que tal"
print (cadena[::-1]) # >>> "lat euq aloH"
```

- Existen funciones para analizar el tamaño del string: **len(string)**.
- Para contar el número de ocurrencias de un caracter en el String se usa: **s.count(caracter)**.

```
cadena = "Hola_que_tal"
print (len(cadena)) # >>> 12
print (cadena.count('a')) # >>> 2
```

**Ejercicio:** Escribe un programa que invierta un string carácter a carácter sin usar el modificador `::-1`. Podrás usar el resto de operaciones que hemos visto hasta ahora.

# Funciones de los string

- **s.find(caracter):** Retorna el índice de la primera ocurrencia del carácter en la cadena de caracteres. Si no lo encuentra devuelve -1.
- **s.rfind(caracter):** Igual que el anterior pero de derecha a izquierda.
- **s.lower():** Devuelve el String s con todos los caracteres en minúscula.
- **s.upper():** Devuelve el String s con todos los caracteres en mayúscula.
- **s.capitalize():** Devuelve el String s con la primera letra en mayúscula.

```
s="aYUDAME_y_tirame_esa_CUERDA"
```

```
s.find('U') #>>> 2
```

```
s.rfind('U') #>>> 22
```

```
s.lower() #>>> 'ayudame y tirame esa cuerda'
```

```
s.upper() #>>> 'AYUDAME Y TIRAME ESA CUERDA'
```

```
s.capitalize() #>>> 'Ayudame y tirame esa cuerda'
```

# Funciones de los string

- **s.split(caracter)**: Divide el String s según las ocurrencias del caracter (token) y retorna las partes en una lista.
- **caracter.join(lista)**: Une una lista mediante el caracter y retorna el String resultante.
- **s.replace(c1, c2)**: Retorna un string igual a s pero cambiando cada ocurrencia del carácter c1 por c2.

```
s="pan#patatas#fresas#cerezas"
lista = s.split('#')
print (lista) #>>> ['pan', 'patatas', 'fresas', \
                    # 'cerezas']
print (lista[2]) #>>> fresas
s2 = "".join(lista)
print (s2) #>>> pan=patatas=fresas=cerezas
print (s2.replace("=fresas=", "#Maiz#"))
#>>> pan=patatas#Maiz#cerezas
```



# Funciones de los string

- **s.strip(caracter)**: Devuelve el String s sin el carácter a ambos lados del String.
- **s.lstrip(caracter)**: Devuelve el String s sin el carácter a la izquierda.
- **s.rstrip(caracter)**: Devuelve el String s sin el carácter a la derecha.
- **s.startswith(caracter)**: Devuelve True si s comienza con el carácter.
- **s.endswith(caracter)**: Devuelve True si s finaliza con el carácter.
- **s.isalpha()**: Devuelve True si s sólo tiene letras.
- **s.isdigit()**: Devuelve True si s sólo tiene números.

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 1 Crea un método que reciba una cadena y devuelva True si puede ser convertido a float.
- 2 Crea un método que reciba una cadena y devuelva True si el String es un correo electrónico.
- 3 Crea un reconocedor de palíndromos que no considere ni espacios ni signos de puntuación.
- 4 Implementa una función **mayor(s1, s2)** de forma que retorne True si s1 es mayor o igual que s2.

# Gestión de la memoria - Recordatorio de conceptos

- En el tema 3 vimos los pasos de parámetros por valor y por referencia en las funciones, siendo estos:
  - **El paso por valor:** Copia los valores de la variable.
  - **El paso por referencia:** vincula las variables a través de sus direcciones de memoria.
- Además, también estudiamos las diferentes de los tipos de datos mutables e inmutable, es decir, que se pueden modificar
- La relación del paso por valor y por referencia y entre los tipos de datos mutables e inmutables está íntimamente relacionado con el funcionamiento de la gestión de memoria en Python.

# Gestión de la memoria - Variables no mutables

- Las variables no mutables se comportan de manera similar al paso por valor, la gestión de las variables se hace por copia de su contenido.
- A través de la llamada **id( variable )** se puede obtener la dirección de memoria de la variable en formato entero (para transformarla en hexadecimal usamos **hex(id( variable ))**).
- Veamos un ejemplo de cómo se comporta Python a la hora de operar con variables no mutables.

```
str = "Prueba_de_cadenas_"
print(str)
print(hex(id(str))) #>>> 0x10107b170
str += "_y_su_concatenacion"
print(str)
print(hex(id(str))) #>>> 0x100fee960
```

# Gestión de la memoria - Variables no mutables

- Como podéis observar, las direcciones de memoria que se muestran son diferentes ya que las cadenas (como los int, float, etc..) son no mutables, de forma que al realizar una operación sobre ellos, se crea una nueva variable donde se almacena el valor nuevo del string creado como la unión de los dos anteriores.
- La dirección de memoria anterior deja de ser útil y la variable str apunta a una nueva dirección de memoria donde está el contenido.
- Esto mismo sería lo que pasaría con tipos de datos más complejos no mutables como las tuplas.

