Libro

Programación I

Convocatoria de Asignaturas en Abierto 2025-26
Material docente en abierto de la Universidad Rey Juan Carlos
Grado en Inteligencia Artificial

Universidad Rey Juan Carlos

Profesores:
Isaac Lozano Osorio y Salvador Sánchez Alonso

Depositado en BURJC Digital:
https://hdl.handle.net/10115/111417

©2025 Autores Isaac Lozano Osorio y Salvador Sánchez Alonso. Algunos derechos reservados.

Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en:
https://creativecommons.org/licenses/by-sa/4.0/

Índice general

Guía de estudio 3

Apuntes 7

Transparencias de la asignatura 53

Cuaderno de ejercicios y exámenes 507

Software 573

Universidad Rey Juan Carlos

Programación I

Guía de estudio

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
***Esta obra está bajo una licencia** Creative Commons "Atribución Compartir Igual 4.0 Internacional"*

CURSO: 2025-2026

Grado en Inteligencia Artificial

Objetivos de la asignatura

Programación I tiene como objetivo dotar al alumno de los conocimientos y competencias básicas para analizar problemas, diseñar algoritmos para resolverlos y codificar esos algoritmos en un lenguaje de programación moderno para ejecutarlos en el ordenador, así como detectar los posibles errores que se cometen al programar y aprender a corregirlos.

Los alumnos conocerán y pondrán en práctica los conceptos importantes de la tecnología de la programación, como las variables y expresiones, los tipos de datos, las estructuras de control de flujo o la subprogramación. Además, aprenderán a comprobar la corrección de su código y a utilizar los elementos básicos de un entorno de programación moderno, tales como el editor o las herramientas de depuración de código. Asimismo, conocerán y utilizarán con familiaridad los tipos de datos estructurados nativos en el lenguaje vehicular de la asignatura: Python 3.

Para más información sobre las competencias que se pretende que el estudiante adquiera una vez superada la asignatura, consúltese la guía docente oficial.

Temario

Tema 1. Introducción: Conceptos básicos de programación. Introducción a la resolución de problemas mediante algoritmos y programas. Bases de un programa y depuración de código.

Tema 2. Subprogramación: Definición y uso de subprogramas. Invocación a funciones y paso de parámetros.

Tema 3. Estructuras de control: Instrucciones de selección. Instrucciones iterativas. Instrucciones de ruptura del flujo de control.

Tema 4. Estructuras de datos: Tipos de datos simples y estructurados.

Tema 5. Recursividad y algoritmos: Principios de recursividad. Algoritmos básicos de búsqueda y ordenación.

Calendario y planificación del trabajo

El curso se desarrolla a lo largo de 14 semanas. El siguiente cuadro muestra la dedicación, sesión por sesión, a cada punto del temario:

Semana	Fechas	Teoría	Laboratorio
1	15-21sep	Presentación + T1	Problemas T1
2	22sep-28sep	T1 / T2	Problemas T1
3	29sep-5oct	T2	Problemas T2
4	6-12oct	T2	Problemas T2
5	13-19oct	T3	Problemas T3
6	20-26oct	T3	Problemas T3
7	27oct-2nov	Repaso	Problemas de repaso
8	3-9nov	**Prueba escrita 1**	Resolución PE1 / T4
9	10-16nov	T4	Problemas T4
10	17-23nov	T4	Problemas T4
11	24-30nov	T4	Problemas T4
12	1-7dic	T4/T5	*No lectivo viernes 5dic*
13	8-14dic	T5	Problemas T5
14	15-21dic	T5	Problemas T5/Repaso
*	12-ene (12-15h)	**Prueba escrita 2**	

A continuación, se listan las competencias que se adquieren al trabajar cada uno de los temas que incluye el temario de la asignatura:

Tema	Competencias que se adquieren
Tema 1. Conceptos básicos de programación	- Comprender los principios fundamentales de la programación. - Analizar problemas y diseñar soluciones mediante algoritmos. - Traducir algoritmos a programas funcionales. - Aplicar técnicas básicas de depuración para identificar y corregir errores.
Tema 2. Definición y uso de subprogramas	- Modularizar programas mediante funciones y procedimientos. - Implementar subprogramas reutilizables. - Gestionar el paso de parámetros por valor y por referencia. - Comprender el alcance y la vida de las variables en subprogramas.
Tema 3. Estructuras de control	- Aplicar instrucciones condicionales para la toma de decisiones. - Implementar bucles para la repetición controlada de tareas. - Utilizar instrucciones de ruptura para optimizar el flujo de ejecución. - Diseñar algoritmos eficientes usando estructuras de control adecuadas.
Tema 4. Estructuras de datos	- Identificar y utilizar tipos de datos simples (enteros, reales, caracteres, booleanos). - Implementar estructuras compuestas (arrays, registros, listas básicas).

Tema	Competencias que se adquieren
	- Seleccionar la estructura de datos adecuada según el problema. - Comprender la relación entre estructuras de datos y eficiencia algorítmica.
5. Recursividad y algoritmos	- Comprender el concepto y la lógica de la recursividad. - Diseñar e implementar algoritmos recursivos. - Aplicar algoritmos básicos de búsqueda (lineal, binaria) y ordenación (burbuja, selección, inserción). - Analizar la complejidad y eficiencia de algoritmos iterativos y recursivos.

Cada estudiante deberá, teniendo en cuenta la siguiente tabla, que muestra cómo utilizar los materiales didácticos para adquirir las competencias de cada tema de la asignatura, programar su tiempo, teniendo en cuenta que aproximadamente el 60% del esfuerzo debe corresponder al trabajo individual.

Tema	Apuntes	Presentaciones	Cuadernos de práctica
1. Introducción	Tema 1	Diapositivas tema 1	Cuaderno 1
2. Subprogramación	Tema 2	Diapositivas tema 2	Cuaderno 2
3. Estructuras de control	Tema 3	Diapositivas tema 3a y 3b	Cuadernos 3a y 3b
4. Estructuras de datos	Tema 4	Diapositivas tema 4	Cuadernos 4 y 5
5. Recursividad y algoritmos	Tema 5	Diapositivas tema 5	Cuadernos 6 y 7

Metodología docente

Los conceptos para la adquisición de competencias se abordarán tanto desde la perspectiva teórica como desde la práctica.

En las sesiones de teoría se explicarán los conceptos fundamentales para que los estudiantes completen su estudio de manera autónoma, con ayuda de la bibliografía y de recursos abiertos en Internet. Este estudio autónomo dará paso a la resolución práctica de algoritmos en las siguientes sesiones de teoría y formará parte esencial del trabajo personal que permitirá afianzar los conceptos adquiridos.

En las sesiones de laboratorio se concretará el uso de un lenguaje de programación, en este caso Python, que permitirá al estudiante adquirir competencias relacionadas con la programación. Durante las sesiones de laboratorio, el estudiante dispondrá de un ordenador con el entorno apropiado para realizar programas que deberá diseñar y depurar hasta obtener su correcto funcionamiento con la ayuda de los profesores. Se hará uso de un juez de programación para complementar las enseñanzas en las clases de laboratorio.

Evaluación

El mecanismo de evaluación de la asignatura está basado en la evaluación continua. La nota final de la asignatura se calcula como la media ponderada de las notas de las pruebas evaluables (según los porcentajes 30% prueba 1 y 70% prueba 2), siempre y cuando se hayan superado con la nota mínima indicada para cada una de ellas.

Prueba	Peso en la evaluación	Nota mínima	Temas que se evalúan
PE1	30%	4	1 a 3
PE2	70%	4	1 a 5

Si no se hubiera alcanzado la nota mínima en alguna de las pruebas de evaluación, o no se hubiera presentado a alguna de las pruebas, la calificación será la media ponderada de todas las actividades que hubiera realizado si esta fuese inferior a 5. Si esa media ponderada fuese 5 o superior, la calificación será 4.

En caso de superar la nota mínima de todas las actividades evaluación y ser inferior a 5 la nota final de la asignatura, el estudiante deberá presentase en convocatoria extraordinaria a las actividades de evaluación que elija para así superar la asignatura.

Si no se ha presentado a ninguna prueba evaluable, la nota final será "No presentado".

Las pruebas se realizarán de manera individual y presencial en las clases de teoría de las semanas en las que han sido programadas. Estarán formadas por dos tipos de cuestiones: unas breves y concretas de carácter teórico-práctico y otras que consistirán en escribir programas en lenguaje Python.

El examen de la convocatoria extraordinaria (junio) constará de dos partes: una primera con resolución de cuestiones cortas teórico-prácticas, y una segunda de implementación de programas en Python. En todo caso, en la convocatoria extraordinaria los estudiantes solamente se presentarán a la revaluación de las pruebas no superadas.

Universidad Rey Juan Carlos

PROGRAMACIÓN I

Grado en Inteligencia Artificial

Material docente en abierto de la Universidad Rey Juan Carlos

APUNTES DE LA ASIGNATURA

CURSO 2025-26



©2025
Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanchez@urjc.es)
Este documento se distribuye bajo la licencia Compartir Igual 4.0 (CC BY-SA 4.0

Contenido

Tema I – Introducción a la Programación 4
Introducción 4
Problemas y algoritmos 4
Resolución de problemas con ordenador 5
Lenguajes de programación 5
Expresión de algoritmos 7
Elementos de un programa 8
Pensamiento computacional y errores 8
Errores de compilación 9
Entrada y salida 10
print() 10
input() 11
Formateo con f-strings 11
Ejercicios propuestos 12
Tema II – Modularidad 13
Funciones y Procedimientos 13
Cohesión y Acoplamiento 15
Paso de Parámetros 16
Documentación de los subprogramas 17
Documentación en Python: Docstrings y PEP 8 17
Bibliotecas 18
Uso de Bibliotecas en Python 18
Ejercicios propuestos 19
Tema III: Estructuras de control 20
Estructuras selectivas: definición y funcionamiento 20
Sangrado en Python 21
Flujo de control selectivo con if y else 21
if simple 22
if sucesivos 22
Modularización de condiciones 23
Estructuras anidadas 23
Estructuras selectivas múltiples con elif 24
Excepciones 24
Estructuras de control iterativas 25
El bucle for en Python 25

Iterables y secuencias 26
La función range() 27
Bucles anidados 28
El bucle while 28
Ejercicios propuestos 30
Tema IV: Colecciones de Datos 31
Colecciones de Tamaño Fijo 31
Arrays 31
Cadenas 31
Registros 32
Colecciones de Tamaño Variable 32
Pilas y Colas 32
Listas 32
Conjuntos 32
Diccionarios 33
Archivos 33
Otros tipos 33
Colecciones en Python 33
Listas en Python 34
Tuplas 35
Cadenas de Texto 36
Diccionarios 36
Tema V: Recursividad 38
Ejercicios Propuestos 39
Tema VI: Algoritmos 41
Notación Big-O 41
Algoritmos de búsqueda 42
Búsqueda Secuencial 42
Búsqueda Binaria 42
Métodos de Ordenación 43
El método de la Burbuja 43
Selección 43
Inserción Directa 44
Bibliografía 45

Tema I – Introducción a la Programación

Introducción

La programación es el núcleo de la informática moderna y constituye una habilidad esencial para cualquier persona que vaya a dedicarse profesionalmente a la informática. Sin programación, los ordenadores serían simples máquinas incapaces de realizar tareas complejas, pero... ¿qué es programar? Programar significa diseñar instrucciones precisas que permitan a un procesador –un circuito electrónico muy complejo, con miles de elementos que permiten realizar operaciones lógicas, aritméticas y de control– ejecutar operaciones para resolver problemas. Este proceso requiere comprender el problema, abstraerlo y traducirlo a un lenguaje que la máquina pueda interpretar.

Aprender a programar no consiste únicamente en memorizar la sintaxis de un lenguaje como Python o C. Implica desarrollar una forma de pensar conocida como pensamiento computacional, que consiste en descomponer problemas en partes más simples, identificar patrones, abstraer detalles irrelevantes y diseñar soluciones que puedan ser automatizadas. Este enfoque es aplicable más allá de la informática: se utiliza en ciencia, ingeniería, economía y cualquier disciplina que requiera resolver problemas complejos.

Este modo de pensar, centrado en la lógica y la sistematización, tiene raíces profundas en la historia del conocimiento. La idea de resolver problemas mediante pasos ordenados no es nueva: ha sido parte del pensamiento matemático desde hace siglos. Así, el pensamiento computacional no surge en el vacío, sino que se apoya en una tradición intelectual que ha evolucionado con el tiempo y que hoy se materializa en los algoritmos que usamos para programar.

Históricamente, el concepto de algoritmo se remonta a Al-Khwārizmī, matemático persa del siglo IX, cuyo nombre dio origen al término. Sus trabajos sobre aritmética introdujeron reglas sistemáticas para resolver operaciones con números decimales. Desde entonces, la idea de describir procesos mediante pasos ordenados ha evolucionado hasta convertirse en el núcleo de la programación. Hoy, los algoritmos son la base de toda solución computacional, desde los sistemas operativos hasta las aplicaciones móviles.

Problemas y algoritmos

Un **problema**, en términos computacionales, es una situación que requiere una solución mediante un conjunto de datos de entrada y un resultado esperado. Resolver un problema implica tres fases: comprender el enunciado, diseñar una estrategia y obtener la solución. La estrategia se materializa en un algoritmo, una secuencia finita de pasos bien definidos que transforman las entradas en salidas.

Un **algoritmo** es una secuencia finita y ordenada de pasos o instrucciones que se siguen para resolver un problema o realizar una tarea específica. Cada paso debe ser claro, preciso y ejecutable, y el algoritmo debe tener un punto de inicio y un punto de finalización.

Para que un algoritmo sea válido, debe cumplir ciertas propiedades esenciales. En primer lugar, debe ser **finito**, es decir, no puede ejecutarse indefinidamente; debe terminar en un número limitado de pasos. En segundo lugar, sus instrucciones deben ser **realizables**: el procesador que las ejecuta debe ser capaz de interpretarlas y llevarlas a cabo. Además, las instrucciones deben estar formuladas sin ambigüedad, evitando interpretaciones múltiples. Finalmente, el algoritmo debe ser **determinista**, lo que significa que, dados los mismos datos de entrada, siempre producirá el mismo resultado.

Un ejemplo cotidiano que ilustra el concepto de algoritmo es una receta de cocina. Preparar una tortilla francesa implica seguir pasos concretos: cascar un huevo, batirlo, calentar aceite, verter el huevo y darle forma. Cada paso es claro, secuencial y finito. Si seguimos la receta correctamente, obtendremos siempre el mismo plato. Este ejemplo muestra que los algoritmos no son exclusivos de la informática: son procedimientos que aplicamos en la vida diaria.

Resolución de problemas con ordenador

Cuando utilizamos un ordenador para resolver problemas, seguimos un proceso iterativo que consta de varias etapas. Primero, realizamos un análisis para determinar qué datos son necesarios y qué información debe producirse. Después, pasamos al diseño, donde descomponemos el problema en subproblemas más simples y especificamos los pasos para resolver cada uno. A continuación, procedemos a la codificación, que consiste en traducir el algoritmo a un lenguaje de programación. Una vez escrito el programa, lo ejecutamos y comprobamos su funcionamiento. Este ciclo no termina con la entrega del programa: el mantenimiento es una fase crucial, ya que los programas deben adaptarse a cambios y corregir errores.

Este proceso es incremental y requiere retroalimentación constante. Por ejemplo, si detectamos un error durante la ejecución, debemos volver al diseño o incluso al análisis para corregirlo.

Lenguajes de programación

Un lenguaje de programación es un lenguaje formal diseñado para expresar algoritmos de manera que un ordenador pueda ejecutarlos. Los lenguajes de programación son considerados lenguajes formales porque están diseñados con reglas sintácticas y semánticas estrictas que permiten una traducción automática, determinista y no ambigua de las instrucciones al lenguaje máquina. A diferencia del lenguaje natural, que admite múltiples interpretaciones y cuyo significado a menudo depende del contexto, los lenguajes de programación deben ser precisos y estructurados para que un compilador o intérprete pueda procesarlos sin errores ni confusiones. Esta formalidad garantiza que cada instrucción tenga un único significado y produzca siempre el mismo resultado bajo las mismas condiciones, lo cual es esencial para el desarrollo de sistemas confiables y reproducibles. Además, esta característica facilita la verificación, depuración y mantenimiento del código, así como su análisis por herramientas automáticas.

Existen distintos niveles de lenguajes, siendo los de más bajo nivel los más cercanos al modo en que “habla” la máquina y de mayor nivel aquellos que se parecen más al modo en que hablamos los humanos. Así, el lenguaje de programación más básico es el **lenguaje máquina**, compuesto por instrucciones en código binario que el procesador interpreta directamente. Por encima se encuentra el **lenguaje ensamblador**, que utiliza símbolos y

mnemónicos para representar instrucciones. Finalmente, los **lenguajes de alto nivel**, como C, Java o Python, permiten escribir programas de forma más cercana al lenguaje humano, facilitando la comprensión y el desarrollo.



Ilustración 1. Código máquina

```
008548ED: jmp   854927H
008548EF: cmp   esi,18H
008548F2: jne   8548E8H
008548F4: cmp   byte [edi][48H],0
008548F8: jne   854927H
008548FA: mov   eax,edi
008548FC: call  near 855118H
00854901: mov   eax,[eax][33H]
00854904: je    854927H
00854907: and   eax,00000200
0085490C: cmp   eax,0
0085490F: je    854927H
00854911: push  0
00854913: lea   ebx,[edi]
00854915: lea   ecx,[edi][4]
00854918: mov   eax,00000000
```

Ilustración 2. Código ensamblador

Por ejemplo, el siguiente código en C imprime un mensaje en pantalla:

```c
#include <stdio.h>
int main(void){
    printf("Hola mundo");
    return 0;
}
```

En Python, la misma tarea se reduce a una sola línea:

```python
print("Hola mundo");
```

Lenguajes como C y Python se consideran de alto nivel porque permiten al programador escribir instrucciones más cercanas al lenguaje humano que al lenguaje máquina, ocultando detalles técnicos como la gestión directa de memoria o el uso de registros. Sin embargo, dentro de esta categoría, la legibilidad del código se ha convertido en un valor cada vez más apreciado. Python, por ejemplo, ha sido diseñado con una sintaxis clara y concisa que facilita la comprensión del código incluso para quienes no son expertos, lo que permite al programador centrarse más en la resolución creativa de problemas que en los aspectos técnicos del lenguaje. Esta accesibilidad no solo mejora la productividad, sino que también fomenta la colaboración, el aprendizaje y la innovación, al reducir las barreras cognitivas que tradicionalmente han acompañado a la programación.Esto muestra cómo los lenguajes de alto nivel simplifican la programación.

Expresión de algoritmos

Los algoritmos pueden expresarse de varias formas: en lenguaje natural, mediante pseudocódigo, diagramas de flujo o directamente en un lenguaje de programación. Consideremos el algoritmo de Euclides para calcular el máximo común divisor (MCD) de dos números. Este matemático griego del siglo III a.C., propuso uno de los primeros algoritmos conocidos para calcular el máximo común divisor (MCD) de dos números. Su método consiste en restar repetidamente el número menor al mayor hasta que ambos sean iguales, o bien usar la versión moderna basada en el resto de la división. Grosso modo, el algoritmo se basa en la idea de que el MCD de dos números no cambia si se reemplaza el mayor por su diferencia (o su resto) con el menor.

El **pseudocódigo** es una forma de describir algoritmos utilizando una mezcla de lenguaje natural y estructuras propias de la programación, pero sin ajustarse a la sintaxis estricta de ningún lenguaje específico. Su objetivo es expresar de manera clara y ordenada los pasos que debe seguir un programa, facilitando la comprensión y el diseño antes de escribir el código real. Gracias a su simplicidad y legibilidad, el pseudocódigo permite centrarse en la lógica del problema sin distraerse con detalles técnicos.

El algoritmo de Euclides en pseudocódigo sería algo así:

```
Algoritmo Euclides(a,b)
Mientras b != 0:
    r = a mod b # resto de a/b
    a = b
    b = r
Fin
```

Los **diagramas de flujo** son representaciones gráficas que muestran, de forma ordenada y visual, los pasos que sigue un algoritmo o proceso. Utilizando símbolos estandarizados (como óvalos, rectángulos y rombos) conectados por flechas para indicar el flujo de ejecución y las decisiones que se toman en cada etapa, un diagrama de flujo facilita la comprensión, el análisis y la comunicación de la lógica de un programa, permitiendo a programadores y no programadores visualizar cómo se resuelve un problema antes de escribir el código.