```
fecha= (25, "Mayo", 1915)
hex(id(fecha)) # >>> '0x100ed2970 '
print (fecha)
fecha= (25, "Mayo", 1915, "Martes")
hex(id(fecha)) # >>> '0x1010eae00 '
print (fecha)
```

# Gestión de la memoria - Variables no mutables

- Par hacerse cargo de la memoria que ha quedado con información pero que ya no es útil debido a que el identificador de la variable ha podido cambiar, Python implmenta un **recolector de basura (Garbage Collector)**.
- Este recolector realiza un reciclado de los objetos en memoria para optimizar el espacio ocupado.
- Cada uno de los objetos creados dinámicamente (por ejemplo con una asignación) son designados automáticamente. El recolector se hará cargo de ellos cuando no existan referencias a un objeto.
- Al no existir referencias a ese espacio de memoria se sobreentiende que no es necesario y que la memoria ocupada por ese objeto se puede reclamar.

# Gestión de la memoria - Variables mutables

- Las variables mutables son variables más cercanas al modo de funcionamiento del paso por referencia, de forma que se permite incorporar nuevos elementos a la estructura de datos.

```
fecha= [25, "Mayo", 1915]
hex(id(fecha)) # >>> '0x10107c580'
print (fecha)
fecha.append("Martes")
hex(id(fecha)) # >>> '0x10107c580'
print (fecha)
```

**¿Qué es lo que ha pasado? ¿Puedes buscarle una explicación y relacionarlo con el caso anterior?**

# Gestión de la memoria - Copia de las variables mutables

- Cuando se quiere usar una estructura mutable en diferentes partes del código, se puede usar una copia de la información de múltiples maneras:
  - **Copia por referencia  $\text{var1} = \text{var2}$ :** Se asocia a la variable `var1` la misma dirección de memoria de `var2`. Es decir, ambas variables apuntan a la misma dirección de memoria. Cualquier cambio que hagamos a través de `var1` o `var2` afectará a la misma estructura de datos.
  - **Copia por valor  $\text{var1} = \text{NombreEstructura}(\text{var2})$ :** Crea una copia de la estructura de datos asociada a `var2` en una dirección de memoria diferente y se la asocia a `var1`. Las variables apuntan a direcciones de memoria diferentes que contienen los mismos datos. Cualquier cambio que hagamos a través de `var1` no afectará al estructura de datos de `var2` y viceversa.



# Gestión de la memoria - Copia de las variables mutables

Ejemplo con diccionarios

```
# copia por referencia
```

```
a={1: 'A', 2: 'B', 3: 'C'}
```

```
b = a
```

```
print (b) #>>> {1: 'A', 2: 'B', 3: 'C'}
```

```
b.pop(2)
```

```
print (b) #>>> {1: 'A', 3: 'C'}
```

```
print (a) #>>> {1: 'A', 3: 'C'}
```

# Combinación de diferentes estructuras anidadas

- Hasta ahora hemos visto las **listas, tuplas y diccionarios** como estructuras de datos fundamentales.
- Una de las grandes ventajas de Python es que podemos anidar estructuras de datos dentro de otras estructuras.
- Al comienzo de este tema ya vimos que se podía anidar un diccionario dentro de otro diccionario. Esto mismo es aplicable a las listas y las tuplas, de forma que podemos crear estructuras más complejas.
- De esta forma podríamos tener:
  - Listas de listas → Matrices
  - Listas con tuplas o diccionarios
  - Diccionarios de diccionarios
  - Diccionarios que contengan listas o tuplas
  - Tuplas que contengan otras tuplas.
  - Tuplas que contengan diccionarios o listas.
- Para poder acceder a cada una de las estructuras, debemos conocer cómo acceder a la estructura original, ya sea lista, tupla o diccionario y seguidamente a la segunda estructura. Ejemplo:

# Combinación de diferentes estructuras anidadas

## Ejemplo

```
diccionario_lista = {}  
diccionario_lista ['Clave1']=[10, 100, 1000]  
print (diccionario_lista ['Clave1'] [1]) #>>> 100  
#Accedemos en el primer corchete a la clave del\  
# diccionario y en el segundo corchete a la\  
# posicion 1 de la lista.
```

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- ❶ Crea un programa que permita cargar un código de producto como clave de un diccionario. Deberás guardar para cada clave de producto el nombre del mismo, su precio, y cantidad de stock. Deberás implementar las siguientes funcionalidades:
  - Insertar los datos en el programa
  - Listar todos los productos
  - Consultar un producto por su clave, mostrando su nombre, precio y stock.
  - Listar todos los productos que tengan un stock de valor cero.
- ❷ Crea una agenda. Esta agenda deberá usar un diccionario cuya clave sea la fecha. Debe permitir almacenar diferentes actividades para un mismo día ( hora y actividad). La agenda deber implementar las siguientes funciones:
  - Insertar datos en la agenda.
  - Listar todos los apuntes de la agenda.
  - Consultar la agenda por fecha.

# Ejercicios

- 3 Escribir una lista de 10 tuplas que guarden el nombre de un país y el número de habitantes. Deberéis crear tres funciones, la primera para insertar los datos, una función de impresión y, por último, una función para que muestre el nombre del país con mayor número de habitantes.
- 4 Supongamos que tenemos una lista anidada donde cada lista interna contiene cadenas que representan los síntomas exhibidos por el paciente correspondiente. Queremos escribir una función que tome esta lista como parámetro y devuelva una nueva lista que contenga números enteros. Para cada paciente, la nueva lista debe contener el número de síntomas que presentaban. Ejemplo:
  - `sintomas([['fatiga', 'hinchazón abdominal', 'hematomas'], ['pérdida de apetito', 'fatiga']])`
  - La salida deberá ser `[3,2]`

# Tema 6: Gestión de Ficheros

# Introducción

- En los anteriores temas hemos ido viendo cómo interactuar con las aplicaciones a través del teclado y y la salida estándar. Muchas veces, debido a la automatización de la información es necesario cargar una buena cantidad de información y éstas opciones no son válidas a la hora de tratar mucha información.
- Es por esto que se utiliza los ficheros para cargar y almacenar grandes cantidades de información que serán utilizados por nuestras aplicaciones.
- Las operaciones más comunes que se realizan con los ficheros son:
  - Crear un fichero.
  - Escribir en un fichero.
  - Borrar fichero.
  - Leer datos de un fichero.
  - **referencia**  
<https://docs.python.org/3/library/functions.html#open>

# Apertura de fichero

- Para crear un fichero se ha de utilizar la siguiente función:
  - **f=open(ruta, 'w')**: Esta instrucción crea un fichero en la **ruta** designada, lo abre en modo solo escritura de texto (esto se define por el argumento **'w'**) y devuelve la referencia del fichero en **f**
  - Es importante aclarar que si el fichero no existe en esa ruta se crea, pero si existiera alguno con el mismo nombre en la ruta **se sobrescribe**.
  - Si en vez de usar el modo texto queremos almacenar información binaria se utilizará el modificador **'wb'**.
- También se pueden incluir datos a un fichero ya creado **sin sobrescribir el contenido**. Para ello se debe abrir el fichero de la siguiente forma:
  - **f=open(ruta, 'a')**: Esta instrucción abre el fichero que se encuentra en la **ruta** designada, lo abre en modo *append* (esto se define por el argumento **'a'**) y devuelve la referencia del fichero en **f**.
  - Si en vez de usar el modo texto queremos seguir escribiendo información binaria en un fichero con modo *append* se utilizará el modificador **'ab'**.



# Escritura de un fichero

- Una vez que el fichero ha sido creado, se puede comenzar a escribir datos en él. Para ello se utiliza el siguiente método:
  - `numCar = f.write(cadena)`: Con esta instrucción se escribe en el fichero referenciado por **f**, la **cadena** que se le pasa por parámetros a la función `write`. Esta función devolverá el número de caracteres escrito en la variable **numCar**.

## Ejemplo creación fichero:

```
f = open ( 'holamundo.txt' , 'w' )  
f.write ( 'Hola_Mundo!' )
```

## Ejemplo insercción nueva información

```
f = open ( 'holamundo.txt' , 'a' )  
f.write ( '\nAdios_Mundo!' )
```

# Lectura de un fichero

- Para realizar la lectura de un fichero necesitaremos abrir el mismo en modo lectura para ello usaremos la siguiente estructura:
  - `f=open(ruta, 'r')`: Esta instrucción crea un fichero en la **ruta** designada, lo abre en modo solo lectura de texto (esto se define por el argumento **'r'**) y devuelve la referencia del fichero en **f**. Si el fichero no existe en esa ruta dará un error ya que no puede leerlo.
  - Si en vez de usar el modo texto queremos leer información binaria se utilizará el modificador **'rb'**.
  - Si queremos abrir el fichero en modo lectura-escritura se utilizará el modificador **'r+'** para ficheros de texto y **'r+b'** para ficheros binarios.
- Para la lectura de la información una vez abierto el fichero se usarán las siguientes funciones:
  - `lectura = f.read()`: Lee todos los datos del fichero como una cadena de caracteres y lo devuelve en la variable **lectura**.
  - `lectura = f.read(longitud)`: Lee del fichero la cantidad de información definida por **longitud** y lo devuelve en la variable **lectura**.
  - `lectura = f.readlines()`: Devuelve una lista donde cada elemento de la misma es una de las línea del fichero. La lista es devuelta en la variable **lectura**.

# Cierre de fichero

- Una vez que se ha finalizado de trabajar con el fichero es necesario cerrarlo para que se liberen todos los recursos reservados:
  - `f.close()`: Esta instrucción cierra el fichero referenciado por `f`.
- Esto es principalmente importante a la hora de trabajar con el modo escritura ya que el fichero está bloqueado por el programa que lo abrió en modo escritura y **hasta que no se cierre no se liberará**.
- En caso de no cerrarse, Python intentará cerrar el fichero cuando considere que no se va a usar más.

## Ejemplo

```
f = open('holamundo.txt', r)
print(f.read(5))
print(f.read())
f.close()
print(f.read()) #Esta ultima lectura producira
                 #un error
```

# Cierre de fichero

- Para evitar tener que estar pendiente del cierre del fichero Python introduce una estructura que permite crear un bloque de código que manejará el fichero:

```
with open (ruta , modo) as f:  
    bloque de codigo
```

- De esta forma se abrirá un fichero en la **ruta** definida, en el **modo** deseado (escritura 'w', lectura 'r' o lectura-escritura 'r+', etc...) y devuelve la referencia del fichero como **f**.
- El fichero permanecerá abierto mientras que el bloque esté abierto y lo cerrará una vez finalice el bloque.

# Otras herramientas del fichero

- Para renombrar o borrar un fichero se utilizan las siguientes funciones:
  - `os.rename (ruta1,ruta2)`: renombra y mueve el fichero de la **ruta1** a la **ruta2**.
  - `os.remove (ruta)`: Elimina el fichero de la **ruta**.
  - `os.path. isfile (ruta)`: Comprueba que el fichero existe en la **ruta** (Devuelve True en caso afirmativo).
  - Para usar cualquiera de estas funciones hay que incluir la librería `os`

## Ejemplo:

```
import os
```

```
f = 'holamundo.txt'
if os.path.isfile(f):
    os.rename (f, 'holapython.txt')
else:
    print(f"El fichero {f} no existe")
```

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- 1 Escribir un programa que pida el nombre de un archivo y un número N e imprima las primeras N líneas del archivo.
- 2 Escribir una función que pida un número entre el 1 y el 10 y guarde en el fichero con el nombre "tabladel-n.txt" la tabla de multiplicar. n será el número introducido.
- 3 Escribir una función que pida un número entre el 1 y el 10, compruebe si la tabla existe (ver ejercicio anterior para ejemplo del nombre) y la muestre por pantalla. Si el fichero no existe deberá mostrar un mensaje por pantalla.

# Ejercicios

- 4 Escribir un programa que pida el nombre de un archivo, lo procese e imprima por pantalla cuántas líneas, cuantas palabras y cuántos caracteres contiene el archivo.
- 5 Escribir un programa que pida un fichero y una palabra e imprima las líneas del archivo que contienen la palabra introducida.

# Tema 7: Programación orientada a objetos

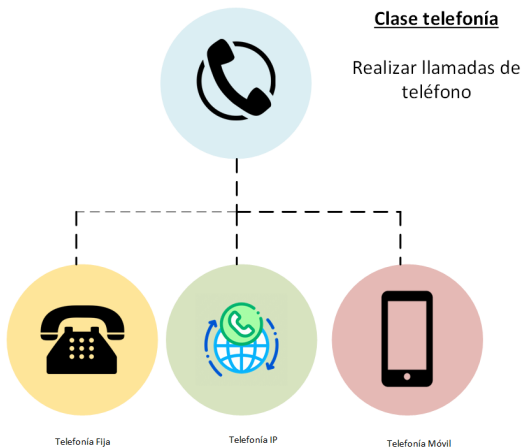


# Conceptos Básicos

- Primera aparición: Simula, de O. J. Dahl et al., 1967  
Objetivo: favorecer el diseño de sistemas software complejos.
- **¿Qué es un objeto?**: Entidad que encapsula los servicios que ofrece a través de una interfaz de paso de mensaje.
- **Características**:
  - El estado está protegido.
  - Los métodos comparten el estado.
  - Acceso al estado desde el exterior mediante los métodos.
- **Conceptos clave**: autonomía e independencia.
- Para marcar el grado de independencia se habla en términos de paso de mensajes. Ejemplos de paso de mensaje en los diferentes lenguajes de programación:
  - En los lenguajes secuenciales: metáfora que expresa interacción entre objetos.
  - En los lenguajes concurrentes: paso de mensaje real como medio de comunicación.

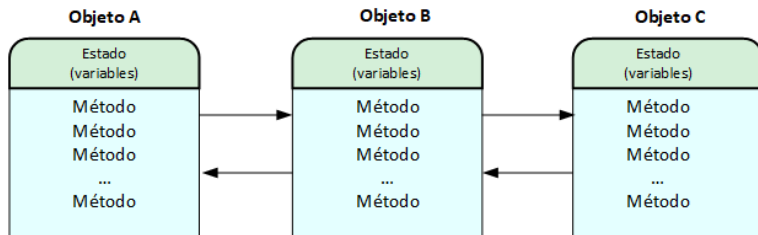
# Conceptos Básicos

## Ejemplo:



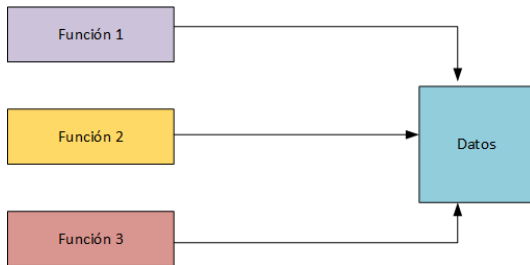
# Conceptos Básicos

## Ejemplo de paso de mensajes entre objetos:



# Conceptos Básicos

## Paradigma Tradicional



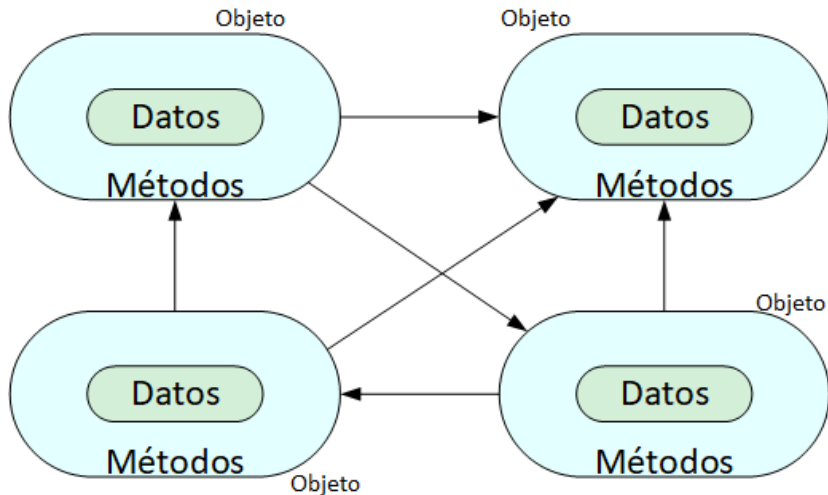
## Causas del auge de los Lenguajes Orientados a Objetos

Entidad que encapsula los servicios que ofrece a través de una interfaz de paso de mensaje.

- 1 Construcción de un sistema software complejo → división del sistema en módulos independientes que ofrecen un servicio.
- 2 El paradigma orientado a objetos se adapta de forma natural, porque ofrece un criterio estricto de **modularización**.

# Conceptos Básicos

## Paradigma Orientado a Objetos



# Principios de la POO

## La POO como metodología de programación

- Esta metodología basada en la idea natural de la existencia de un mundo lleno de objetos.
- La resolución de problemas se realiza en términos de objetos.

Los principios en los que se basan son los siguientes:

- Mensajes y métodos
  - Los mensajes inician acciones
  - Los mensajes se envían a objetos
  - Un objeto recibe mensajes y ejecuta métodos: ocultación de información

# Principios de la POO

## Diferencias entre metodología POO y estructurada

- Diferencias entre el paso de mensaje y las llamadas a procedimiento:
  - En el receptor. Está bien definido en el paso de mensaje, mientras que en las llamadas a procedimiento el receptor puede no estar bien definido.
  - Interpretación: ligadura estática y ligadura dinámica
    - Una ligadura es el momento en el que se resuelven las llamadas a métodos y la referencia de los datos.
    - En los lenguajes tradicionales la ligadura se realiza en tiempo de **compilación** → Ligadura estática
    - En los LOO no siempre se podrá o convendrá que la ligadura sea estática. Cuando la ligadura se produce en tiempo de **ejecución** → Ligadura dinámica. Esto será uno de los principios en los que se basan otras herramientas de la POO como es la **Herencia** y el **Polimorfismo**.

# Principios de la POO

- Clases e instancias

- Un objeto es una instancia de clase. Equivale a una variable
- Las clases definen el protocolo del objeto. Esto es, define la misma interfaz.
- Es decir, la clase es considerada como la unidad de encapsulación