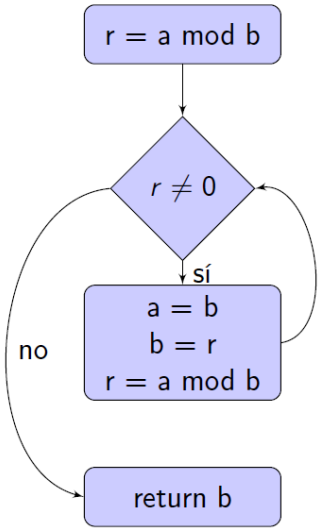


Ilustración 3. Representación mediante diagrama de flujo del Algoritmo de Euclides

Este ejemplo ilustra cómo un mismo algoritmo puede representarse en distintos niveles de abstracción.

Elementos de un programa

Un programa está compuesto por instrucciones, constantes, variables, expresiones y palabras reservadas. Las **instrucciones** son órdenes que el procesador ejecuta. Las **constantes** son valores fijos que no cambian durante la ejecución. Las **variables** son elementos cuyo valor puede variar. Las **expresiones** combinan operadores y operandos para producir un resultado. Las **palabras reservadas** son términos con significado especial en el lenguaje, como `if` o `while`. Además, los programas incluyen **comentarios**, que documentan el código y facilitan su comprensión.

En Python, los tipos de datos básicos incluyen enteros, flotantes, cadenas y booleanos. También existen colecciones como listas, tuplas y diccionarios. Las conversiones de tipo pueden ser implícitas o explícitas, utilizando funciones como `int()` o `float()`.

Pensamiento computacional y errores

El **pensamiento computacional** es una forma de abordar y resolver problemas que implica descomponerlos en partes más simples, identificar patrones, abstraer detalles irrelevantes y diseñar algoritmos que puedan ser ejecutados por una máquina. Para un programador, este tipo de pensamiento es esencial porque le permite estructurar soluciones de manera lógica, eficiente y escalable, facilitando la creación de programas que no solo funcionen correctamente, sino que también sean mantenibles y reutilizables. El pensamiento computacional implica, en suma, diseñar algoritmos y traducirlos a programas.

Ahora bien, durante el proceso de diseño, codificación de un algoritmo en un lenguaje de programación, pueden surgir errores. En cualquier lenguaje de programación, los errores se dividen en dos grandes categorías: **errores de compilación** y **errores de ejecución**. Aunque Python es un lenguaje interpretado y no tiene una fase de compilación tradicional como C o Java, el concepto sigue siendo aplicable: los errores sintácticos en Python son equivalentes a los errores de compilación en lenguajes compilados.

Errores de compilación

Estos errores, también denominados sintácticos en los lenguajes interpretados, ocurren antes de que el programa se ejecute. Se producen cuando el código no respeta las reglas del lenguaje: faltan paréntesis, hay indentación incorrecta, o se usan palabras reservadas de forma indebida. El compilador (o el intérprete en Python) detecta el problema y detiene la ejecución.

Ejemplo:

```
print("Hola mundo"

Salida:
SyntaxError: unexpected EOF while parsing
```

El error indica que falta cerrar el paréntesis y en consecuencia se informa al programador de que el programa no puede ejecutarse hasta corregirlo.

Otro caso común podría ser el siguiente:

```
if x>0:
print("Hola")

Salida:
IndentationError: expected an indented block
```

El problema aquí radica en que Python exige bloques correctamente indentados, pero el print no está correctamente tabulado hacia la derecha pues su ejecución depende del if. Este tipo de error es muy frecuente en principiantes.

Observa que estos errores son estáticos, es decir, se detectan antes de ejecutar el código. En lenguajes compilados, el compilador no genera el ejecutable; en Python, el intérprete no avanza a la siguiente instrucción.

El segundo tipo de errores son los errores de ejecución. Estos errores aparecen cuando el programa ya está en marcha. El código es sintácticamente correcto, pero ocurre algo inesperado durante la ejecución. Son más peligrosos porque dependen de los datos y condiciones reales.

Ejemplo:

```
x = 20
y = int(input('Introduce un valor para y: '))
print("Resultado:", x / y)

Salida (si el usuario introduce un cero para la y):
ZeroDivisionError: division by zero
```

El error no sucede siempre, solo en aquellas ejecuciones en que el usuario elige el valor 0 para y. El código es sintácticamente correcto pero su ejecución está sujeta a un riesgo: que el usuario introduzca valores distintos de cero.

Otro ejemplo:

```
edad = input("Introduce tu edad: ")
print(edad + 10)

Salida:
TypeError: can only concatenate str (not "int") to stry = 0
```

El error ocurre porque `input()` devuelve un texto (`string`) lo cual no se puede sumar directamente con un entero.

Observa que los errores de tiempo de ejecución son dinámicos, ya que dependen de los datos y del contexto. El código puede funcionar con ciertos valores y fallar con otros.

En resumen:

- Errores de compilación: relacionados con la forma del código. Se detectan antes de ejecutar. Impiden que el programa se ejecute. Para prevenirlos usamos editores con resaltado de sintaxis, linters y herramientas que verifiquen el código.
- Errores de ejecución: relacionados con datos y condiciones. Se detectan únicamente mientras el programa se ejecuta y para evitarlos debemos validar los datos de entrada, usar estructuras de control para evitar condiciones peligrosas (por ejemplo, comprobar que el divisor no sea cero), y llevar a cabo pruebas exhaustivas del código.

En cualquier lenguaje, esta distinción es fundamental: los primeros son más fáciles de corregir porque el compilador o intérprete indica la línea exacta; los segundos requieren análisis lógico y pruebas.

Entrada y salida

La interacción con el usuario es una parte fundamental de cualquier programa. En Python, esta interacción se realiza principalmente mediante dos funciones: `print()` para mostrar información en pantalla y `input()` para recibir datos del usuario. Además, Python ofrece mecanismos avanzados para formatear la salida, como las f-strings, que permiten construir cadenas dinámicas de manera clara y eficiente.

print()

Es la función más utilizada para mostrar información en la consola. Su sintaxis básica es:

```
print(objeto1, objeto2, ..., sep=' ', end='\n')
```

- sep: define el separador entre los elementos. Por defecto es un espacio.
- end: define qué se imprime al final. Por defecto es un salto de línea (\n).

Ejemplo:

```
print("Hola", "mundo")
# Salida: Hola mundo

print("Python", "Java", "C++", sep=" | ")
# Salida: Python | Java | C++

print("Adios", end=", ")
print("mundo cruel")
# Salida: Adios, mundo cruel
```

input()

Permite leer datos desde la consola. Siempre devuelve una cadena de texto (tipo `str`), incluso si el usuario introduce números. Por ello, es habitual convertir el valor a otro tipo usando funciones como `int()` o `float()`. Su sintaxis es:

```
input(mensaje_para_el_usuario)
```

Ejemplo:

```
edad = int(input("Introduce tu edad: "))
print("Tendrás", edad + 1, "años el próximo año.")
```

Dao que hay una conversión del original (texto, `str`) al destino (entero, `int`) si el usuario introduce un valor no convertible (por ejemplo, texto en lugar de número), se producirá un error de tipo `ValueError`. En consecuencia, y para evitar estos problemas, se recomienda usar mensajes claros en `input()` para guiar al usuario y validar siempre los datos introducidos.

Formateo con f-strings

Introducidas en Python 3.6, las f-strings permiten incrustar expresiones dentro de cadenas usando llaves `{}`. Son más legibles y eficientes que concatenar cadenas con + o que usar `format()`. Agunas de las ventajas de las f-strings son su mayor claridad en comparación con las alternativas, que permiten incluir operaciones directamente y que soportan especificadores de formato (decimales, alineación, etc.).

Ejemplo:

```
nombre = "Ana"
edad = 25
print(f"Hola, {nombre}. Tienes {edad} años.")
```

Otro ejemplo con expresiones dentro de llaves:

```
a, b = 5, 3
print(f"La suma de {a} y {b} es {a + b}.")
```

Finalmente, un ejemplo con formato numérico:

```
pi = 3.14159265
print(f"Valor de pi con 2 decimales: {pi:.2f}")
```

Ejercicios propuestos

1. Redacta un algoritmo en lenguaje natural para preparar café, asegurándote de que cumpla con las propiedades mencionadas: finito, realizable y determinista.
2. Crea un programa simple en Python que pida dos números y muestre su suma.
3. Crea un programa que pida el nombre y la edad del usuario y muestre un mensaje usando f-strings.
4. Escribe un programa que solicite dos números y muestre su suma, diferencia y producto en una sola línea formateada.
5. Modifica el programa anterior para que muestre los resultados con dos decimales.

Tema II – Modularidad

La modularidad es fundamental en el diseño moderno de software. Consiste en dividir un programa en partes independientes llamadas módulos o subprogramas, cada uno encargado de una tarea específica. Esta división permite que el desarrollo sea más organizado, que el código sea más fácil de entender y que se puedan realizar pruebas y mantenimientos de forma más eficiente. En esencia, modularizar es aplicar el principio de "divide y vencerás" al desarrollo de software.

Cuando programamos, es muy común encontrarnos repitiendo las mismas operaciones una y otra vez. Imagina que estás desarrollando una aplicación que necesita convertir temperaturas de grados Celsius a Fahrenheit en varios puntos del programa. Podrías escribir la fórmula cada vez que la necesites, pero eso no solo es tedioso, sino que también aumenta el riesgo de cometer errores. En lugar de eso, puedes crear un subprograma —una función— que encapsule esa lógica y cada vez que necesites hacer esa conversión, simplemente llamas a la función. Esto es reutilización de código: escribes una vez, usas muchas. Y si algún día necesitas cambiar la fórmula (por ejemplo, para ajustarla a una nueva norma), solo hay que modificarla en un único lugar.

El diseño modular de programas fomenta la reutilización del código. Un subprograma bien diseñado puede ser utilizado en múltiples programas o en diferentes contextos dentro del mismo programa. Por ejemplo, una función que calcule el área de una figura geométrica puede ser utilizada en una aplicación educativa, en una herramienta de diseño gráfico o en un simulador físico, sin necesidad de reescribirla. Esta capacidad de reutilización es clave para reducir el tiempo de desarrollo y mejorar la calidad del software.

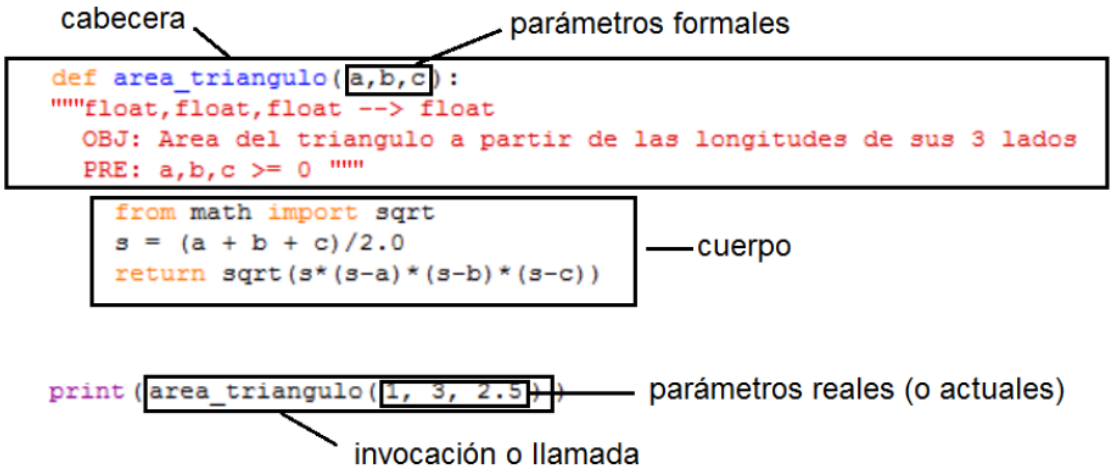
Otra ventaja es el hecho de que los subprogramas hacen que el código sea más legible. Cuando vemos una línea como `celsius_a_fahrenheit(25)`, uno sabe inmediatamente qué está ocurriendo, sin tener que descifrar una fórmula matemática. Esto es especialmente útil cuando trabajas en equipo o cuando revisas tu propio código semanas después. Cada subprograma actúa como una pequeña caja con una etiqueta clara que dice "esto hace tal cosa".

Finalmente, los subprogramas facilitan la depuración y el mantenimiento. Si algo no funciona como se esperaba, se puede probar el subprograma por separado, sin tener que revisar todo el programa completo. Y de cara al mantenimiento, cuando se necesiten hacer mejoras, el diseño modular permite que podamos hacerlo de forma localizada, modificando solo aquellos módulos o subprogramas afectados por el cambio, pero sin temor a "estropear" otras partes del código.

Funciones y Procedimientos

En programación, los subprogramas se dividen principalmente en dos tipos: funciones y procedimientos. Esta terminología varía según los lenguajes que utilicemos, ya que por ejemplo C o Python llaman función a cualquier módulo, mientras que otros lenguajes como Pascal distinguen claramente entre procedimiento y función.

Sintaxis para declarar subprogramas en Python:



Una función es un subprograma que realiza un cálculo y devuelve un valor. Por ejemplo, una función que calcule el cuadrado de un número y lo retorne al código que la ha invocaod, puede definirse en Python como:

```
def cuadrado(x):
    return x * x
```

Por otro lado, un procedimiento es un subprograma que realiza una acción pero no devuelve un valor. Su propósito puede ser mostrar información en pantalla, modificar estructuras de datos o interactuar con el usuario. Un ejemplo clásico sería:

```
def saludar(nombre):
    print(f"Hola, {nombre}")
```

La distinción entre funciones y procedimientos es importante porque afecta la forma en que se integran en el flujo del programa. Las funciones suelen formar parte de expresiones más complejas, mientras que los procedimientos se utilizan como instrucciones independientes. Las funciones como hemos dicho están diseñadas para calcular y devolver un valor. Por ello, pueden integrarse fácilmente dentro de expresiones más complejas, como parte de asignaciones, condiciones, bucles o incluso como argumentos de otras funciones. Por ejemplo:

```
def cuadrado(x):
    return x * x

# Uso dentro de una expresión
resultado = 3 + cuadrado(4) / 2
```

Aquí, la función cuadrado se utiliza como parte de una operación matemática más amplia. Este tipo de integración es muy común.

En cambio, los procedimientos (también llamados funciones `void` en algunos lenguajes como C o Java) no devuelven valores, sino que realizan acciones: mostrar información, modificar estructuras, registrar datos, etc. Por esta razón, se utilizan como instrucciones

independientes, y no como parte de expresiones. Su invocación suele ser una línea de código por sí sola. Por ejemplo:

```
def mostrar_saludo(nombre):
    print(f"Hola, {nombre}")

# Uso como instrucción
mostrar_saludo("Angela")
```

Una invocación a este procedimiento no puede formar parte de una expresión como por ejemplo `x = mostrar_saludo("Sandra")`, porque no devuelve nada útil para asignar. Su propósito es realizar una acción visible o con efectos secundarios.

Entender esta diferencia es clave para modelar correctamente el comportamiento deseado. Si el objetivo es obtener un resultado que se pueda usar más adelante (por ejemplo, un cálculo), se debe diseñar una función. Si el objetivo es realizar una acción directa (como imprimir o guardar), se debe usar un procedimiento. Además, esta distinción ayuda a mantener la separación entre lógica de cómputo e interfaz, lo cual es una buena práctica de diseño. Por ejemplo, una función que calcule la media de dos números no debería imprimir el resultado, sino devolverlo. La impresión puede hacerse en otro lugar del programa, donde se controle la interfaz con el usuario.

El siguiente es un ejemplo de mala práctica, pues mezcla cómputo e interfaz:

```
def calcular_media(a, b):
    media = (a + b) / 2

print(f"La media es: {media}") # No recomendable
```

Con la misma función, un uso correcto sería:

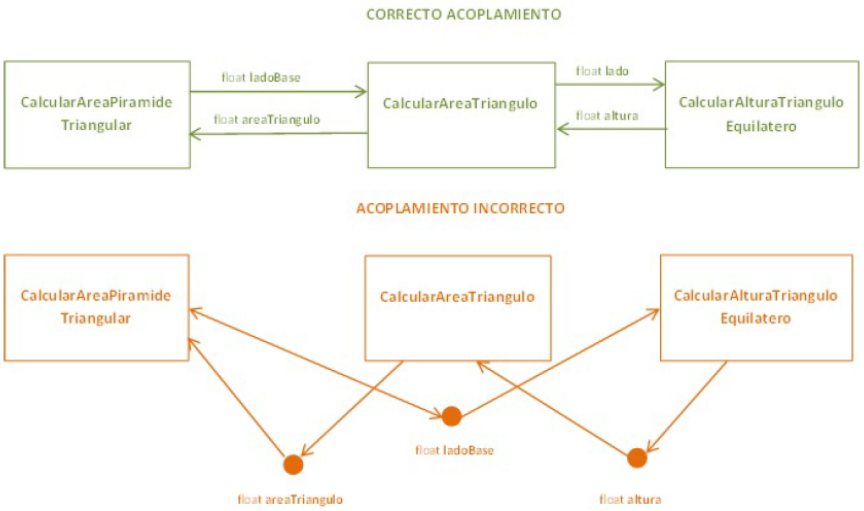
```
def calcular_media(a, b):
    return (a + b) / 2

# Interfaz separada
resultado = calcular_media(10, 20)
print(f"La media es: {resultado}")
```

Este enfoque modular permite reutilizar la función calcular_media en contextos donde no se desea imprimir, sino almacenar, comparar o transmitir el resultado.

Cohesión y Acoplamiento

Dos conceptos clave en el diseño de subprogramas son la cohesión y el acoplamiento. La cohesión se refiere al grado en que las instrucciones dentro de un subprograma están relacionadas entre sí. Un subprograma con alta cohesión realiza una única tarea bien definida. Por ejemplo, una función que calcule la hipotenusa de un triángulo rectángulo tiene alta cohesión porque todas sus instrucciones contribuyen a ese único objetivo.



El acoplamiento, en cambio, mide la dependencia entre subprogramas. Un buen diseño busca bajo acoplamiento, lo que significa que los subprogramas interactúan entre sí solo mediante parámetros y valores de retorno, evitando el uso de variables globales. Esto reduce los efectos secundarios y facilita la reutilización y el mantenimiento del código.

Paso de Parámetros

El paso de parámetros es el mecanismo mediante el cual los subprogramas (funciones o procedimientos) reciben datos desde el código que los invoca. Este concepto es fundamental en cualquier lenguaje de programación, ya que permite que los subprogramas operen sobre datos externos sin depender de variables globales, lo que mejora la modularidad y la claridad del código.

Existen principalmente dos formas de pasar parámetros:

1. **Paso por valor**: Se envía una copia del valor al subprograma. Esto significa que cualquier modificación dentro del subprograma no afectará al valor original. Es común en lenguajes como C, Java (para tipos primitivos), y Python (para tipos inmutables como int, float, str).
2. **Paso por referencia**: Se envía una referencia al valor original, lo que permite modificar directamente el dato en el contexto del programa llamante. Esto se utiliza en lenguajes como C++ (con punteros o referencias), Java (para objetos), y Python (para objetos mutables como listas o diccionarios).

El uso correcto del paso de parámetros permite diseñar subprogramas más seguros, reutilizables y fáciles de probar. Además, ayuda a evitar efectos secundarios indeseados que pueden surgir al modificar variables globales.

Python utiliza un modelo de paso de parámetros que puede parecer híbrido: todo se pasa por asignación de referencia, pero el comportamiento depende de si el objeto es mutable o inmutable.

- Para **tipos inmutables** (`int`, `float`, `str`, `tuple`), el comportamiento es similar al paso por valor: si se modifica el parámetro dentro del subprograma, el valor original no cambia.
- Para **tipos mutables** (`list`, `dict`, `set`), el comportamiento es como el paso por referencia: las modificaciones dentro del subprograma afectan al objeto original.

Documentación de los subprogramas

Documentar subprogramas es una práctica esencial en cualquier lenguaje de programación. Una buena documentación permite entender qué hace un subprograma, qué espera recibir como entrada, qué devuelve como salida y bajo qué condiciones funciona correctamente. Esto es especialmente útil en equipos de desarrollo, en proyectos a largo plazo y en entornos educativos.

Para los principiantes, escribir primero la documentación antes de implementar la función ayuda a modelar correctamente el problema. Obliga a pensar en los tipos de datos, en las precondiciones necesarias y en el objetivo del subprograma. Esta técnica, conocida como programación orientada a contratos, mejora la calidad del diseño y reduce errores.

Documentación en Python: Docstrings y PEP 8

En Python, la documentación de subprogramas se realiza mediante docstrings, que son cadenas de texto encerradas entre triple comillas (""") justo después de la definición del subprograma. Estas docstrings pueden seguir diferentes estilos, siendo el más recomendado el de la PEP 257, y complementado por la PEP 8, que establece buenas prácticas de estilo de código.

Ejemplo de docstring estilo contrato (propuesto en la asignatura):

```
def hipotenusa(a, b):
    """ float, float -> float
        OBJ: Calcula la hipotenusa de un triángulo rectángulo.
        PRE: a > 0, b > 0
    """
    return (a**2 + b**2)**0.5
```

Este estilo compacto es útil en entornos educativos, ya que obliga a pensar en:

- Tipos de entrada y salida
- Objetivo del subprograma
- Precondiciones necesarias

Ejemplo estilo PEP 484 (más formal y usado en entornos profesionales):

```
def hipotenusa(a: float, b: float) -> float:
```

```
    """
       Calcula la hipotenusa de un triángulo rectángulo.

       Args:
          a (float): Longitud del primer cateto.
          b (float): Longitud del segundo cateto.

       Returns:
          float: Longitud de la hipotenusa.
    """
    return (a**2 + b**2)**0.5
```

Si bien este estilo es más detallado y compatible con herramientas de análisis estático, como linters y generadores de documentación automática, para un primer contacto con la programación puede resultar excesivamente prolijo.

Bibliotecas

Una biblioteca (*library*) es un conjunto de funciones, clases y constantes agrupadas en un módulo que puede ser reutilizado en distintos programas. Las bibliotecas permiten encapsular funcionalidades relacionadas, como operaciones matemáticas, manipulación de cadenas, acceso a redes, etc.

En cualquier lenguaje, el uso de bibliotecas es una forma de reutilizar código probado y optimizado, evitando reinventar la rueda. Además, muchas bibliotecas están mantenidas por comunidades o empresas, lo que garantiza su calidad y evolución.

Uso de Bibliotecas en Python

En Python, las bibliotecas se importan usando la instrucción `import`. Existen dos formas principales:

- Importar todo el módulo:

```
import math
print(math.sqrt(16)) # 4.0
```

- Importar solo elementos específicos:

```
from math import pi, sin
print(pi) # 3.141592...
print(sin(pi / 2)) # 1.0
```

Ejemplo:

```
import random

# Generar un número aleatorio entre 1 y 10
print(random.randint(1, 10))
```

```
# Seleccionar un elemento aleatorio de una lista
colores = ["rojo", "verde", "azul"]
print(random.choice(colores))
```

Estas bibliotecas (`math`, `random`, etc.) son parte de la **biblioteca estándar de Python**, lo que significa que vienen incluidas con el lenguaje y no requieren instalación adicional, pero en aquellos casos en que las bibilitecas sean muy esecíficas será necesario instalarlas en nuesto entorno local antes de importarlas en nuestros programas y poder utilizarlas.

Finalmente, indicar que es posible crear nuestras propias bibliotecas. Para ello solo debemos agrupar las funciones deseadas en un archivo .py y luego importarlas en otros programas. Por ejemplo:

```
# archivo: conversiones.py
def euros_a_dolares(euros, tasa):
   return euros * tasa

# archivo principal
import conversiones
print(conversiones.euros_a_dolares(50, 1.1))
```

Ejercicios propuestos

1. Función de área de pirámide. Define una función que calcule el área total de una pirámide cuadrangular regular, utilizando subprogramas para el cálculo del área base, perímetro y apotema.
2. Procedimiento de saludo bilingüe. Crea un procedimiento que reciba un nombre y muestre un saludo en español e inglés.
3. Biblioteca de conversiones. Diseña una biblioteca con funciones para convertir entre distintas unidades: euros a pesetas, metros a kilómetros, grados Celsius a Fahrenheit.
4. Función con precondiciones. Implementa una función que calcule la media aritmética de dos números, documentando claramente las precondiciones (por ejemplo, que ambos sean enteros positivos).

Tema III: Estructuras de control

En programación, el flujo de control se refiere a la secuencia en la que se ejecutan las instrucciones de un programa. En su forma más básica, sin estructuras de control, el flujo es estrictamente secuencial, es decir, las instrucciones se ejecutan una tras otra en el orden en que aparecen en el código. Sin embargo, los programas reales rara vez se limitan a esta ejecución lineal. Para que un programa pueda tomar decisiones, repetir acciones o reaccionar ante diferentes situaciones, se utilizan las llamadas estructuras de control.

Estas estructuras permiten modificar el flujo de ejecución, y se dividen principalmente en dos tipos: estructuras selectivas y estructuras repetitivas. Las primeras permiten ejecutar diferentes bloques de código dependiendo de una condición, mientras que las segundas permiten repetir bloques de código mientras se cumpla una condición.

Estructuras selectivas: definición y funcionamiento

Una estructura selectiva permite que, dependiendo del resultado de una condición lógica, se ejecuten unas instrucciones y se omitan otras. Este tipo de estructura es esencial cuando se requiere que el programa actúe de manera diferente ante distintas entradas o estados.

Por ejemplo, si un programa debe responder de forma distinta si el usuario introduce un número positivo o negativo, se utilizará una estructura selectiva para decidir qué bloque de código ejecutar.

La condición lógica que se evalúa siempre produce un resultado booleano: verdadero (True) o falso (False). En Python, esto se logra mediante operadores condicionales como:

```
== (igualdad)

!= (diferente)

> (mayor que)

< (menor que)

>= (mayor o igual que)

<= (menor o igual que)
```

Y también mediante operadores lógicos como:

```
and (y lógico)

or (o lógico)

not (negación lógica)
```

El resultado de evaluar la condición determina qué bloque de código se ejecuta. Si la condición es verdadera, se ejecuta el bloque asociado; si es falsa, se puede ejecutar otro bloque alternativo o simplemente no ejecutar nada.

Ejemplo básico: par o impar

```
import random
n = random.randint(0, 10)
if n % 2 == 0:
   print("El número generado es par")
else:
   print("El número generado es impar")
```

Este ejemplo genera un número aleatorio entre 0 y 10 y luego evalúa si es par. La condición n % 2 == 0 verifica si el resto de dividir n entre 2 es cero, lo que indica que el número es par.

Sangrado en Python

Python utiliza el sangrado (indentación) para definir bloques de código. Cada vez que una sentencia termina con dos puntos (`:`), como en `if`, `for`, `while`, `def`, etc., se espera que las instrucciones que dependen de ella estén indentadas.

Por ejemplo:

```
if x > 0:
    print("x es positivo")
```

Aquí, la línea `print("x es positivo")` está indentada, lo que indica que pertenece al bloque del `if`.

Una excepción a esta regla ocurre cuando hay una sola instrucción dependiente, que puede escribirse en la misma línea:

```
if x > 0: print("x es positivo")
```

Aunque esto es válido, se recomienda mantener el estilo indentado para mejorar la legibilidad del código.

Algunos consejos sobre la utilización del sangrado:

- Las líneas en blanco y los comentarios no afectan el sangrado.
- Es recomendable evitar el uso de tabuladores en el editor y usar espacios en su lugar, ya que los tabuladores pueden causar errores difíciles de detectar.
- Se pueden usar tablas mentales o diagramas para visualizar la estructura jerárquica del código y entender mejor qué instrucciones dependen de otras.

Flujo de control selectivo con if y else

Aunque no siempre es obligatorio incluir ambas ramas, la estructura `if` en Python tiene un carácter bifurcador, lo que significa que permite dividir el flujo de ejecución del programa en dos caminos distintos, dependiendo del resultado de una condición lógica. Esta condición se evalúa como una expresión booleana, es decir, su resultado será `True` (verdadero) o `False` (falso). Si la condición se cumple (es verdadera), se ejecuta un bloque de instrucciones específico. En caso contrario, se puede ejecutar un bloque alternativo utilizando la cláusula `else`.

Este comportamiento es fundamental en programación, ya que permite que el software reaccione de manera diferente ante distintas situaciones. Por ejemplo, un programa puede mostrar un mensaje distinto si el usuario introduce una edad menor de 18 años, o si introduce una mayor.

La sintaxis general de una estructura `if-else` en Python es la siguiente:

```
if condición:
  # bloque de código que se ejecuta si la condición es verdadera
else:
  # bloque de código que se ejecuta si la condición es falsa
```

Ejemplo práctico:

```
edad = 20
if edad >= 18:
   print("Eres mayor de edad.")
else:
   print("Eres menor de edad.")
```

En este ejemplo, la condición `edad>=18` se evalúa y si el valor de `edad` es mayor o igual que 18, se imprime el mensaje "Eres mayor de edad". Si no se cumple la condición, se imprime "Eres menor de edad". Un ejemplo similar sería el siguiente:

```
if x > 0:
    print("x es positivo")
else:
    print("x es negativo")
```

Este otro ejemplo muestra otra estructura selectiva doble. Si la condición `x > 0` se cumple, se ejecuta el primer bloque. Si no se cumple, se ejecuta el bloque del else.

if simple

Cuando no hay parte `else`, hablamos de un if simple. Por ejemplo:

```
if x == 5:
   print("x es 5")
   print("aun x es 5")
   print("sigue siendo 5")
```

Aquí se ejecutan varias instrucciones dentro del bloque del `if` si la condición se cumple. Todas las instrucciones están indentadas, lo que indica que pertenecen al mismo bloque.

if sucesivos

A menudo un if sucede a otro por lo que el código es cuestionado varias veces sucesivas: tras salir del primer if nos encontramos con el siguiente... y así sucesivamente. Ejemplo:

```
edad = 5
if edad < 2:
```

```
   print("Bebé")
if edad > 18:
   print("Adulto")
print("Fin")
```

Este ejemplo muestra dos estructuras if independientes. Cada una evalúa una condición distinta. El programa imprime "Fin" independientemente del resultado de las condiciones anteriores.

Modularización de condiciones

Las condiciones de una estructura selectiva pueden incluir llamadas a funciones, lo que permite modularizar el código y mejorar su legibilidad.

```
import random
def es_par(x):
   return x % 2 == 0

n = random.randint(0, 10)
if es_par(n):
   print("El número generado es par")
else:
   print("El número generado es impar")
```

Aquí, la función es_par encapsula la lógica para determinar si un número es par. Esto permite reutilizar la función en otros contextos y hace que el código principal sea más limpio.

Estructuras anidadas

Las estructuras de control pueden contener otras estructuras de control dentro de sus bloques. Esto se conoce como anidamiento.

Veamos un ejemplo. Un año es bisiesto si es divisible entre 4. Sin embargo, si el año también es divisible entre 100, entonces no es bisiesto. Pero si el año es divisible entre 400, entonces sí es bisiesto, incluso si es divisible entre 100. Así, para determinar si un año es bisiesto escribiríamos el siguiente código:

```
if año % 4 == 0:
   if año % 100 == 0:
      if año % 400 == 0:
         print("Es bisiesto")
      else:
         print("No es bisiesto")
   else:
      print("Es bisiesto")
else:
   print("No es bisiesto")
```

Este ejemplo muestra cómo se pueden anidar múltiples if para evaluar condiciones complejas.

Estructuras selectivas múltiples con elif

Cuando hay más de dos posibles caminos, se utiliza un condicional múltile que en otros lenguajes como C o Pascal tiene una estructura específica para ello. En Python, el soporte para estructuras condicionales múltiples es `elif` (`else if`):

```
import random
dia = random.randint(0, 10)
if dia == 1:    print("Lunes")
elif dia == 2:    print("Martes")
elif dia == 3:    print("Miércoles")
elif dia == 6:    print("Sábado")
elif dia == 7:    print("Domingo")
else:    print("ERROR: Día incorrecto")
```

Excepciones

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de instrucciones. Este evento suele estar relacionado con un error o una situación inesperada, como intentar dividir por cero, acceder a un archivo que no existe, convertir una cadena de texto que no representa un número a tipo entero, etc.

Cuando ocurre una excepción y no se gestiona adecuadamente, el programa se detiene y muestra un mensaje de error. Para evitar que esto suceda y permitir que el programa continúe funcionando o reaccione de forma controlada, Python ofrece un mecanismo para capturar y manejar excepciones. Esta estructura se compone de varios bloques:

try

Este bloque contiene el código que puede generar una excepción. Python intenta ejecutar las instrucciones dentro del `try`. Si todo va bien, se continúa normalmente. Si ocurre una excepción, se salta al bloque `except`.

except

Aquí se define cómo reaccionar ante una excepción. Se puede capturar una excepción genérica o especificar el tipo de error que se espera (por ejemplo, `ValueError`, `ZeroDivisionError`, etc.).

else

Este bloque es opcional y se ejecuta solo si no ocurre ninguna excepción en el bloque try. Es útil para separar el código que debe ejecutarse solo cuando no hay errores.

finally

También es opcional. Este bloque se ejecuta siempre, haya ocurrido o no una excepción. Se usa para realizar tareas de limpieza, como cerrar archivos o liberar recursos.

La sintaxis general es la siguiente:

```
try:    # Código que puede causar una excepción
except TipoDeExcepcion:   # se ejecuta si ocurre esa excepción
else:    # se ejecuta si no ocurre ninguna excepción
finally:    # se ejecuta siempre, ocurra o no una excepción
```

Ejemplo completo

```
try:
    numero = int(input("Introduce un número entero: "))
except ValueError:
    print("Error: no se ha introducido un número válido.")
else:
    print("Has introducido el número:", numero)
finally:
    print("Fin del proceso.")
```

En este ejemplo, si el usuario introduce algo que no puede convertirse a entero (como "abc"), se lanza una excepción `ValueError`, y se ejecuta el bloque `except`. Si la conversión es exitosa, se ejecuta el bloque `else`. Independientemente del resultado, el bloque `finally` se ejecuta al final.

Por ejemplo:

```
anno_actual = 2020
x = input("Introduzca el año en que nació: ")
try:
    anno_nacimiento = int(x)
except:
    print("Algo ha fallado: no es posible calcular su edad")
else:
    edad = anno_actual - anno_nacimiento
    print("Su edad es:", edad, "años")
```

Este código intenta convertir la entrada del usuario en un número. Si falla, se captura la excepción y se muestra un mensaje de error. Si tiene éxito, se calcula la edad.

El uso de excepciones permite que los programas sean más robustos y tolerantes a errores**,** facilitando una ejecución más fluida y segura.

Estructuras de control iterativas

Las estructuras de control iterativas permiten ejecutar un bloque de instrucciones repetidamente, lo cual es esencial para automatizar tareas, procesar grandes volúmenes de datos, realizar cálculos acumulativos, entre otros. A diferencia de las estructuras selectivas, que deciden entre caminos alternativos, las iterativas repiten acciones mientras se cumpla una condición o durante un número determinado de veces.

En Python, las estructuras iterativas son: `for`(para repetir un bloque de código sobre una secuencia conocida) y `while` (para repetir mientras se cumpla una condición). En Python no existe un soporte para la estructura `do...while` o `repeat...until` pero se puede implementar mediante `while` con lógica adaptada.

El bucle for en Python

El bucle `for` permite ejecutar un bloque de código un número conocido de veces, recorriendo una secuencia o objeto iterable. La variable de control toma sucesivamente los valores de la secuencia, y por cada uno de ellos se ejecuta el cuerpo del bucle.

El bucle `for` en Python es una estructura de control iterativa que permite recorrer secuencias o objetos iterables (como listas, cadenas, rangos, diccionarios, etc.) y ejecutar un bloque de código una vez por cada elemento de la secuencia.

A diferencia de otros lenguajes como C o Java, donde el bucle for se basa en una condición y una variable de control explícita (por ejemplo, `for (i=0; i<5; i++)`), en Python el bucle for se basa en la iteración directa sobre los elementos de un iterable.
Sintaxis genérica del bucle for en Python

```
for variable in iterable:
    # bloque_de_instrucciones (cuerpo del bucle)
```

donde:

- `variable`: es una variable de control que toma el valor de cada elemento del iterable en cada iteración.
- `iterable`: es el objeto que se recorre. Puede ser una lista, una cadena, un rango, un conjunto, una tupla, etc.
- `bloque_de_instrucciones`: es el conjunto de instrucciones que se ejecutan en cada iteración. Debe estar indentado correctamente.

Por ejemplo:

```
for i in [1, 2, 3]:
    print(i)
```

Este bucle imprimirá los números 1, 2 y 3, uno por uno.

Hay que tener en cuenta que si al for se le proporciona una secuencia está vacía, el cuerpo del bucle no se ejecuta en absoluto.

Iterables y secuencias

En Python, un iterable es cualquier objeto que puede ser recorrido elemento por elemento. Las secuencias son un tipo especial de iterable que tienen un orden definido y permiten operaciones como indexación. Ejemplos de secuencias:

- Listas (list): mutables, permiten añadir, modificar y eliminar elementos.
- Tuplas (tuple): inmutables.
- Cadenas de texto (str): secuencias de caracteres, también inmutables.

Ejemplos:

```
nombres = ["Ana", "Luis", "Carlos"]
for nombre in nombres:
    print("Hola", nombre) # muestra por pantalla Ana, Luis y Carlos

for letra in "Python":
    print(letra)  # Muestra letra a letra la palabra Python
```

La función range()

`range()` es una función muy utilizada en bucles for para generar secuencias de números enteros. Su sintaxis es:

```
range(inicio, fin, paso)
```

donde

- inicio: valor inicial (incluido).
- fin: valor final (excluido).
- paso: incremento entre valores (puede ser negativo).

Ejemplos del uso de range:

```
range(5)         # [0, 1, 2, 3, 4]
range(2, 5)      # [2, 3, 4]
range(3, 10, 2)  # [3, 5, 7, 9]
range(10, 5, -1) # [10, 9, 8, 7, 6]
```

Ejemplos de uso combinado con for:

```
# Ejemplo 1: imprimir números del 1 al 19
for i in range(1, 20):
    print(i, end=" ")
# Salida: 1 2 3 ... 19

# Ejemplo 2: recorrer una lista de palabras
Pythonwords = ['cat', 'dog', 'lion']
for w in words:
    print(w, end='/')
# Salida: cat/dog/lion/

# Ejemplo 3: suma de números impares
x = 0
for i in range(1, 20, 2):
    x += i
print(x)
# Salida: 100

# Ejemplo 4: suma de 10 números aleatorios
import random
suma = 0
for i in range(10):
    numero_actual = random.randint(0, 100)
    suma += numero_actual
print("La suma es:", suma)

# Ejemplo 5: cálculo de factorial
def factorial(n):
   result = 1
   for i in range(1, n+1): result *= i
return result
```

Bucles anidados

Un bucle anidado es una estructura en la que un bucle está contenido dentro del cuerpo de otro bucle. Es decir, el bucle interno se ejecuta completamente cada vez que el bucle externo realiza una iteración. Este tipo de estructura es muy útil cuando se necesita repetir acciones en dos dimensiones, como recorrer filas y columnas de una matriz, generar combinaciones de elementos, o construir estructuras jerárquicas.

Imaginemos que tenemos dos bucles:

```
for i in range(3):       # bucle externo
    for j in range(2):   # bucle interno
        print(i, j)
```

El funcionamiento paso por paso es el siguiente:

- El bucle externo comienza con i = 0.
- En ese momento, el bucle interno se ejecuta completamente: j = 0, luego j = 1.
- Luego el bucle externo pasa a i = 1, y nuevamente el bucle interno se ejecuta completamente.
- Esto se repite hasta que el bucle externo termina.

Un ejemplo más complejo sería el siguiente código, que imprime las tablas de multiplicar del 2, 4, 6, 8 y 10:

```
for i in range(2, 11, 2):
    for j in range(1, 11):
        print(i, "x", j, "=", i * j)
    print("-------------")
```

El bucle while

El bucle while ejecuta un bloque de código mientras se cumpla una condición. Puede ejecutarse cero o más veces.

Sintaxis:

while condición: instrucciones

Es crucial asegurarse de que la condición eventualmente se vuelva falsa, para evitar bucles infinitos.