```
class VendedorAutos:
```

```
    ventasRealizadas : int = 0
```

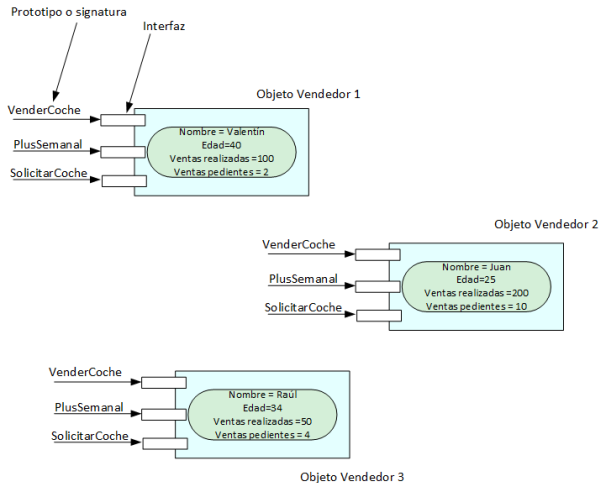
```
    def __init__(self , nombre_i: string , anios: int):  
        self.nombre = nombre_i  
        self.edad = anios
```

```
    def venderCoche (self):  
        self.__ventasRealizadas += 1
```

```
v1 = VendedorAutos("David" , 25) #instancia
```



# Principios de la POO



# Ejercicio

- Implementa la clase `Coche`, que tiene tres atributos, **velocidadActual**, **bateria** y **marchaActual**, de tipo entero y cuatro métodos `acelerar()`, `frenar()`, `cambiarMarcha (marcha:int)` y `estadoBateria()`, donde el primero dobla la velocidad actual y divide la batería a la mitad, el segundo reduce a la mitad la velocidad actual y deja la batería como está, y el tercero ajusta la marcha y cuarto muestra el estado de la batería. La clase debe tener además un constructor que inicialice todos los atributos.
- Crea dos instancias de la clase `coche`: `miCoche` y `tuCoche`.

# Ejercicios

## Ejercicios - Implementa los ejercicios en un script de Python.

Introduce los comentarios necesarios para entender cada ejercicio en el script.

- ❶ Crear una clase llamada Ciudadano. Sus atributos son: nombre, edad y DNI. Construye los siguientes métodos para la clase:
  - Un constructor, que inicialice todos los atributos.
  - Un método obtener por cada atributo que devuelva el valor que contiene (comúnmente llamados getters).
  - Un método mostrar() que muestra por pantalla los datos de la persona.
  - un método esMayorDeEdad() que devuelve un booleano si lo es.
- ❷ Crear la clase Persona con los métodos “setNombre”, “setEdad”, “getNombre”, “getEdad” y “mostrarPersona”. Luego crear dos objetos del tipo Persona e imprimirlos.

# Ejercicios

- ③ Escribir un programa que pida diferentes Ciudadanos, los introduzca en una lista y a continuación los muestre.
- ④ Reescribir el ejercicio anterior, utilizando la mayor parte del código posible, en el que la clase ciudadano ahora contendrá un único atributo que será un diccionario. Este diccionario deberá contenga la información de un ciudadano.

# Algoritmos I: Algoritmos de Aproximación

# Importancia de Algoritmos de Aproximación

- Situaciones dónde obtener una solución exacta puede ser computacionalmente costoso o imposible.
- Problemas **NP-Hard**: No hay algoritmos eficientes que encuentren la solución óptima en *tiempo polinómico*.
- En estos casos, aproximar la solución a un problema utilizando **Algoritmos de Aproximación**.
- **Algoritmos de Aproximación**: Estrategia computacional para abordar problemas de optimización, donde se busca encontrar la mejor solución posible en un tiempo razonable.

# Ejemplos: Problemas de optimización

- Este tipo de algoritmos son empleados para situaciones como las siguientes:
  - Asignar recursos limitados → Espacio en una mochila o Tiempo de entrega de tareas.
  - Planificar rutas eficientes → Minimizando la distancia recorrida o los costes.
  - Problemas de asignación de tareas → Distribuir tareas entre trabajadores.
- Generan una solución práctica y eficiente a problemas de optimización, en escenarios donde encontrar la solución exacta es muy costoso computacionalmente o impracticable.

# Concepto de Optimización

- La optimización es el proceso de lograr la mejor solución posible dadas las restricciones y recursos disponibles.
- Dos tipos principales de optimización:
  - 1 Optimización lineal: Función objetivo y restricciones lineales.
    - Problema del viajante de comercio.
  - 2 Optimización no lineal: Función objetivo o restricciones no lineales.
    - Programación cuadrática: Valores de variables que minimicen o maximicen una función cuadrática.
    - Problema de la Mochila.



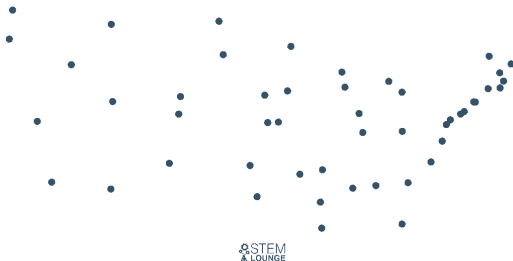
# Problemas de Decisión vs. Problemas de Optimización

- **Problemas de Decisión:** Se determina si existe al menos una solución que satisface ciertos criterios.
- **Problemas de Optimización:** No solo buscamos una solución, sino la mejor solución posible.