Ejemplo de bucle infinito:

Pythonn = 10while n > 0: print(n) n = n + 1 # Error: n nunca disminuye

Ejemplos con while

Cruzar valores de dos variables

Pythonimport randoma = random.randint(1, 100)b = random.randint(1, 100)print(f"Iniciales: a={a}, b={b}")while a < b: a += 1 b -= 1print(f"Finales: a={a}, b={b}")

Este bucle ajusta a y b hasta que se cruzan.

Invertir las cifras de un número

Pythonimport randominverso = 0n = random.randint(1, 1000)while n % 10 == 0: n //= 10 # Elimina ceros finalesprint(f"Número a invertir: {n}")while n > 0: inverso *= 10 inverso += n % 10 n //= 10print("Resultado:", inverso)

Este código invierte las cifras de un número entero.

Simulación de repeat...until

Python no tiene una estructura repeat, pero se puede simular con while.

Ejemplo: calcular la media de números aleatorios hasta que salga un cero

Pythonimport randomCENTINELA = 0cuantos = 0suma = 0valor = random.randint(0, 100)while valor != CENTINELA: suma += valor cuantos += 1 valor = random.randint(0, 100)if cuantos > 0: print("La media es:", suma / cuantos)

Ejemplo: encontrar un elemento en una lista

Pythonimport randommy_list = ["One", "Two", "Three", "Four"]intentos = 1my_choice = random.choice(my_list)while my_choice != "Three": intentos += 1 my_choice = random.choice(my_list)print(f"Ha costado {intentos} intentos dar con el 3")

Este ejemplo muestra cómo repetir una acción hasta encontrar un valor específico.

Control forzado del bucle: break y continue

break: termina el bucle inmediatamente.

continue: salta a la siguiente iteración sin ejecutar el resto del cuerpo.

Ejemplo con break:

Pythonfor i in range(10): if i == 5: break print(i)

Salida: 0 1 2 3 4

Ejemplo con continue:

Pythonfor i in range(5): if i == 2: continue print(i)

Salida: 0 1 3 4

Ejercicios propuestos

1. Clasifica la estatura de una persona que se introduce por teclado como bajita, normal, alta o muy alta usando estructuras if, elif, else.
2. Calcula la suma de todos los números impares entre 1 y 100 utilizando un bucle for.
3. Muestra las tablas de multiplicar del número ingresado y de los tres siguientes usando bucles anidados.
4. Invierte las cifras de un número entero positivo utilizando un bucle while.
5. Adivina un número secreto entre 1 y 50, dando pistas al usuario hasta que acierte o consuma 3 oportunidades, usando while y funciones.

Tema IV: Colecciones de Datos

Las colecciones de datos son tipos de datos que agrupan múltiples elementos bajo una misma estructura. Estas abstracciones permiten manejar y acceder a los datos de forma eficiente y organizada. Cada tipo de colección tiene características específicas que definen su comportamiento:

- Homogeneidad o heterogeneidad de los elementos.
- Orden.
- Acceso por posición o por clave.
- Tamaño fijo o variable.
- Mutabilidad o inmutabilidad.

La disponibilidad de colecciones varía según el lenguaje de programación. Por ejemplo, en C los arrays están integrados como estructuras básicas, mientras que, en Python, aunque también existen arrays, estos no están directamente disponibles en el lenguaje sino a través de bibliotecas específicas (p.e. `numpy`). Otros ejemplos serían Java, donde existen colecciones como HashMap para pares clave-valor y TreeSet para conjuntos ordenados, o Haskell, que ofrece estructuras persistentes como listas inmutables y mapas (Data.Map), que facilitan la programación segura y concurrente.

Colecciones de Tamaño Fijo

Las colecciones de tamaño fijo son estructuras que tienen una capacidad determinada desde su creación y no pueden crecer ni reducirse dinámicamente. Es decir, no es posible durante la ejecución del programa modificar su tamaño. Un ejemplo típico son los arrays en C, donde el número de elementos se define al inicializar y permanece constante.

Arrays

Un array es una colección de elementos del mismo tipo, accesibles por posición. Su tamaño debe definirse previamente, y puede tener una o más dimensiones:

- Unidimensional: vector.
- Bidimensional: matriz.
- Etc.

Ejemplo en C:

```
int numeros[5] = {1, 2, 3, 4, 5};
```

Cadenas

Las cadenas son secuencias de caracteres que representan texto y permiten operaciones como concatenación, búsqueda y manipulación de subcadenas. Son fundamentales en la mayoría de los lenguajes de programación. Ejemplo:

```
cadena = "Hola mundo"
```

Registros

Los registros agrupan campos de distintos tipos, accesibles por nombre. Representan entidades que podrían asimilarse a una ficha de información sobre algún elemento del mundo real tal como la información que se almacena en una biblioteca sobre un libro, la información que almacena una universidad sobre uno de sus alumnos, etc. Ejemplos:

- Libro: título, autor, editorial, número de páginas, etc.
- Coche: matrícula, color, marca, modelo, cilindrada, etc.

Los registros rara vez se utilizan de manera aislada y, así, son útiles como base para colecciones más complejas, como arrays de registros o pilas de registros.

Colecciones de Tamaño Variable

Las colecciones de tamaño variable son estructuras que pueden crecer o reducirse dinámicamente según las necesidades del programa, sin requerir una capacidad fija desde el inicio. Ejemplos comunes son los ArrayList en Java o las listas en Python, que permiten añadir o eliminar elementos de forma flexible.

Pilas y Colas

Las pilas y las colas son estructuras que organizan los elementos según el orden de entrada y salida. Una pila sigue el principio LIFO (*Last In, First Out*), donde el último elemento en entrar es el primero en salir, lo que resulta útil en algoritmos de retroceso, gestión de llamadas en la pila de ejecución y operaciones de deshacer. Por otro lado, una cola sigue el principio FIFO (*First In, First Out*), donde el primer elemento en entrar es el primero en salir, siendo común en sistemas de impresión, planificación de procesos y colas de mensajes. Su ventaja principal es la simplicidad y el control del flujo, mientras que su desventaja es la falta de acceso aleatorio a elementos intermedios. Existen variantes como colas de prioridad, colas dobles (deque) y colas concurrentes para escenarios más complejos.

Listas

Las listas son estructuras que permiten acceder a elementos por índice, lo que facilita operaciones rápidas de lectura. En muchos lenguajes son homogéneas (todos los elementos del mismo tipo), aunque en otros, como Python, pueden ser heterogéneas. Existen variantes como las tuplas, que son listas inmutables y útiles para datos que no deben cambiar, o las listas ordenadas, que mantienen un orden específico según criterios definidos. Las listas son muy versátiles, pero su desventaja es que operaciones como inserción o eliminación en posiciones intermedias pueden ser costosas.

Conjuntos

Un conjunto es una estructura de datos que representa una colección no ordenada de elementos únicos, lo que significa que no permite duplicados. Su objetivo principal es garantizar la unicidad y ofrecer operaciones eficientes como comprobación de pertenencia, inserción y eliminación. Ejemplo en Python:

```
conjunto = {1, 2, 3}
```

Diccionarios

Los diccionarios son estructuras que almacenan pares clave-valor, permitiendo acceder a la información mediante la clave en lugar de la posición. Son muy útiles para búsquedas rápidas y organización de datos asociativos. Cada clave debe ser única, mientras que los valores pueden repetirse. Un ejemplo típico sería un diccionario que relaciona nombres con números de teléfono.

```
persona = {"nombre": "Ana", "edad": 30}
```

Archivos

Los archivos permiten almacenar datos de forma persistente fuera de la memoria principal, evitando su pérdida al apagar el sistema. Pueden ser homogéneos, cuando contienen datos del mismo tipo, o heterogéneos, si incluyen distintos tipos. Existen tres tipos principales: los archivos de texto, que guardan caracteres en formato legible por humanos; los binarios, que almacenan datos en su representación interna para mayor eficiencia; y los indizados, que incorporan un índice para acceder rápidamente a registros específicos, muy usados en bases de datos.

Otros tipos

Hay muchos otros tipos de colecciones. Las que pueden tablas hash permiten acceso rápido a pares clave-valor, con la ventaja de su eficiencia y la desventaja de colisiones y dependencia de una buena función hash. Los arrays dinámicos son un tipo especial de array que ajustan su tamaño automáticamente, ofreciendo acceso por índice y que si bien son flexibles y simples, las inserciones/eliminaciones son costosas en términos de eficiencia. Además, algunos lenguajes incluyen estructuras avanzadas como árboles balanceados (*TreeSet* en Java) para mantener orden de los elementos con tiempos de acceso eficientes, colas concurrentes para entornos multihilo, o las estructuras persistentes de algunos lenguajes funcionales como Clojure, Haskell o Scala, que mantienen versiones anteriores cuando se modifican, en lugar de cambiar el estado original.

Colecciones en Python

Python permite colecciones heterogéneas, es decir, pueden contener elementos de distintos tipos:

```
mi_lista = [1, "texto", 3.14, True]
```

Cada posición de esta lista puede contener tipos simples como: entero, real, booleano, string, o tipos compuestos como otras listas, objetos, archivos, etc. Lo mismo sucede con otras colecciones.

Un concepto clave en Python es el de *mutabilidad*. Un objeto es mutable si su contenido puede cambiar después de su creación, mientras que los inmutables no permiten modificaciones y cualquier cambio genera un nuevo objeto. Las estructuras mutables, como las listas y los diccionarios, son útiles cuando se necesita actualizar datos dinámicamente, pero requieren precaución en entornos concurrentes porque pueden provocar efectos colaterales. Por otro lado, los objetos inmutables, como los números, las cadenas y las tuplas, aportan seguridad y estabilidad, ya que su estado no cambia, lo que

facilita la programación funcional y reduce errores. Esta distinción influye en el rendimiento y en el diseño de algoritmos, por lo que es importante elegir la estructura adecuada según el caso de uso.

Listas en Python

Las listas son colecciones mutables y ordenadas. Se definen con corchetes:

```
lista = [1, 2, [3, 4, [5]], 'a', 3.2, True]
```

Las listas de Python permiten acceder a sus elementos por índice, que puede ser positivo o negativo. El primer elemento se encuentra en el índice 0, mientras que los índices negativos permiten acceder desde el final, siendo -1 el último elemento. Esta característica hace que las listas en Python sean muy flexibles pues facilitan los recorridos y manipulación de los datos tanto desde el inicio como desde el final.

Operaciones comunes:

```
lista.index(elem): posición del elemento.
lista[i]: acceso por índice.
elem in lista: pertenencia.
lista.count(elem): número de apariciones.
```

Modificación de listas:

```
len(lista): tamaño
append(elem): añadir al final
extend(lista2): añadir elementos de otra lista
insert(pos, elem): insertar en posición
remove(elem): eliminar primera aparición
sort(): ordenar
reverse(): invertir orden
pop(): eliminar y retornar último elemento
+: concatenar
*: repetir
```

Las secuencias en Python, como las listas, son la base de la iteración con for. Esto es así porque la estructura for recorre los elementos de una secuencia uno a uno y ejecuta el cuerpo del bucle cada vez para cada valor, sin necesidad de utilizar los índices de cada elemento para ello. Por ejemplo:

```
frutero = ['pera', 'higo', 'fresa', 'caqui']
for fruta in frutero:
    print("Una pieza de fruta:", fruta)
```

slicing

El slicing (traducido sería algo así como “rebanado”) es una técnica para extraer partes de una secuencia (como listas, cadenas o tuplas) usando la notación `[inicio:fin:salto]`.

Es muy útil para copiar partes de una lista sin modificar la original, pero debe tenerse en cuenta que:

- inicio: índice donde comienza la subsecuencia (incluido).
- fin: índice donde termina (excluido).
- salto: número de posiciones que se avanza entre elementos (opcional).

Si se omite inicio o fin, Python asume el principio o el final de la secuencia. Si se omite el salto, se usa 1 por defecto.

Ejemplos prácticos:

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Desde el índice 2 hasta el 6
print(nums[2:6])    # [2, 3, 4, 5]

# Desde el inicio hasta el índice 4
print(nums[:4])    # [0, 1, 2, 3]

# Desde el índice 5 hasta el final
print(nums[5:])    # [5, 6, 7, 8, 9]

# Copia completa
print(nums[:])        # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Con salto: cada 2 elementos
print(nums[::2])      # [0, 2, 4, 6, 8]

# Sublista invertidaprint(nums[::-1])     # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# Desde el índice 1 hasta el 8, saltando de 3 en 3
print(nums[1:8:3])    # [1, 4, 7]
```

Más ejemplos:

```
preposiciones = ['a','ante','bajo','cabe','con','contra','de','desde','en']
a = preposiciones[1:5]  # ['ante', 'bajo', 'cabe', 'con']
b = preposiciones[4:]   # ['con', 'contra', 'de', 'desde', 'en']
c = preposiciones[:3]   # ['a', 'ante', 'bajo']
d = preposiciones[:]    # copia completa
```

En resumen, el slicing es muy potente porque permite copiar, invertir y extraer patrones sin bucles explícitos.

Tuplas

Las tuplas en Python son colecciones ordenadas e inmutables, lo que significa que mantienen el orden de los elementos y no pueden modificarse después de su creación. Se definen con paréntesis, por ejemplo:

```
a = (0.056, 38.15, -6.09, 1.267, -51.2)
```

Al ser inmutables, no es posible añadir, eliminar ni cambiar elementos, lo que las hace seguras para datos que no deben modificarse. Una ventaja importante es que pueden

usarse como claves en diccionarios, algo que las listas no permiten. Además, es sencillo convertir entre listas y tuplas:

```
lista = list(a)      # Convierte tupla en lista
tupla = tuple(lista) # Convierte lista en tupla
```

Las tuplas son útiles para representar datos fijos, como coordenadas, fechas o registros. También admiten desempaquetado, es decir, extraer los datos de cada uno de sus índices por separado. Así:

x, y, z = (1, 2, 3)

Y soportan operaciones como indexación, *slicing* y concatenación, igual que las listas, pero sin modificación del contenido.

Cadenas de Texto

Las cadenas en Python son secuencias inmutables de caracteres, lo que significa que no se pueden modificar una vez creadas. Se definen usando comillas simples o dobles, por ejemplo:

```
cadena1 = "Universidad Rey Juan Carlos"
cadena2 = 'Campus de Móstoles'
```

Al ser inmutables, cualquier operación que "modifique" una cadena en realidad crea una nueva. Sin embargo, ofrecen múltiples métodos útiles:

```
upper()             # convierte todo a mayúsculas
lower()             # convierte todo a minúsculas
find(car)           # devuelve la posición de un carácter o subcadena
replace(car1, car2) # reemplaza texto
split()             # divide la cadena en una lista según un separador
```

Ejemplos prácticos:

```
texto = "Hola Mundo"
print(texto.upper())        # "HOLA MUNDO"
print(texto.lower())        # "hola mundo"
print(texto.find("Mundo"))  # 5
print(texto.replace("Hola", "Adiós"))  # "Adiós Mundo"
print(texto.split())        # ['Hola', 'Mundo']
```

Además, las cadenas soportan indexación y slicing, igual que las listas:

```
print(texto[0])     # 'H'
print(texto[1:5])   # 'ola '
print(texto[::-1])  # 'odnuM aloH' (invertida)
```

Diccionarios

Los diccionarios en Python son estructuras que almacenan pares clave-valor, permitiendo acceder a los datos mediante la clave en lugar de un índice. Son muy eficientes para

búsquedas, inserciones y eliminaciones, con tiempo promedio O(1) gracias a su implementación basada en tablas hash. Se definen con llaves {}, por ejemplo:

```
rae = {
   "colapso": "Destrucción, ruina de una institución o sistema.",
   "desinformación": "Acción y efecto de desinformar.",
   "todólogo": "Persona que cree saber y dominar varias especialidades."
}
```

Operaciones básicas:

```
# Agregar o modificar:
rae["nuevo"] = "Definición nueva"

# Eliminar:
del rae["colapso"]

# Vaciar:
rae.clear()

# Comprobar pertenencia:
"todólogo" in rae  # True

# Tamaño:
len(rae)

# Copiar:
copia = rae.copy()

# Listar contenido:
Claves: rae.keys()
Valores: rae.values()
Items (pares clave-valor): rae.items()
```

Los diccionarios son muy versátiles y se usan para representar datos estructurados, como configuraciones, registro, etc. Además, permiten anidamiento (diccionarios dentro de diccionarios) y son mutables, lo que implica que pueden modificarse en cualquier momento.

Tema V: Recursividad

La recursividad es una técnica en la que una función se define en términos de sí misma para resolver problemas complejos dividiéndolos en subproblemas más simples. Cada llamada recursiva debe acercarse al caso base, que es la condición de salida que detiene el proceso y evita bucles infinitos.

Más allá de la programación, la recursividad aparece en la naturaleza: estructuras como los fractales (por ejemplo, la forma de una coliflor o de la costa de un continente) se repiten a diferentes escalas con el mismo patrón. También está presente en fenómenos como la ramificación de árboles, el sistema circulatorio o las conchas marinas, donde cada parte es una versión reducida del todo.

En programación, esta idea se traduce en algoritmos como el recorrido de árboles, la búsqueda en grafos o el cálculo de secuencias. Por ejemplo, el factorial se define como:

```
n!=n×(n−1)!
```

Aquí, el problema se reduce a calcular el factorial de un número menor hasta llegar al caso base, lo que en Python podríamos programar como:

```
def factorial(n):
if n == 0:
   return 1  # Caso base
else:
   return n * factorial(n - 1)  # Caso recursivo
```

La recursividad es poderosa porque refleja la auto-similitud que vemos en fractales: cada parte contiene la estructura del todo. Sin embargo, debe usarse con cuidado por el coste en memoria y tiempo.

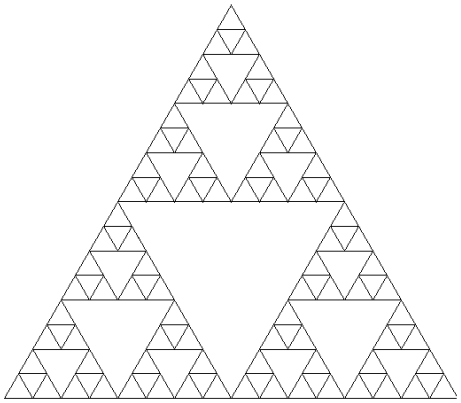


Ilustración 4. El triángulo de Sierpinski es un ejemplo típico de autosimilitud recursiva

Para diseñar un algoritmo recursivo eficaz, se deben seguir estos pasos:

1. Identificar el problema.
2. Definir el caso base, es decir, el caso más simple que se puede resolver directamente.

3. Descomponer el problema en uno o varios subproblemas que tiendan al caso base.
4. Evitar el desbordamiento de pila, asegurando que cada llamada recursiva se acerque al caso base.

Esquema básico:

```
def R(problema):
   if problema es simple:
      resolver directamente
   else:
      descomponer en subproblemas
      invocar R(subproblema)
```

Ejemplos:

```
# Factorial
def factorial(n):
   if n == 0:
      return 1
   else:
      return n * factorial(n - 1)

# Multiplicación por sumas sucesivas
def multiplicar(n, m):
   if m == 1:  return n
   else:       return n + multiplicar(n, m - 1)

# Serie de Fibonacci
def fibonacci(n):
   if n <= 1:  return 1
   else:       return fibonacci(n - 2) + fibonacci(n - 1)

# Palíndromo
def es_palindromo(pal, inicio, fin):
   if inicio >= fin:   return True
   elif pal[inicio] != pal[fin]:   return False
   else:   return es_palindromo(pal, inicio + 1, fin - 1)

#Palíndromo con slicing:
def es_palindromo(frase):
   frase = frase.replace(' ', '')
   if len(frase) <= 1:         return True
   elif frase[0] != frase[-1]: return False
   else:                       return es_palindromo(frase[1:-1])
```

Ejercicios Propuestos

1. Programar la división entera entre dos números sin usar / ni *.
2. Invertir una cadena utilizando recursividad.
3. Programa una función que cuente las vocales en una frase.

Tema VI: Algoritmos

La algoritmia es la rama de la informática que estudia los algoritmos, es decir, secuencias de pasos que permiten resolver un problema o realizar una tarea. Como ya se dijo en el primer tema, un algoritmo no es solo un conjunto de instrucciones: debe ser correcto, eficiente y realizable. Pero cuando hablamos de eficiencia, nos referimos principalmente a dos aspectos:

- Tiempo: ¿Cuántas operaciones necesita el algoritmo para terminar?
- Espacio: ¿Cuánta memoria consume mientras se ejecuta?

Estos factores son importantes porque, aunque dos algoritmos puedan resolver el mismo problema, uno puede hacerlo mucho más rápido o usando menos recursos y no es algo irrelevante. Si bien en un primer curso de programación la eficiencia de los algoritmos no es un aspecto crítico y es más importante aprender la lógica de la programación y las bases de la modularidad, entre otros, más adelante sí lo será.

¿Por qué medir la eficiencia? Imagina que tienes que ordenar una lista de 10 elementos: cualquier método funcionará rápido. Pero si la lista tiene 10 millones de elementos, la elección del algoritmo puede marcar la diferencia entre tardar segundos o horas.

Notación Big-O

Para comparar algoritmos, usamos la notación Big-O (O grande), que describe cómo crece el número de operaciones cuando aumenta el tamaño de los datos (n). No mide el tiempo exacto, sino de manera relativa al tamaño de los datos (n) y particularmente lo hace observando el comportamiento asintótico, es decir, cómo se comporta el algoritmo cuando n es muy grande.

Así, el más bajo nivel de complejidad es O(1), a lo que se denomina “tiempo constante”, es decir, no depende del tamaño de los datos. Si la complejidad es O(n), hablamos de una eficiencia que crece linealmente con el tamaño de la entrada. O(log n) indica que el tiempo de ejecución crece muy lentamente, típico en algoritmos que dividen el problema en mitades (como la búsqueda binaria), $O(n^2)$ indica que crece cuadráticamente, común en algoritmos de ordenación simples, y así sucesivamente n^3, etc.

Por ilustrar este concepto, algunos ejemplos prácticos. La búsqueda lineal, es decir, el algoritmo que recorre todos los elementos hasta encontrar el buscado tiene una complejidad asintótica de orden O(n), mientras que la búsqueda binaria, que como veremos más adelante divide la lista ordenada en mitades sucesivas, realiza el mismo trabajo en O(log n).

En la práctica, si tenemos una lista de 1.000.000 elementos:

- Un algoritmo O(n) puede tardar segundos.
- Uno $O(n^2)$ puede tardar horas.
- Uno O(log n) será casi instantáneo.

Algoritmos de búsqueda

Búsqueda Secuencial

Este método recorre una lista elemento por elemento hasta encontrar el valor buscado o llegar al final. Es simple, pero poco eficiente para listas grandes, ya que su complejidad es O(n).