- En el problema del viajero... ¿Cuál sería el problema de decisión? ¿Y el de optimización?
- En el problema de la mochila... ¿Cuál sería el problema de decisión? ¿Y el de optimización?
- Los algoritmos de aproximación se centran en lograr un equilibrio entre tiempo de ejecución y calidad de la solución.
- En líneas generales, cuánto más rápido proporcione el algoritmo la solución más precisión habrá sacrificado.

# Ejemplos: Problema Viajante de Comercio.

- Alcanzar un conjunto de objetivos (lugares) regresando al punto de partida minimizando distancia y/o costes.

<https://tspvis.com/>



# Ejemplos: Problema de la mochila.

- Para un conjunto de elementos (con una tupla valor, peso), y una capacidad máxima de mochila, cual es el conjunto de elementos que maximiza la suma de los valores, asegurando que la suma de sus pesos no exceda la capacidad de la mochila.

<https://monicagranbois.com/knapsack-algorithm-visualization/>

# Búsqueda Exhaustiva como Algoritmo de Aproximación

- En clase, nos centraremos en los algoritmos de búsqueda exhaustiva para abordar problemas de optimización.
- Los algoritmos de búsqueda exhaustiva son algoritmos que consideran todas las posibles soluciones a un problema, y luego eligen la mejor solución.
- Son eficientes para problemas pequeños, pero pueden ser muy costosos computacionalmente para problemas grandes.

# Tema 8: Eficiencia del código

# Conceptos Básicos

Para medir la eficiencia de los programas se debería tener en cuenta:

- **Eficiencia de un algoritmo:** Medir el Coste/Rendimiento/Complejidad como de la cantidad de recursos necesarios para su ejecución.
- **Eficiencia Temporal:** Tiempo de ejecución de un algoritmo.
- **Eficiencia Espacial:** Cantidad de memoria usada para ejecutar un algoritmo.

Una idea rápida de cómo medir la eficiencia sería a través de **pruebas de bechmarking**

- Se ejecuta el programa con una muestra típica de los posibles datos de entrada.
- Se miden los tiempos de ejecución.
- Se realiza un análisis estadístico para inferir el rendimiento del programa. Pero:
  - Método fácil.
  - No siempre fiable.
  - Dependiente de la máquina

# Conceptos Básicos

## A través de técnicas analíticas

- Procedimientos matemáticos para determinar el coste.
- Como características:
  - Fiable.
  - Su realización puede ser difícil a efectos prácticos.
  - Independiente de la máquina.

En estas técnicas analíticas normalmente se ha de tener en cuenta los datos de entrada del algoritmo que marcan el tiempo de ejecución del mismo:

- Cálculo de los dígitos de un número  $\rightarrow$  N<sup>o</sup> de cifras del número.
- Algoritmo de ordenación  $\rightarrow$  N<sup>o</sup> de elementos que hay que ordenar.
- Algoritmo de búsqueda  $\rightarrow$  N<sup>o</sup> de elementos entre los que hay que buscar el elemento dado.
- Algoritmo que actúa sobre un conjunto  $\rightarrow$  N<sup>o</sup> de elementos que pertenecen al conjunto.

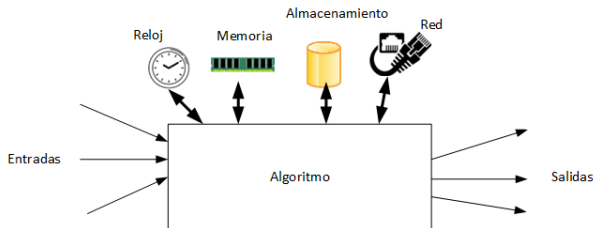
# $T(n)$

- $T(n)$  representa el número de unidades de tiempo (segundos, milisegundos, ...) que un algoritmo tarda en ejecutarse para una entrada de tamaño  $n$ .
- A priori no se conoce el tiempo de ejecución de cada una de las instrucciones simples del programa.
- Como el tiempo de ejecución varía mucho de un ordenador a otro,  $T(n)$  se toma como **el número de instrucciones simples** (asignaciones, comparaciones, operaciones aritméticas) **ejecutadas**.
- Si  $T(n)$  es el coste de un algoritmo, podemos suponer que  $n \geq 0$  y que  $T(n)$  es positiva para cualquier valor del número de entrada  $n$ .
- El coste puede depender de la particularidad de los datos de entrada, no sólo de su número  $n$ .



# Ejemplo del cálculo $T(n)$

- En estos casos,  $T(n)$  será el coste en el caso peor, el coste máximo para datos de tamaño  $n$ .
- Caso mejor (no demasiado interesante).
- Coste promedio (útil, difícil de calcular ¿misma probabilidad para todas las posibles entradas?).



# Ejemplo de cálculo $T(n)$

Ejemplo: Número de cifras de un número entero positivo

```
def cifras (Num):
```

```
    Cont = 1
```

*#1 \* Ta*

```
    while (Num > 9):
```

*#n \* Tc*

```
        Cont = Cont + 1
```

*#(n-1) \* Ts + (n-1) \* Ta*

```
        Num = Num // 10
```

*#(n-1) \* Td + (n-1) \* Ta*

```
    cifras = Cont
```

*#1 \* Ta*

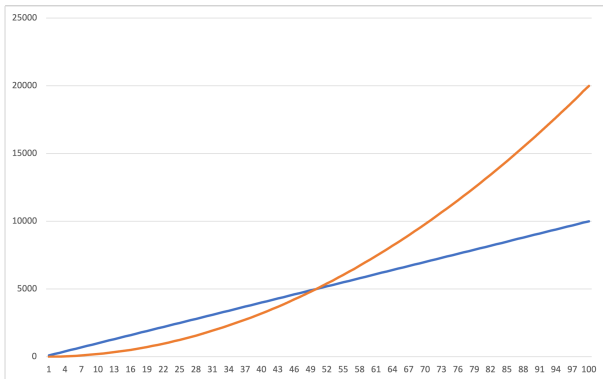
Tiempo total:  $2 * n * Ta + n * Tc + (n-1) * Ts + (n-1) * Td$

$T(n) = 5 * n - 2$  (siendo  $n$  el número de cifras)

# Introducción a la complejidad

Para un problema tenemos dos algoritmos A y B que lo resuelven.  
Evaluamos sus costes:

- $TA(n) = 100 * n$  y  $TB(n) = 2 * n^2$
- ¿Cuál de los dos algoritmos es más eficiente?
  - $n < 50$ , B es más eficiente que A, aunque no demasiado.
  - $n \geq 50$ , A es mucho más eficiente que B (mejor cuanto mayor sean).



# Introducción a la complejidad

Para un problema tenemos dos algoritmos A y B que lo resuelven. Evaluamos sus costes:

- Es más importante la forma funcional ( $n$  en lugar de  $n^2$ ) que las constantes que intervienen (2 frente a 100).
- Como el tiempo de ejecución depende del ordenador (aparece multiplicado por una constante), no nos preocuparemos por los factores que multiplican a la expresión a la hora de analizar la complejidad de un algoritmo.
- En lugar de decir que el coste  $T(n)$  del algoritmo A es  $T(n) = 100 * n$ , diremos que es de orden  $n$ ,  **$O(n)$** .
- $O(n)$  significa que el coste es “alguna constante multiplicada por  $n$ ” siendo  $n$  el tamaño del problema.
- con esta notación de  **$O$** , o **Big O** busca acotar superiormente a la función  **$T(n)$**  con otra función  **$f(n)$** . Es decir, el peor caso en la ejecución de un algoritmo, siendo su complejidad entonces  **$O(f(n))$**

# Reglas de cálculo de la complejidad

Regla de la suma:

- Sean P1 y P2 dos fragmentos de programa con tiempos de ejecución  $T_1(n)$  y  $T_2(n)$ .
- Si  $T_1(n)$  es  $O(f(n))$  y  $T_2(n)$  es  $O(g(n))$ , la ejecución de los dos fragmentos de programa es de orden  $O(\max(f(n), g(n)))$ .
  - $T_1(n) + T_2(n)$  es  $\max(O(f(n)), O(g(n)))$

Regla del producto:

- Sean P1 y P2 dos fragmentos de programa con tiempos de ejecución  $T_1(n)$  y  $T_2(n)$ .
- Si  $T_1(n)$  es  $O(f(n))$  y  $T_2(n)$  es  $O(g(n))$ , entonces:
  - $T_1(n) * T_2(n)$  es  $O(f(n)) * O(g(n))$

# Reglas de cálculo de la complejidad

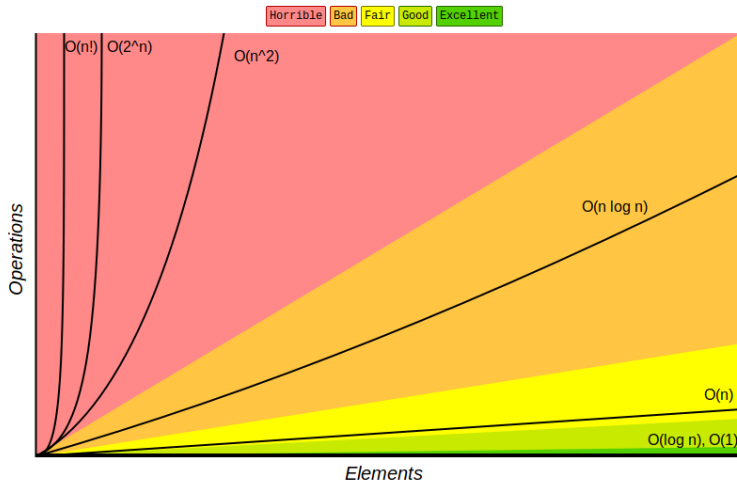
- Los factores constantes no importan:
  - $c * O(f(n))$  es  $O(f(n))$
- Los términos de orden inferior no importan:
  - Si  $T(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ ,