```
def posicion(lista, elemento):
   pos = None
   i = 0
   while pos is None and i < len(lista):
       if lista[i] == elemento:
           pos = i
       i += 1
   return pos
```

Búsqueda Binaria

Este método de búsqueda solo puede aplicarse en listas ordenadas. El algoritmo divide el rango de búsqueda en mitades sucesivas, descartando la mitad donde no puede estar el elemento. Su complejidad es O(log n), mucho más eficiente que la secuencial, pero si la lista de elementos no se encuentra ordenada hay que añadir la ordenación previa.

```
def posicion(A,elemento):
   """ list, object -> int
       OBJ: Posicion en una lista, None si no esta.
            Utiliza el algoritmo de busqueda binaria
       PRE: lista ordenada
   """
   pos = None
   ini = 0
   fin = len(A)-1
   while ini <= fin and pos == None:
     centro = (ini + fin)//2
     if lista[centro] == elemento:  pos = centro
     elif elemento < lista[centro]: fin = centro-1
     else: ini = centro + 1
   return pos
```

Una alternativa de implementación es su implementación con recursividad, la cual es muy clara conceptualmente y también tiene complejidad O(log n).

Su implementación sería la siguiente:

```python
def posicion(A,elem,inicio,fin):
    """ list, object, int, int -> int
        OBJ: Posicion en una lista entre 2 limites, None si no esta
             Utiliza el algoritmo de busqueda binaria recursiva
        PRE: lista ordenada
    """
    if inicio > fin: pos = None
    else:
      centro = (inicio + fin) // 2
      if A[centro] == elem: pos= centro
      elif elem < A[centro]: pos = posicion(A,elem,inicio,centro-1)
      else: pos = posicion(A,elem,centro+1,fin)
    return pos
```

Métodos de Ordenación

El método de la Burbuja

El método posiblemente más sencillo conceptualmente, compara pares de elementos adyacentes y los intercambia si están en orden incorrecto. Repite el proceso hasta que la lista esté ordenada. Es fácil de entender, pero muy ineficiente: $O(n^2)$.

```python
def ordenar_burbuja(v):
    """ lista -> None
        OBJ: Ordena una lista completa ascendentemente.
             Metodo burbuja. Modifica la lista.
    """
    long = len(v) - 1
    # haremos long - 1 pasadas: 10 elementos -> 9 pasadas
    for pasada in range(0,long):
        # en cada pasada comparamos el elemento i con el i+1
        for i in range (0,long-pasada):
        # cambiamos de posicion si el izquierdo es mayor,
            para ordenar ascendente
             if (v[i] > v[i+1]): v[i], v[i+1] = v[i+1], v[i]
```

Selección

El método de búsqueda por selección busca el menor elemento en cada pasada y lo coloca en su posición correcta. También tiene complejidad $O(n^2)$, pero realiza menos intercambios que el Burbuja.

Una posible implementación sería la siguiente:

```
def ordenar_seleccion(A):
    """ lista -> None
        OBJ: Ordena ascendentemente una lista.
             Metodo de seleccion. Modifica la lista.  """
    for inicio_pasada in range(len(A)):
        # Encuentra el menor de la parte desordenada
        posicion_menor = inicio_pasada
        for i in range(inicio_pasada + 1, len(A)):
            if A[i] < A[posicion_menor]:
                posicion_menor = i
        # Intercambia el primero con el menor
        A[inicio_pasada], A[posicion_menor] =
                    A[posicion_menor], A[inicio_pasada]
```

Inserción Directa

Este método toma cada elemento y lo inserta en la posición correcta dentro de la parte ya ordenada. Es similar al modo en que ordenamos las cartas en una mano. Resulta eficiente para listas pequeñas o casi ordenadas. Complejidad $O(n^2)$ en el peor caso.

Una posible implementación sería:

```
def insercion(A):
    """ lista -> None
        OBJ: Ordena ascendentemente una lista.
             Metodo de insercion. Modifica la lista.  """
    for i in range(1, len(A)):
        clave = A[i]
        # Mover elementos de la zona A[0..i-1], mayores que
        # la clave una posicion mas a la derecha
        j = i-1
        while j >=0 and clave < A[j] :
                A[j+1] = A[j]
                j -= 1
        # insertar el elemento 'clave' en la posicion que le
            corresponde
        A[j+1] = clave
```

Bibliografía

Severance, C. (2016). *Python for Informatics: Exploring Information*. Online gratis en: https://do1.dr-chuck.net/py4inf/EN-us/book.pdf

Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2ª ed.). Green Tea Press. Online gratis en: https://www.greenteapress.com/thinkpython/thinkCSpy.pdf

Lutz, M. (2014). *Python Pocket Reference: Python in Your Pocket* (5ª ed.). O'Reilly Media.

Matthes, E. (2024). *Curso intensivo de Python* (3ª ed.). Anaya Multimedia.

Peña Ros, R. (2016). *Resolución de problemas para ingenieros con Python estructurado*. Marcombo.

Tema I - Introducción

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Septiembre de 2025



Licencia

© 2025

Algunos derechos reservados.

- Isaac Lozano Osorio (isaac.lozano@urjc.es)
- Salvador Sánchez Alonso (salvador.sanchez@urjc.es)

Este documento se distribuye bajo la licencia Compartir Igual 4.0 (CC BY-SA 4.0)



Contenidos

- Problemas
- Algoritmos
- Lenguajes
- Pensamiento computacional
- Elementos de un programa
- Resumen, referencias y lecturas recomendadas

Problemas y algoritmos

Resolver problemas

- **Problema**: enunciado $\rightarrow$ estrategia de resolución $\rightarrow$ solución
- Un algoritmo es un método para resolver un problema
  - Al-Khwārizmī: matemático persa (siglo IX) que enunció reglas paso a paso para la resolución de operaciones aritméticas de números decimales
- Uno de los **objetivos fundamentales** de esta asignatura es aprender a diseñar algoritmos



Figura: Al-Khwārizmī

Características de un algoritmo

- **Finitud**: el número de instrucciones debe ser finito
- **Instrucciones realizables**: el procesador debe ser capaz de comprenderlas y llevarlas a cabo
- **No ambigüedad**: las instrucciones del algoritmo deben estar expresadas de manera inequívoca en lo referente a:
  - El orden en que deben llevarse a cabo
  - El contenido de las mismas
- **Determinismo**: partiendo de unos mismos datos, un algoritmo debe producir siempre los mismos resultados.

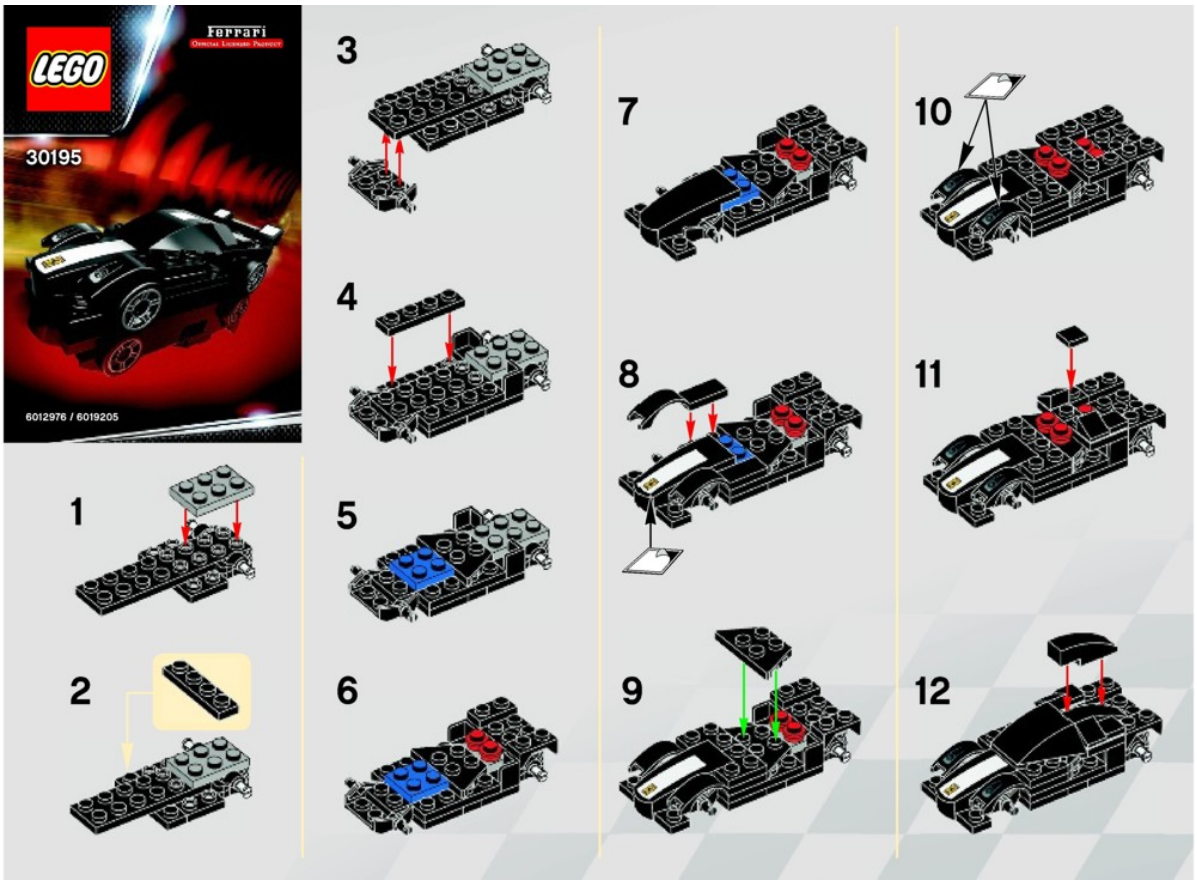
Algoritmo

- Una receta es un algoritmo, pues describe cómo llevar a cabo un cierto trabajo, en este caso, la elaboración de una **Tortilla francesa**
  - Cascar un huevo de gallina en un plato
  - Batir el huevo con un tenedor
  - Calentar 10cc de aceite de oliva en una sartén
  - Si el aceite está caliente, verter el contenido del plato
  - Darle la forma deseada volteando el huevo con un tenedor
  - Cuando haya terminado, quitar la sartén del fuego

Algoritmos para humanos

- Otro algoritmo (muy popular)...
  - Estirar el brazo izquierdo
  - Estirar el brazo derecho
  - Tocar el hombro derecho con brazo izquierdo
  - Tocar el hombre izquierdo con bazo derecho
  - Poner la mano izquierda en la nuca
  - Poner la mano derecha en la nunca
  - Poner la mano izquierda en la cadera
  - Poner la mano izquierda en la caldera
  - Giro completo de la cadera
  - Saltar y girar 90° a la derecha

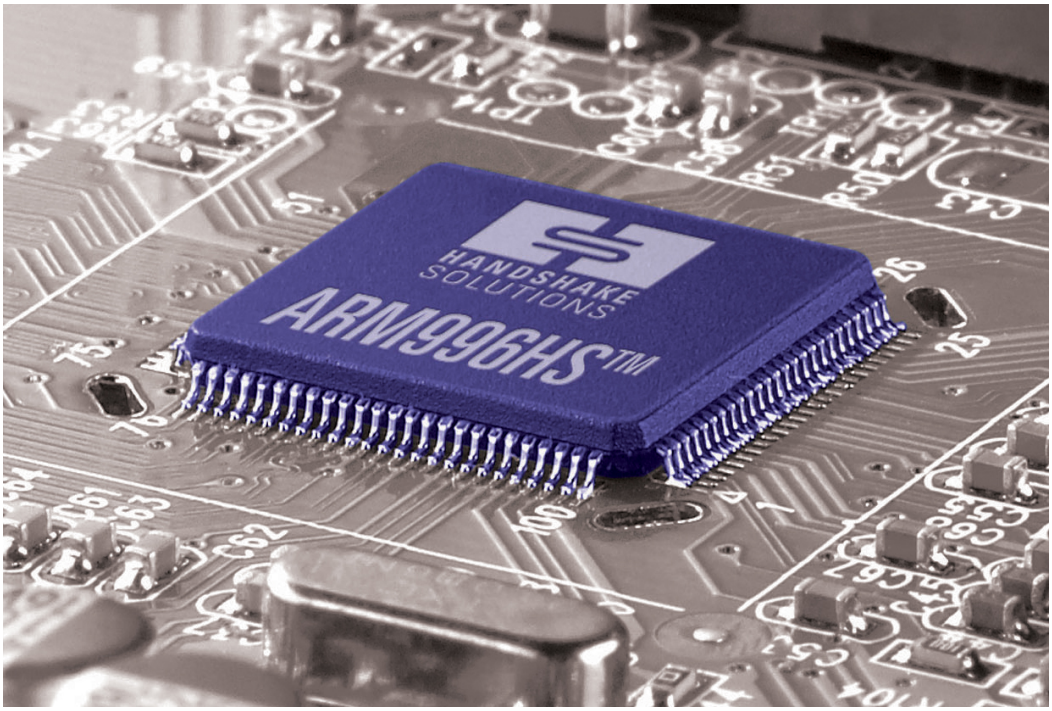
Algoritmos para humanos



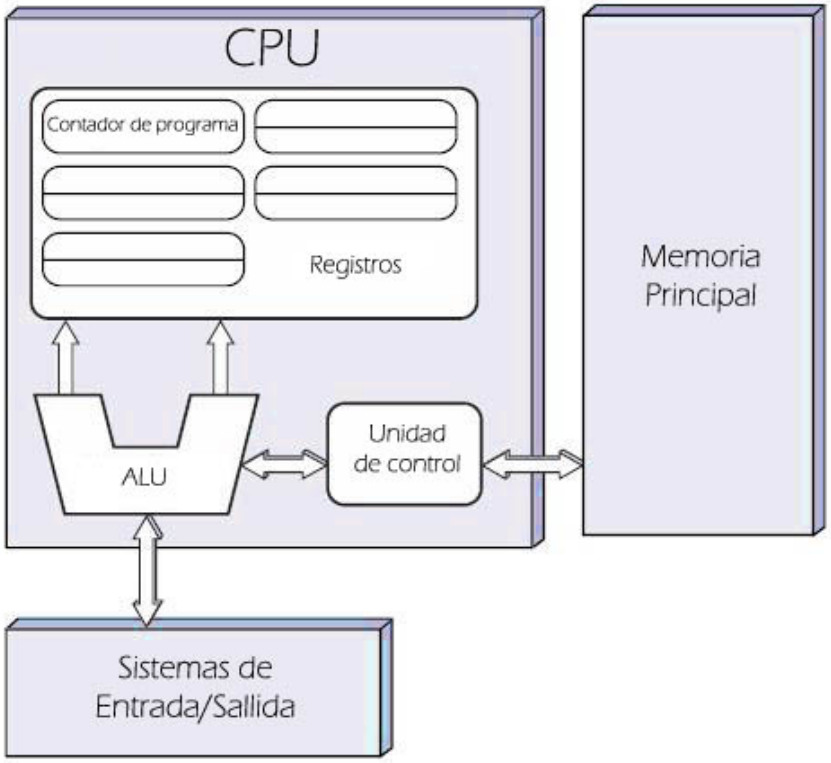
Procesadores

- Un **procesador** es toda entidad capaz de comprender un algoritmo y llevar a cabo el trabajo indicado en el mismo
  - En los dos primeros ejemplos, una persona que sepa leer español y que disponga de los utensilios necesarios
  - En el ejemplo de Lego, cualquiera que sepa interpretar los diagramas
- El **microprocesador** de un ordenador es un procesador capaz de ejecutar instrucciones mediante operaciones simples y accesos a la memoria del ordenador.
  - Las instrucciones deben ser escritas en un lenguaje específico

Un microprocesador



Esquema simplificado de un ordenador



Resolución de problemas con ordenador

- Ayudado por un ordenador, la resolución de un problema consiste en:

Resolución de problemas con ordenador

- Ayudado por un ordenador, la resolución de un problema consiste en:
  - Analizar el problema
  - Diseñar una estrategia de resolución o algoritmo
  - Resolver el algoritmo en el ordenador

Resolución de problemas con ordenador

- Ayudado por un ordenador, la resolución de un problema consiste en:
  - Analizar el problema
  - Diseñar una estrategia de resolución o algoritmo
  - Resolver el algoritmo en el ordenador
- Se trata de un proceso iterativo e incremental

Resolución de problemas con ordenador

- Ayudado por un ordenador, la resolución de un problema consiste en:
  - Analizar el problema
  - Diseñar una estrategia de resolución o algoritmo
  - Resolver el algoritmo en el ordenador
- Se trata de un proceso iterativo e incremental
- No termina con la entrega del programa: mantenimiento

Resolución de problemas con ordenador

- **Análisis**: ¿qué datos necesito para resolverlo? ¿qué información ha de producir su resolución?

Resolución de problemas con ordenador

- **Análisis**: ¿qué datos necesito para resolverlo? ¿qué información ha de producir su resolución?
- **Diseño**: Descomponer el problema en problemas más simples. Especificar los pasos para cada subproblema.

Resolución de problemas con ordenador

- **Análisis**: ¿qué datos necesito para resolverlo? ¿qué información ha de producir su resolución?
- **Diseño**: Descomponer el problema en problemas más simples. Especificar los pasos para cada subproblema.
- **Resolución**: Codificación del algoritmo en un lenguaje de programación. Ejecución del programa y comprobación.

La programación de ordenadores: lenguajes, depuración y pensamiento computacional

Lenguajes de programación

- Un **lenguaje de programación** es un lenguaje formal concebido para comunicar instrucciones a un ordenador.

Lenguajes de programación

- Un **lenguaje de programación** es un lenguaje formal concebido para comunicar instrucciones a un ordenador.
- Un **programa** es la representación de un algoritmo en un lenguaje de programación que permite ejecutar dicho algoritmo en un ordenador.

Lenguajes de programación

- Un **lenguaje de programación** es un lenguaje formal concebido para comunicar instrucciones a un ordenador.
- Un **programa** es la representación de un algoritmo en un lenguaje de programación que permite ejecutar dicho algoritmo en un ordenador.
- La representación se puede hacer a varios niveles:
  - **Lenguaje máquina** (código binario: por ejemplo el conjunto de instrucciones para los microprocesadores de la familia x86).
  - **Lenguaje ensamblador** (por ejemplo, la sintaxis Intel para el conjunto de instrucciones x86).
  - **Lenguajes de alto nivel** (por ejemplo, C, Java, Python, etc.).

Antes del Código máquina

- Programación de las instrucciones "directamente" mediante cables
- Sustitución de cables e interruptores por "ceros y unos": almacenamiento externo (tarjetas, cintas magnetofónicas, discos magnéticos)
- Lenguaje máquina

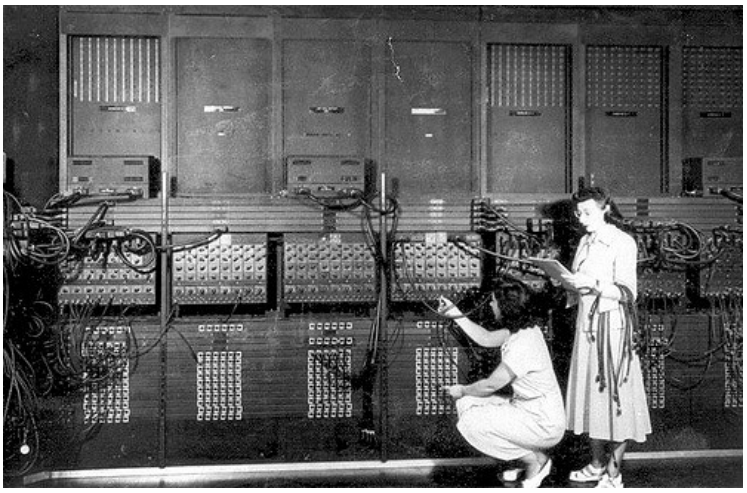
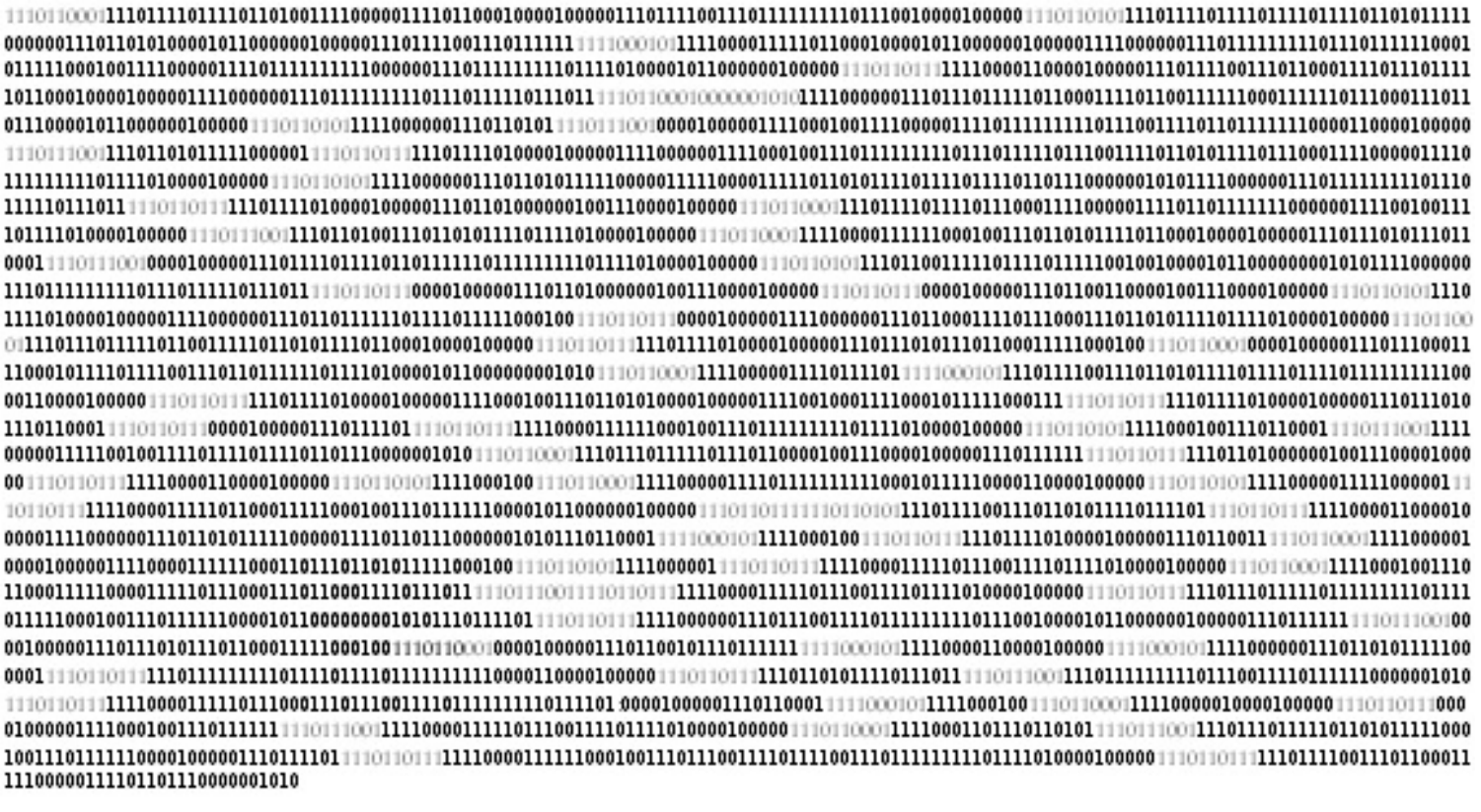


Figura: Las chicas del Eniac (the ENIAC six)

Código máquina



Ensamblador

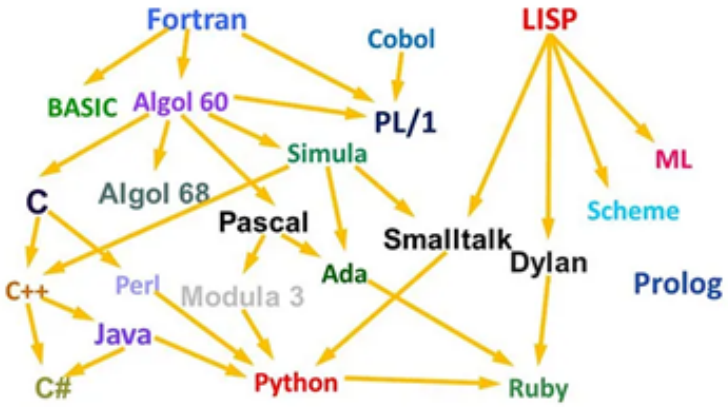
```
008548ED: jmp   854927H
008548EF: cmp   esi,18H
008548F2: jne   8548E8H
008548F4: cmp   byte [edi][48H],0
008548F8: jne   854927H
008548FA: mov   eax,edi
008548FC: call  near 855118H
00854901: mov   eax,[eax][33H]
00854904: je    854927H
00854907: and   eax,00000200
0085490C: cmp   eax,0
0085490F: je    854927H
00854911: push  0
00854913: lea   ebx,[edi]
00854915: lea   ecx,[edi][4]
00854918: mov   eax,00000000
```

Lenguajes de programación de alto nivel

- La forma de expresar las órdenes es más parecida al modo en que se expresan los humanos

Lenguajes de programación de alto nivel

- La forma de expresar las órdenes es más parecida al modo en que se expresan los humanos
- Desde la aparición de Fortran en 1954, se han contado centenares de lenguajes de alto nivel.
- No todos se usan: Muchos de ellos están "en peligro de extinción".



Un ejemplo

```c
/* Un programa sencillo en lenguaje C */
#include <stdio.h>
int main(void)
{
   printf("En un lugar de la Mancha, de cuyo nombre no
       quiero acordarme...");
   return 0;
}
```

Un ejemplo

```
#Un programa sencillo en python
print("En un lugar de la Mancha,")
print("de cuyo nombre no quiero acordarme,")
print("no ha mucho tiempo que vivia un hidalgo...")
```

Cómo se expresan los algoritmos

- Lenguaje natural.
- Pseudocódigo.
- Diagramas de flujo.
- Lenguaje de programación.
- Otros modos: tablas de control, lenguajes formales.

Lenguaje natural

- Problema: Calcular el máximo común divisor de dos enteros positivos.

Lenguaje natural

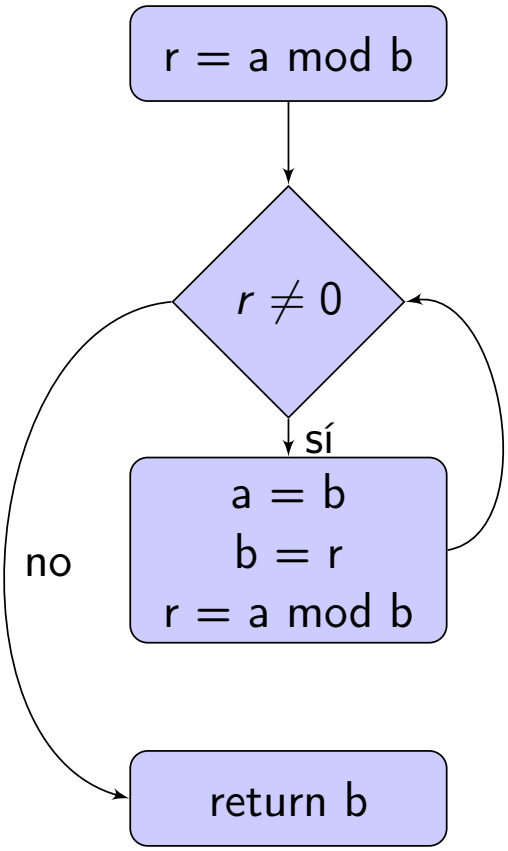
- Problema: Calcular el máximo común divisor de dos enteros positivos.
- Algoritmo (Euclides): *Tomar los dos enteros, y calcular un nuevo par formado por el menor de ellos y el resto de la división entera de dividir el mayor entre el menor. Repetir el proceso hasta que los números en el par sean el mismo o el resto de la división sea cero. El resultado será el divisor de la última división*.

Pseudocódigo

Algorithm 1 Algoritmo de Euclides

Requiere: $a \geq 0$ and $b \geq 0$ and $a > b$ ▷ Precondición
  procedimiento Euclides(a, b) ▷ Calcula el m.c.d. de a y b
    $r \leftarrow a$ mód b
    mientras $r \neq 0$ **hacer** ▷ Hemos terminado si r es 0
      $a \leftarrow b$
      $b \leftarrow r$
      $r \leftarrow a$ mód b
    fin mientras
    devolver b ▷ El m.c.d es b
  fin procedimiento

Diagrama de flujo



Lenguaje de alto nivel (Python)

```
# Calculo del MCD mediante el algoritmo de Euclides
a = 42
b = 56
r = a % b
while r != 0 :
    a = b
    b = r
    r = a % b
print(b)
```

Pulsa aquí para ejecutar en PythonTutor

Errores en los algoritmos

- Revisando el algoritmo visto anteriormente...
  - Estirar el brazo izquierdo
  - Estirar el brazo derecho
  - Tocar el hombro derecho con brazo izquierdo
  - Tocar el hombre izquierdo con bazo derecho
  - Poner la mano izquierda en la nuca
  - Poner la mano derecha en la nunca
  - Poner la mano izquierda en la cadera
  - Poner la mano izquierda en la caldera
  - Giro completo de la cadera
  - Saltar y girar 90° a la derecha

Errores en los algoritmos

- Revisando el algoritmo visto anteriormente...
  - Estirar el brazo izquierdo
  - Estirar el brazo derecho
  - Tocar el hombro derecho con brazo izquierdo
  - Tocar el hombr**e** izquierdo con **ba**zo derecho
  - Poner la mano izquierda en la nuca
  - Poner la mano derecha en la nu**n**ca
  - Poner la mano izquierda en la cadera
  - Poner la mano izquierda en la ca**l**dera
  - Giro completo de la cadera
  - Saltar y girar 90° a la derecha

Errores en los algoritmos

- Revisando el algoritmo visto anteriormente...
  - Estirar el brazo izquierdo
  - Estirar el brazo derecho
  - Tocar el hombro derecho con brazo izquierdo
  - Tocar el hombro izquierdo con brazo derecho
  - Poner la mano izquierda en la nuca
  - Poner la mano derecha en la nuca
  - Poner la mano izquierda en la cadera
  - Poner la mano izquierda en la cadera
  - Giro completo de la cadera
  - Saltar y girar 90° a la derecha

Errores en los algoritmos

- Revisando el algoritmo visto anteriormente...
  - Estirar el brazo izquierdo
  - Estirar el brazo derecho
  - Tocar el hombro derecho con brazo izquierdo
  - Tocar el hombro izquierdo con brazo derecho
  - Poner la mano izquierda en la nuca
  - Poner la mano derecha en la nuca
  - Poner la mano izquierda en la cadera
  - Poner la mano izquierda en la cadera
  - Giro completo de la cadera
  - Saltar y girar 90° a la derecha
- Se trata de errores que se detectan en el momento de la traducción (**tiempo de compilación**)

Bug o error

Definición

Bug: *error o fallo en un programa de computador o sistema de software que desencadena un resultado indeseado [de Wikipedia]*



Errores en los algoritmos

- Revisando el algoritmo visto anteriormente...
  - Estirar el brazo izquierdo
  - Estirar el brazo derecho
  - Tocar el hombro derecho con brazo izquierdo
  - Tocar el hombro izquierdo con brazo derecho
  - Poner la mano izquierda en la nuca
  - Poner la mano derecha en la nuca
  - Poner la mano izquierda en la cadera
  - Poner la mano izquierda en la cadera
  - Giro completo de la cadera
  - Saltar y girar 90° a la derecha
- Se trata de errores que se detectan durante la ejecución del programa (**tiempo de ejecución**)
- Muy común su forma en inglés: *bugs*

Programador vs. usuario

- El **programador** escribe un programa en un lenguaje de programación siguiendo una estrategia de resolución (algoritmo)

Programador vs. usuario

- El **programador** escribe un programa en un lenguaje de programación siguiendo una estrategia de resolución (algoritmo)
- Para un **usuario** todo es más fácil: solo tiene que elegir entre los varios programas listos para usarse que alguien puso a su alcance



Pensar computacionalmente



- Pensar **computacionalmente** es saber expresar la solución a los problemas de una forma que permita su resolución con un ordenador.
  - Implica (a) idear un algoritmo y (b) codificarlo como un programa
- Al traducir un algoritmo a un programa, aparecen **limitaciones prácticas**, como el valor máximo de un número entero:
  - En C el máximo valor positivo de un entero (`MAX_INT`) depende de la plataforma, si bien como mínimo es 32.767 (16 bits)
  - En Java, es en todos los casos 2.147.483.647 (32 bits)
  - En Python no hay un límite al respecto

Elementos de un programa

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa
- **Constantes**: valores fijos que no cambian durante la ejecución

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa
- **Constantes**: valores fijos que no cambian durante la ejecución
- **Variables**: elementos cuyo valor puede cambiar

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa
- **Constantes**: valores fijos que no cambian durante la ejecución
- **Variables**: elementos cuyo valor puede cambiar
- **Expresiones**: combinaciones de operandos y operadores que se evalúan dando como resultado un valor

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa
- **Constantes**: valores fijos que no cambian durante la ejecución
- **Variables**: elementos cuyo valor puede cambiar
- **Expresiones**: combinaciones de operandos y operadores que se evalúan dando como resultado un valor
- **Palabras reservadas**: palabras con un significado especial en el lenguaje de programación utilizado (ej. `while`, `if`, `for`...)

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa
- **Constantes**: valores fijos que no cambian durante la ejecución
- **Variables**: elementos cuyo valor puede cambiar
- **Expresiones**: combinaciones de operandos y operadores que se evalúan dando como resultado un valor
- **Palabras reservadas**: palabras con un significado especial en el lenguaje de programación utilizado (ej. `while`, `if`, `for`...)
- **Texto no ejecutable**: Texto complementario que bien no se ejecuta o bien lo hace cuando se traduce el programa

Elementos de un programa

- **Instrucciones**: cada una de las órdenes que componen el programa
- **Constantes**: valores fijos que no cambian durante la ejecución
- **Variables**: elementos cuyo valor puede cambiar
- **Expresiones**: combinaciones de operandos y operadores que se evalúan dando como resultado un valor
- **Palabras reservadas**: palabras con un significado especial en el lenguaje de programación utilizado (ej. `while`, `if`, `for`...)
- **Texto no ejecutable**: Texto complementario que bien no se ejecuta o bien lo hace cuando se traduce el programa
- **Comentarios**: nunca, sirven para documentar el programa

Instrucciones

- Una secuencia de instrucciones para un ordenador escrita en un cierto lenguaje de programación.

Instrucciones

- Una secuencia de instrucciones para un ordenador escrita en un cierto lenguaje de programación.
  - Las instrucciones de un programa y los datos necesarios para ejecutarlo se cargan en la memoria del ordenador.

Instrucciones

- Una secuencia de instrucciones para un ordenador escrita en un cierto lenguaje de programación.
  - Las instrucciones de un programa y los datos necesarios para ejecutarlo se cargan en la memoria del ordenador.
  - El microprocesador va realizando los cálculos correspondientes a las diferentes instrucciones según los datos disponibles.

Instrucciones

- Una secuencia de instrucciones para un ordenador escrita en un cierto lenguaje de programación.
  - Las instrucciones de un programa y los datos necesarios para ejecutarlo se cargan en la memoria del ordenador.
  - El microprocesador va realizando los cálculos correspondientes a las diferentes instrucciones según los datos disponibles.
- Las instrucciones o sentencias se estructuran en líneas

Instrucciones

- Una secuencia de instrucciones para un ordenador escrita en un cierto lenguaje de programación.
  - Las instrucciones de un programa y los datos necesarios para ejecutarlo se cargan en la memoria del ordenador.
  - El microprocesador va realizando los cálculos correspondientes a las diferentes instrucciones según los datos disponibles.
- Las instrucciones o sentencias se estructuran en líneas
- Algunas instrucciones ocupan una línea, otras más:

Instrucciones

- Una secuencia de instrucciones para un ordenador escrita en un cierto lenguaje de programación.
  - Las instrucciones de un programa y los datos necesarios para ejecutarlo se cargan en la memoria del ordenador.
  - El microprocesador va realizando los cálculos correspondientes a las diferentes instrucciones según los datos disponibles.
- Las instrucciones o sentencias se estructuran en líneas
- Algunas instrucciones ocupan una línea, otras más:

```
a = 4          # linea 1: sentencia de asignacion
b = a + 12     # linea 2: asignacion con expresion
if a < b:      # lineas 3 a 6: sentencia alternativa (if)
   print("a es mayor")
else:
   print("b es mayor")
```

Flujo de un programa

- Durante su ejecución, un programa va pasando por los diferentes pasos que le ha especificado el programador.

Flujo de un programa

- Durante su ejecución, un programa va pasando por los diferentes pasos que le ha especificado el programador.
- Algunos pasos son secuenciales: el programa pasa de un paso al siguiente.

Flujo de un programa

- Durante su ejecución, un programa va pasando por los diferentes pasos que le ha especificado el programador.
- Algunos pasos son secuenciales: el programa pasa de un paso al siguiente.
- Otras veces se ejecutan o no en función de una condición

Flujo de un programa

- Durante su ejecución, un programa va pasando por los diferentes pasos que le ha especificado el programador.
- Algunos pasos son secuenciales: el programa pasa de un paso al siguiente.
- Otras veces se ejecutan o no en función de una condición
- Hay pasos que se repiten varias veces

¡Identifica una instrucción de cada tipo en la receta de la tortilla!

Tipos de instrucciones en un algoritmo

- Cascar un huevo de gallina en un plato
- Batir el huevo con un tenedor
- Calentar 10cc de aceite de oliva en una sartén
- Si el aceite está caliente, verter el contenido del plato
- Darle la forma deseada volteando el huevo con un tenedor
- Cuando haya terminado, quitar la sartén del fuego

Tipos de instrucciones en un algoritmo

- Cascar un huevo de gallina en un plato [sec]
- Batir el huevo con un tenedor [rep]
- Calentar 10cc de aceite de oliva en una sartén [rep]
- Si el aceite está caliente, verter el contenido del plato [cond]
- Darle la forma deseada volteando el huevo con un tenedor [rep]
- Cuando haya terminado, quitar la sartén del fuego [sec]

Variable

Definición

*Una **variable** es un elemento del programa cuyo valor puede cambiar durante la ejecución del mismo*

Variable

Definición

*Una **variable** es un elemento del programa cuyo valor puede cambiar durante la ejecución del mismo*

- Dicho nombre (identificador) puede utilizarse para almacenar un dato y luego recuperarlo

Variable

Definición

*Una **variable** es un elemento del programa cuyo valor puede cambiar durante la ejecución del mismo*

- Dicho nombre (identificador) puede utilizarse para almacenar un dato y luego recuperarlo
- El nombre de las variables lo elige el programador

Variable

Definición

*Una **variable** es un elemento del programa cuyo valor puede cambiar durante la ejecución del mismo*

- Dicho nombre (identificador) puede utilizarse para almacenar un dato y luego recuperarlo
- El nombre de las variables lo elige el programador
- El dato almacenado puede modificarse

Experimentemos un poco...

Vamos a buscar juntos el mayor número de una serie de varios...

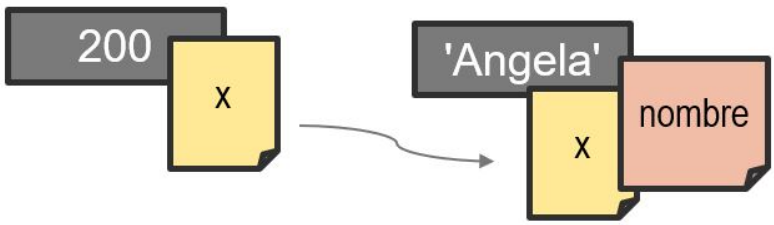
Variables en Python

- En la mayoría de lenguajes (C, Java...) las variables en memoria son algo así como una caja con nombre en la que se pueden meter y sacar valores del tipo declarado
- En Python las variables se parecen más a notas adhesivas que se pueden asociar (pegar) sobre un valor existente pero que también puede luego despegar y asociar a otro valor del mismo o distinto tipo

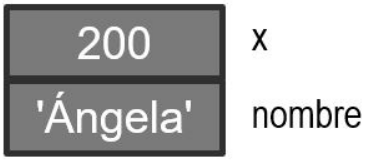




Variables en Python



Python (nota adhesiva)



C/Java (contenedor)

Constante

Definición

*Una **constante** es un elemento cuyo valor no cambia durante el desarrollo del algoritmo (durante la ejecución del programa).*

Constante

Definición

*Una **constante** es un elemento cuyo valor no cambia durante el desarrollo del algoritmo (durante la ejecución del programa).*

- Ejemplos: PI = 3.141592... , E = 2.71828, LADO = 10, etc.

Constante

Definición

*Una **constante** es un elemento cuyo valor no cambia durante el desarrollo del algoritmo (durante la ejecución del programa).*

- Ejemplos: PI = 3.141592... , E = 2.71828, LADO = 10, etc.
- No sólo numéricas, también textos (`string`):
  - WARNING = 'Prohibido el paso', SALUDO = "Hola"
  - Más de una línea: poniendo una barra inclinada (\) tras cada parte de la cadena: "En un lugar \ de la Mancha \ de cuyo nombre..."

Constante

Definición

*Una **constante** es un elemento cuyo valor no cambia durante el desarrollo del algoritmo (durante la ejecución del programa).*

- Ejemplos: PI = 3.141592... , E = 2.71828, LADO = 10, etc.
- No sólo numéricas, también textos (`string`):
  - WARNING = 'Prohibido el paso', SALUDO = "Hola"
  - Más de una línea: poniendo una barra inclinada (\) tras cada parte de la cadena: "En un lugar \ de la Mancha \ de cuyo nombre..."
- Python no tiene constantes en el sentido estricto de la palabra: hay que definir una variable y mantener la disciplina de evitar operaciones de asignación sobre ella

Tipo de datos

Definición

*Un **tipo de datos** es un conjunto de datos cuyos valores tienen características comunes y predefinidas.*

- Determina los valores posibles, las operaciones permitidas, el significado de los datos y la forma en que se almacenan los valores en la memoria del ordenador.
- Tipos básicos:
  - Numéricos: enteros, reales, complejos...
  - Texto: *strings* o cadenas de caracteres
  - Lógicos: booleanos
- Tipos derivados: arrays, listas, tuplas, registros, diccionarios, clases...
- La instrucción `type` en Python permite conocer el tipo de una expresión.

Tipos de datos en Python

```python
# Numeros
edad = 30                          # int
temperatura = 22.5                 # float
impedancia = 5 + 2j                # complex

# Texto y booleanos
nombre_usuario = "Salvador"        # str
es_profesor = True                 # bool

# Colecciones
estudiantes_2025 = ["Ana", "Luis", "Carlos"]   # list
coordenadas = (40.45, -3.75)                   # tuple
materias = {"Programacion", "Algoritmia"}      # set
notas = {"Ana": 9.5, "Luis": 8.0}              # dict
```

Tipos de datos en Python

- El tipo se tiene en cuenta cuando se solicita una operación, y si la misma no está permitida el intérprete emite un error:

```
a = "Hola"
b = a + 10
# Respuesta:
#>>>Traceback (most recent call last):
#>>>  File "<stdin>", line 1, in <module>
#>>> TypeError: Can't convert 'int' object to str
    implicitly
```

- La función "type" permite conocer el tipo de una variable o expresión

```
x = 100
type(x)
#respuesta:
# <class 'int'>
```

Conversiones de tipo

- Internamente todas las operaciones se realizan entre datos del mismo tipo: a menudo es necesario convertir un dato a otro tipo distinto.
- Conversiones implícitas:
  - Promoción: las conversiones que se hacen para hacer concordar los tipos dentro de una expresión.
  - Las lleva a cabo el traductor, no el programador
- Conversiones explícitas (*casting*):
  - A través de mecanismos del lenguaje: En python las funciones `int()` y `float()` permiten convertir texto en números o números entre sí
  - Indicadas por el programador

Conversiones implícitas y explícitas en Python

- Conversión implícita:

```
numero_entero = 5          # int
numero_decimal = 2.5       # float

resultado = numero_entero + numero_decimal   # Convierte 5 en 5.0

print(resultado)           # 7.5
print(type(resultado))     # <class 'float'>
```

- Conversión explícita:

```
texto = "42"               # str
numero_convertido = int(texto)   # Conversion explicita

print(numero_convertido)         # 42
print(type(numero_convertido))   # <class 'int'>

# Otro ejemplo: convertir float a int
decimal = 3.99
entero = int(decimal)            # Se trunca, no se redondea
print(entero)                    # 3
```

Expresiones

Definición

*Una **expresión** es una combinación de constantes, variables, símbolos de operación, paréntesis, funciones especiales y otros elementos que se evalúa produciendo un valor.*

- Numéricas: `(3+4)*9`, `x-1`, `2**5`

Expresiones

Definición

*Una **expresión** es una combinación de constantes, variables, símbolos de operación, paréntesis, funciones especiales y otros elementos que se evalúa produciendo un valor.*

- Numéricas: `(3+4)*9`, `x-1`, `2**5`
- Texto: `"Hola"+ "Amigo"`

Expresiones

Definición

*Una **expresión** es una combinación de constantes, variables, símbolos de operación, paréntesis, funciones especiales y otros elementos que se evalúa produciendo un valor.*

- Numéricas: `(3+4)*9`, `x-1`, `2**5`
- Texto: `"Hola"+ "Amigo"`
- Lógicas: $(a > 5)$ and $(x < 0)$
- Otros tipos de expresiones (más adelante...)
- Las expresiones se interpretan de acuerdo a unas reglas (de precedencia y asociación) específicas de cada lenguaje

Expresiones (cont.)

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

Precedencia de operadores

Definición

Orden en el que un determinado lenguaje lleva a cabo las operaciones

Ejemplo:

```
9 + 4 * 2
```

Precedencia de operadores

Definición

Orden en el que un determinado lenguaje lleva a cabo las operaciones

Ejemplo:

```
9 + 4 * 2
```

- Imprime 17 porque el * tiene más **precedencia** que el operador + y ambos **asocian** de izquierda a derecha.
- Los paréntesis se pueden utilizar para forzar otra interpretación.

Precedencia de operadores

Definición

Orden en el que un determinado lenguaje lleva a cabo las operaciones

Ejemplo:

```
9 + 4 * 2
```

- Imprime 17 porque el * tiene más **precedencia** que el operador + y ambos **asocian** de izquierda a derecha.
- Los paréntesis se pueden utilizar para forzar otra interpretación.

```
(9 + 4) * 2
```

- Imprime 26 porque el paréntesis tiene mayor **precedencia**.

Uso de variables lógicas

Utilizar las constantes “True” y “False” sólo para inicializar, evitando escribir algo como:

```
if (ready == True) ...
while (empty == False) ...
```

Uso de variables lógicas

Utilizar las constantes "True" y "False" sólo para inicializar, evitando escribir algo como:

```
if (ready == True) ...
while (empty == False) ...
```

expresiones que pueden siempre reemplazarse por:

```
if (ready) ...
while (not empty) ...
```

... lo cual tiene sentido al traducirse y leerse en voz alta

Uso de variables lógicas

Cuando elegimos nombres para las variables lógicas deben evitarse nombres "negativos" para facilitar la legibilidad. Así, es mejor usar:

```
if (completo) ...
if (not completo) ...
```

... evitando los más confusos:

```
if (not noCompleto) ...
if (noCompleto) ...
```

Uso de variables lógicas

Utilizar las variables booleanas para recoger el valor de una expresión, sin usar `ifs` innecesarios. Así, en lugar de:

```
if condicion:
  booleano = True
else:
  booleano = False
```

debe escribirse:

```
booleano = condicion
```

Nombres de variable en Python

Python permite utilizar **cualquier conjunto de letras, números y subrayados "_"** como nombre de variable, con algunas limitaciones:

- Deben comenzar por letra o subrayado: `contador`, `_index`, etc.

Nombres de variable en Python

Python permite utilizar **cualquier conjunto de letras, números y subrayados "_"** como nombre de variable, con algunas limitaciones:

- Deben comenzar por letra o subrayado: `contador`, `_index`, etc.
- Python distingue entre mayúsculas y minúsculas: no es la misma variable "`Valor`" que "`valor`"

Nombres de variable en Python

Python permite utilizar **cualquier conjunto de letras, números y subrayados "_"** como nombre de variable, con algunas limitaciones:

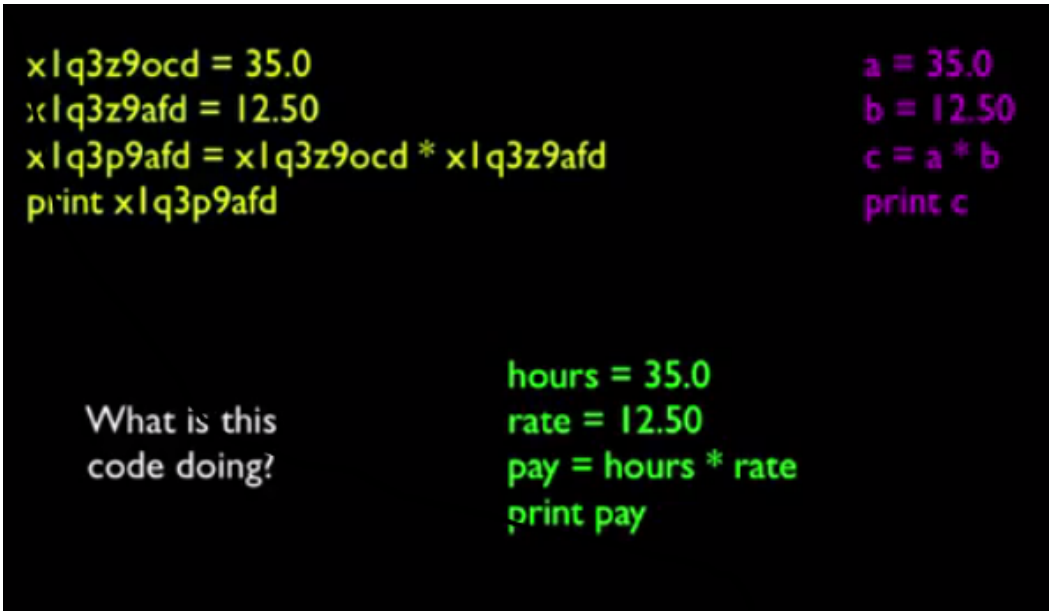
- Deben comenzar por letra o subrayado: `contador`, `_index`, etc.
- Python distingue entre mayúsculas y minúsculas: no es la misma variable "`Valor`" que "`valor`"
- No se permite el uso de símbolos tales como !, @, #, $, %, etc.
- No está permitido usar palabras reservadas: `and`, `assert`, `break`, `class`, `continue`, `def`, `if`...

Nombres de variable en Python

Python permite utilizar **cualquier conjunto de letras, números y subrayados "_"** como nombre de variable, con algunas limitaciones:

- Deben comenzar por letra o subrayado: `contador`, `_index`, etc.
- Python distingue entre mayúsculas y minúsculas: no es la misma variable "`Valor`" que "`valor`"
- No se permite el uso de símbolos tales como !, @, #, $, %, etc.
- No está permitido usar palabras reservadas: `and`, `assert`, `break`, `class`, `continue`, `def`, `if`...
- Deben evitarse símbolos exclusivos de un idioma (vocales con tilde, eñe, etc.) que puedan visualizarse incorrectamente en otros entornos.

Nombres significativos de variable



Reglas de estilo en Python

- Se deben usar nombres de variables descriptivos, no ambiguos y que definan claramente lo que contienen.

Reglas de estilo en Python

- Se deben usar nombres de variables descriptivos, no ambiguos y que definan claramente lo que contienen.
  - Por ejemplo para almacenar el número de alumnos no uses `n = 20`, sino, `num_alumnos = 20`.
- Pueden usarse mayúsculas y minúsculas, pero por convenio se suelen usar solo las minúsculas

Reglas de estilo en Python

- Se deben usar nombres de variables descriptivos, no ambiguos y que definan claramente lo que contienen.
  - Por ejemplo para almacenar el número de alumnos no uses `n = 20`, sino, `num_alumnos = 20`.
- Pueden usarse mayúsculas y minúsculas, pero por convenio se suelen usar solo las minúsculas
- Se suele usar el guión bajo "_" para separar palabras: `alumnos_aprobados`

Entrada/Salida en Python

- La función `input` permite solicitar un valor al usuario
- Lo habitual es almacenar dicho valor en una variable

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

- La función habitual de salida es `print()`
- Para mostrar varios elementos, separar por comas:

```
print("Encantado, ", nombre, "yo soy Python :)")
```

- Operaciones con cadenas: + (concatena), * (concatenación múltiple)

```
print("Hola "*3) # La salida es Hola Hola Hola
```

- Más sobre `print`: https://realpython.com/python-f-strings/

Comentarios

Información que el compilador o intérprete ignorará, pero importante para:

- documentar y clarificar nuestro código
- identificar el autor, fecha, ultimas modificaciones, etc.
- desactivar líneas de código (tal vez sólo temporalmente)

```
# Get the name of the file and open it
name = raw_input("Enter file:")
handle = open(name, "r")
text = handle.read()
words = text.split()

# Count word frequency
counts = dict()
for word in words:
   counts[word] = counts.get(word,0) + 1

# Find the most common word
bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

# All done
print bigword, bigcount
```

Comentarios en Python

- Sólo existen los comentarios "unilinea" (#)
- Bien preceden a un bloque de código o bien comentan una línea al final de la misma
- OJO: las *docstrings* (" " ") no es forma de comentar
  - deben usarse unicamente para documentar código al inicio de la pieza
  - se muestran en la documentación al usar p.e. `help`

```
def es_anterior(fecha1,fecha2):
  """ Fecha, Fecha -> bool
  OBJ: Calcula si fecha1 es anterior a fecha2  """
    if (fecha2.anno > fecha1.anno):
        return True
    elif (fecha1.anno > fecha2.anno):
        return False
    else:      # si los dos annos son iguales
        if (fecha2.mes > fecha1.mes):
            ...
            etc.
```

Resumen

- Un algoritmo es una secuencia de pasos para obtener un resultado a partir de entradas bien definidas.
- Un programa es una representación de un algoritmo en un lenguaje de programación.
- Los programas trabajan con valores constantes y variables.
- Los valores tienen un tipo asociado.
- Las expresiones tienen que tener en cuenta qué tipos de datos se pueden combinar.
- La legibilidad del código es importante: deben usarse nombres significativos, así como comentarios para explicar todo aquello que no sea evidente.

Referencias y reconocimientos

- Algunos de los contenidos de esta presentación han sido adaptados de los materiales del curso de "Programming for Everybody (Python)", creado por Charles Severance y disponible en `https://www.coursera.org/course/pythonlearn`.
- También se han utilizado ejemplos y adaptado partes del libro ''Fundamentos de la Programación, 2ed" de Luis Joyanes Aguilar, con permiso expreso del autor.

Lecturas y vídeos recomendados

- Capítulo 1 del libro ''Python for Informatics" de Charles Severance.
- Capítulo 1 del libro ''Fundamentos de la Programación, 2ed" de Luis Joyanes Aguilar.
- Ver el vídeo "Computer Programming | A short interesting film"
- TED talk "You Should Learn to Program"
- Divertida lectura al respecto de los nombres de variable: `http://computationaltales.blogspot.com.es/2011/03/importance-of-variable-names.html`

Para experimentar

Visitar y probar los siguientes enlaces:

- Área de principiantes de programmingbasics.org
- One hour of Code
- Codepad

Tema I - Introducción

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Septiembre de 2025



Tema II: Estructuras de control

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Octubre de 2025



Licencia

© 2025.

Algunos derechos reservados.

- Isaac Lozano Osorio (isaac.lozano@urjc.es)
- Salvador Sánchez Alonso (salvador.sanchez@urjc.es)

Este documento se distribuye bajo la licencia según la licencia Compartir Igual 4.0 (CC BY-SA 4.0)



El flujo de control

- Flujo de control: secuencia de ejecución de las instrucciones del programa.

El flujo de control

- Flujo de control: secuencia de ejecución de las instrucciones del programa.
- En un programa sin estructuras de control el flujo es **secuencial**.

El flujo de control

- Flujo de control: secuencia de ejecución de las instrucciones del programa.
- En un programa sin estructuras de control el flujo es **secuencial**.
- El flujo de control incluye la **activación** de módulos, es decir, la ejecución del código dentro de los mismos cuando se los invoca.

El flujo de control

- Flujo de control: secuencia de ejecución de las instrucciones del programa.
- En un programa sin estructuras de control el flujo es **secuencial**.
- El flujo de control incluye la **activación** de módulos, es decir, la ejecución del código dentro de los mismos cuando se los invoca.
- Para modificar el flujo de control se utilizan las estructuras de control de flujo: **selectivas** y **repetitivas**

Estructuras selectivas

Definición

Una **estructura selectiva** permite, de acuerdo a una condición, ejecutar o no ciertas instrucciones.

- Cuando se requiere actuar de modo diferente ante diferentes entradas o en diferentes estados.

Estructuras selectivas

Definición

Una **estructura selectiva** permite, de acuerdo a una condición, ejecutar o no ciertas instrucciones.

- Cuando se requiere actuar de modo diferente ante diferentes entradas o en diferentes estados.
- Una condición lógica (verdadero/falso) determina qué instrucciones se deben ejecutar.

Estructuras selectivas

Definición

Una **estructura selectiva** permite, de acuerdo a una condición, ejecutar o no ciertas instrucciones.

- Cuando se requiere actuar de modo diferente ante diferentes entradas o en diferentes estados.
- Una condición lógica (verdadero/falso) determina qué instrucciones se deben ejecutar.
  - Operadores condicionales en Python: $==$, $>$, $<$, $<=$, $>=$, $!=$
  - Operadores lógicos: `and`, `or`, `not`

Estructuras selectivas

Definición

Una **estructura selectiva** permite, de acuerdo a una condición, ejecutar o no ciertas instrucciones.

- Cuando se requiere actuar de modo diferente ante diferentes entradas o en diferentes estados.
- Una condición lógica (verdadero/falso) determina qué instrucciones se deben ejecutar.
  - Operadores condicionales en Python: $==$, $>$, $<$, $<=$, $>=$, $!=$
  - Operadores lógicos: `and`, `or`, `not`
- El resultado de la evaluación de la condición (verdadero/falso) lleva a la ejecución de uno o de ningún bloque.

Ejemplo: Par o impar