  - $T(n) = O(n^k)$
- La base de los logaritmos no importa:
  - $O(\log n)$  sin especificar la base
  - $(\log_b n = \log_c n * \log_b c)$ .

# Funciones de complejidad habituales

- $O(1) \rightarrow$  Complejidad constante
- $O(\log n) \rightarrow$  Complejidad logarítmica
- $O(n) \rightarrow$  Complejidad lineal
- $O(n^2) \rightarrow$  Complejidad cuadrática
- $O(n^3) \rightarrow$  Complejidad cúbica
- $O(n^k) \rightarrow$  Complejidad polinómica ( $k > 3$ )
- $O(2^n) \rightarrow$  Complejidad exponencial
- $O(n!) \rightarrow$  Complejidad factorial !!!

# Funciones de complejidad habituales

## Big-O Complexity Chart



Fuente: <https://www.bigocheatsheet.com>



# Complejidad de las instrucciones básicas

## Asignación, lectura, escritura, comparaciones

Asignación:  $\rightarrow O(1)$

$S = 1$

Secuencia:  $\rightarrow O(\max(t_{s1}, t_{s2}))$

Secuencia 1 #ts1

Secuencia 2 #ts2

Alternativa:  $\rightarrow O(\max(t_B, t_{s1}, t_{s2}))$

**if** B: #tB

    S1 #ts1

**else**:

    S2 #ts2

# Complejidad de las instrucciones básicas

Bucle For:  $\rightarrow O(\text{num\_iter} * t_{s1})$

```
for i in range (Vini , Vfin ):
    S1 #ts1 (num_iter depende de n)
```

Bucle While:  $\rightarrow O(\text{max\_iter} (t_{s1} + T_B))$

```
while B: #tB
    S1 #ts1
```

# Ejercicios

## Ejercicios - Calcula la complejidad de los siguientes códigos.

### ❶ Ejercicio 1:

```
x += 1
```

### ❷ Ejercicio 2:

```
for i in range (N):  
    x = x + 1
```

### ❸ Ejercicio 3:

```
for i in range (N):  
    for j in range (N):  
        for k in range (N):  
            x = x + 1
```

# Ejercicios

## 4 Ejercicio 4:

```
i = 1
while i < N
    i = i * 2
```

- 5 Desarrollar una función que calcule si un número positivo es primo o no (con un bucle while y un bucle for, para calcular el mejor y el peor caso).
- Recordad que un número es primo solo si es divisible entre 1 y entre sí mismo; en caso de que sea divisible entre cualquier otro número, se dice que no es primo.

# Creditos



Python logo

Python Software Foundation (Trademark)

<https://www.python.org/community/logos/>



GitLab logo

GitLab (Trademark)

<https://about.gitlab.com/press/press-kit/>



©2022 Jesús M. González Barahona, David Cortés Polo, Gregorio Robles  
Algunos derechos reservados.

Este artículo se distribuye bajo la licencia  
“Reconocimiento-CompartirIgual 4.0 Internacional” de Creative Commons,  
disponible en

<http://creativecommons.org/licenses/by-sa/4.0/deed.es>

Este documento (o uno muy similar) está disponible en  
<https://gitlab.etsit.urjc.es/cursoprogram>