```
import random
n = random.randint(0,10)
if n %2 == 0:
    print("El numero generado es par")
else:
    print("El numero generado es impar")
```

Sangrado

- Cada vez que una sentencia acaba con dos puntos, Python espera que la sentencia o sentencias que le siguen –bloque de código– aparezcan con un mayor sangrado.

Sangrado

- Cada vez que una sentencia acaba con dos puntos, Python espera que la sentencia o sentencias que le siguen –bloque de código– aparezcan con un mayor sangrado.
- Es la forma de marcar el inicio y el fin de una serie de sentencias que *dependen* de otra.

Sangrado

- Cada vez que una sentencia acaba con dos puntos, Python espera que la sentencia o sentencias que le siguen –bloque de código– aparezcan con un mayor sangrado.
- Es la forma de marcar el inicio y el fin de una serie de sentencias que *dependen* de otra.
- Excepción: si solo hay una sentencia que depende de otra, pueden escribirse ambas en la misma línea.

Consejos sobre sangrado

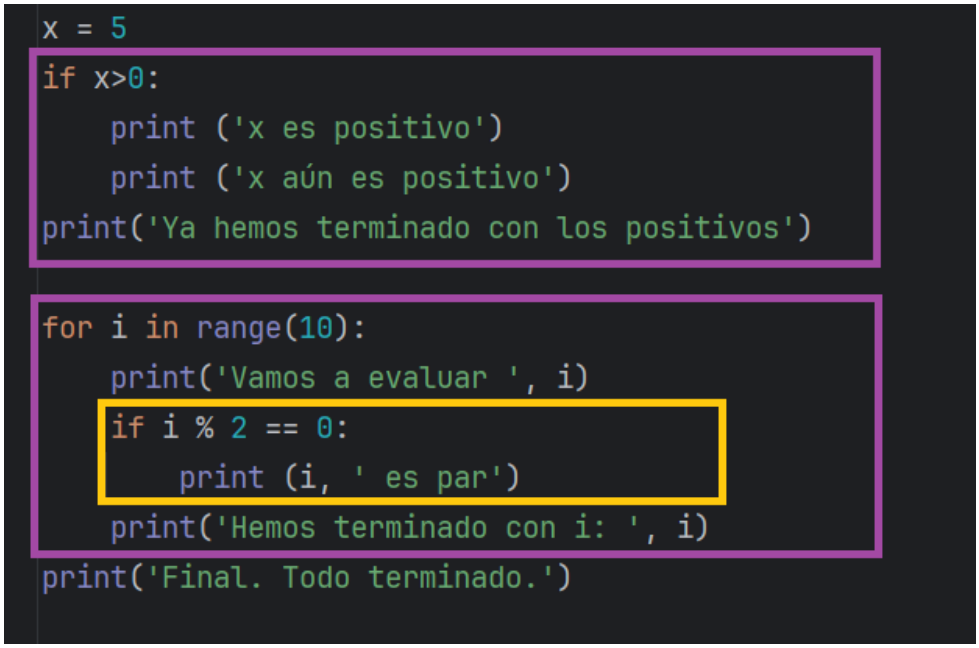
- Líneas en blanco y comentarios no afectan al sangrado

Consejos sobre sangrado

- Líneas en blanco y comentarios no afectan al sangrado
- Eliminar los tabuladores en el editor

Consejos sobre sangrado

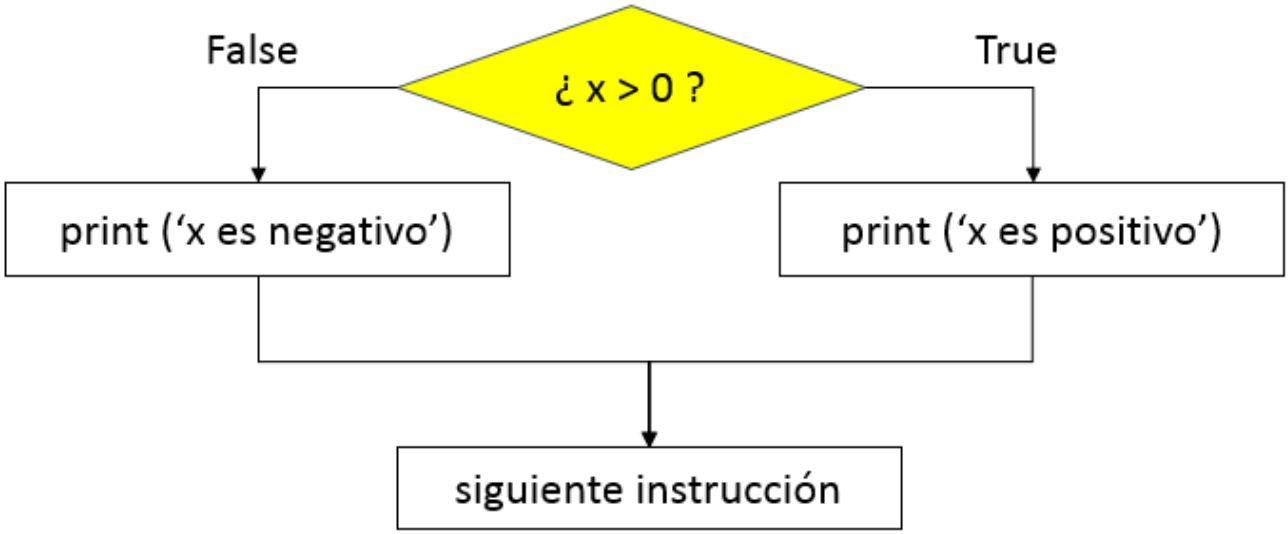
- Líneas en blanco y comentarios no afectan al sangrado
- Eliminar los tabuladores en el editor
- Usar "tablas mentales":



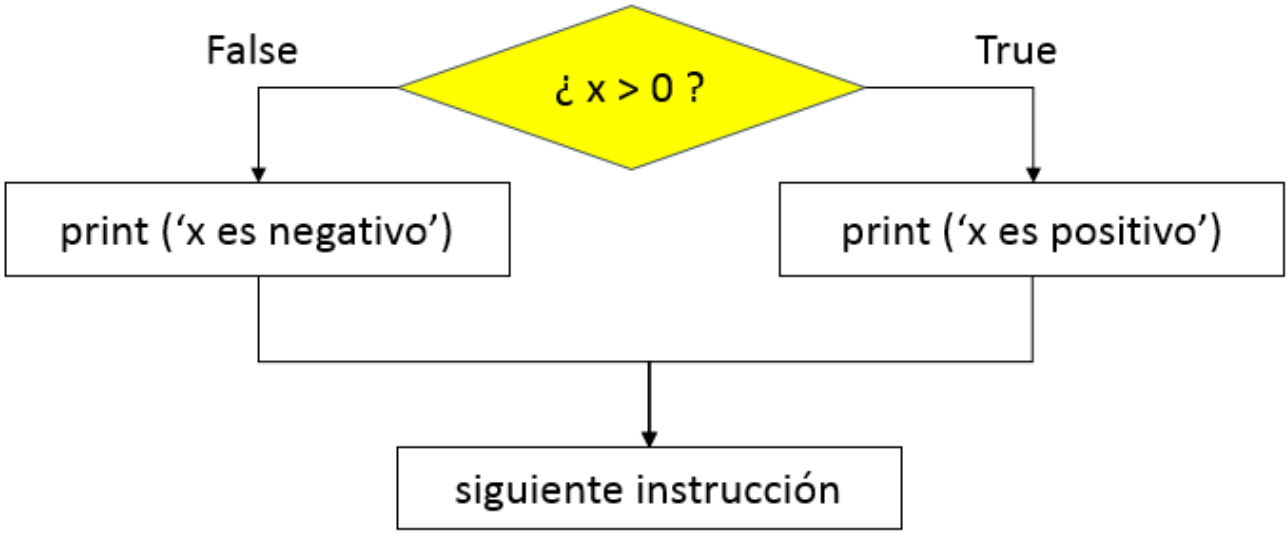
```
x = 5
if x>0:
    print ('x es positivo')
    print ('x aún es positivo')
print('Ya hemos terminado con los positivos')

for i in range(10):
    print('Vamos a evaluar ', i)
    if i % 2 == 0:
        print (i, ' es par')
    print('Hemos terminado con i: ', i)
print('Final. Todo terminado.')
```

Flujo de control selectivo

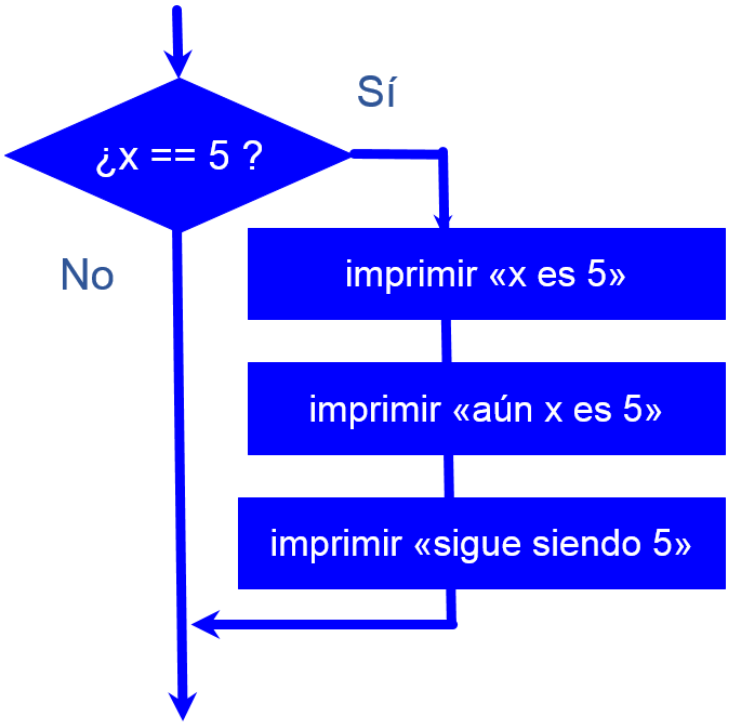


Flujo de control selectivo



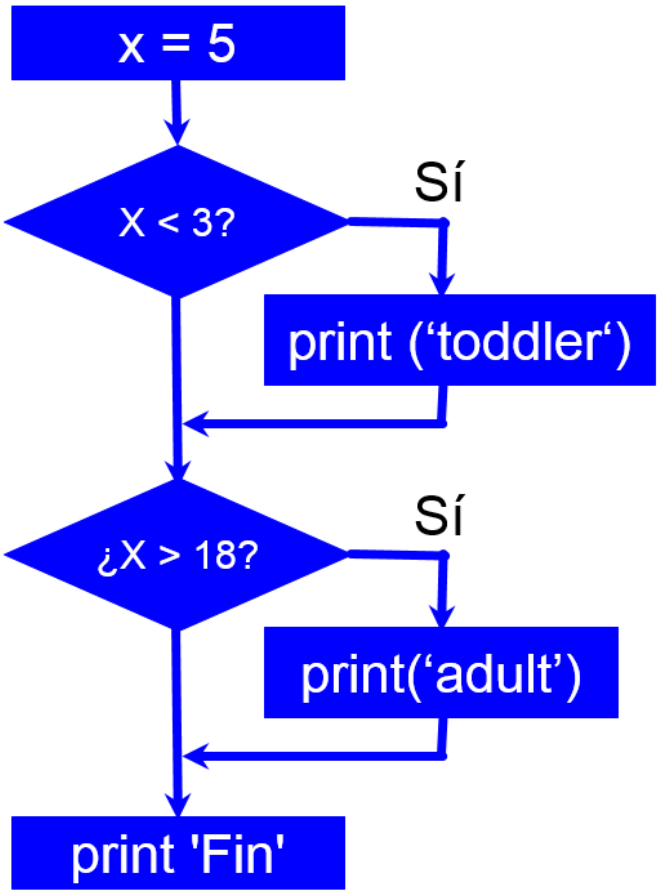

```
if x > 0:
  print('x es positivo')
else:
  print('x es negativo')
```

If simple




```
if x == 5:
 print('x es 5')
 print('aun x es 5')
 print('sigue siendo 5')
```

Ifs sucesivos




```
x = 5
if x < 3:
   print ('toddler')
if x > 18:
   print ('adult')
print('Fin')
```

Ejemplo: Par o impar, modularizado

- Como toda expresión, la condición puede incluir llamadas a funciones.

Ejemplo: Par o impar, modularizado

- Como toda expresión, la condición puede incluir llamadas a funciones.
- Ej. pueden utilizarse funciones que retornan valores lógicos en la condición.

Ejemplo: Par o impar, modularizado

- Como toda expresión, la condición puede incluir llamadas a funciones.
- Ej. pueden utilizarse funciones que retornan valores lógicos en la condición.

```
import random

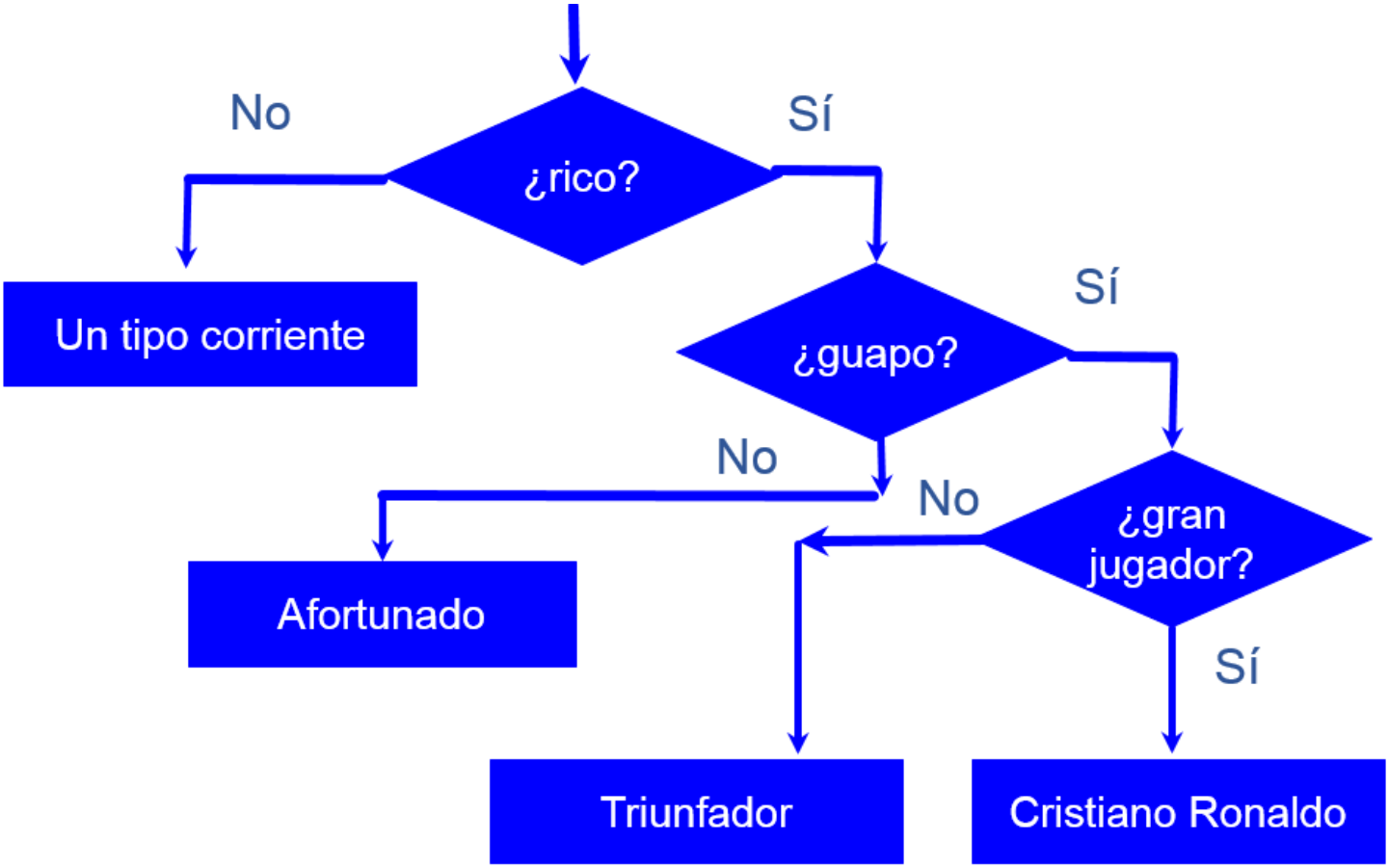
def es_par(x):
    return x % 2 == 0

n = random.randint(0,10)
if es_par(n):
    print("El numero generado es par")
else:
    print("El numero generado es impar")
```

Estructuras anidadas

- Los bloques “si” y “si no” de una selectiva (y de cualquier estructura de control) pueden a su vez contener otras estructuras de control.
- Por ejemplo: Un año es bisiesto si es divisible entre cuatro, excepto si es múltiplo de cien, en cuyo caso no es bisiesto salvo, a su vez, que sea múltiplo de cuatrocientos, en cuyo caso sí es bisiesto.

Ifs anidados



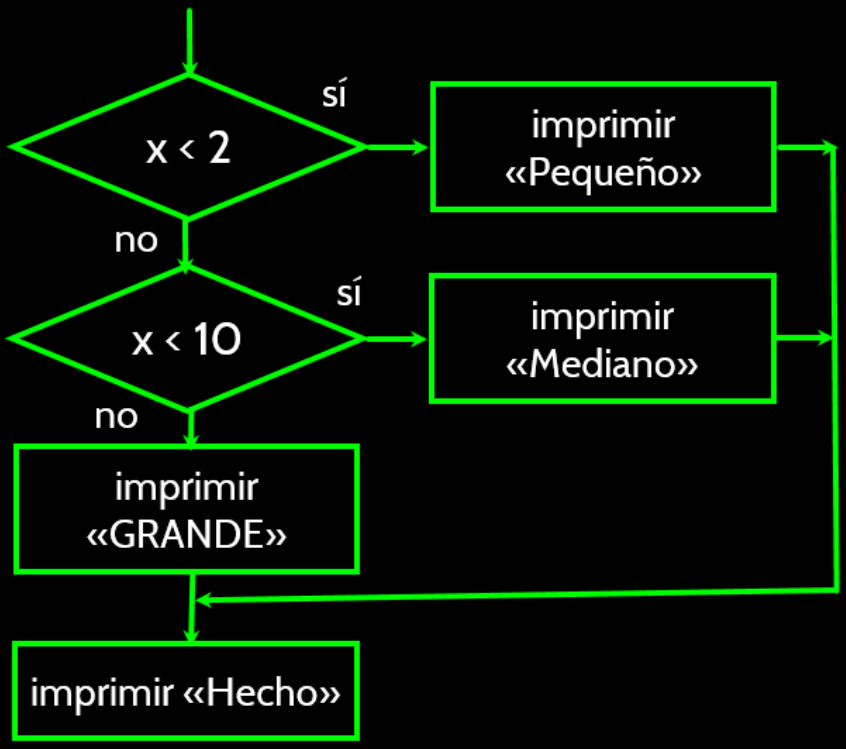
Estructuras selectivas múltiples

- Hay más de una condición
- Si no se cumple la primera se evalua otra distinta (u otras) usando la instrucción `elif`).

```
import random
dia = random.randint(0,10)
if dia == 1    : print( "Lunes" )
elif dia == 2  : print( "Martes" )
elif dia == 3  : print( "Miercoles" )
        ...
elif dia == 6  : print( " Sabado" )
elif dia == 7  : print( " Domingo" )
else           : print( " ERROR: Dia incorrecto" )
```

Condicional múltiple

```
if x < 2 :
    print 'Pequeño'
elif x < 10 :
    print 'Mediano'
else :
    print 'GRANDE'
print 'Hecho'
```



Ejercicio

¿Cuál no se imprimirá?

```
if x < 2 :
    print('menor que 2')
elif x >= 2 :
    print('2 o más')
else :
    print('Otra cosa')
```

```
if x < 2 :
    print('menor que 2')
elif x < 20 :
    print('menor que 20')
elif x < 10 :
    print('menor que 10')
else :
    print('Otra cosa')
```

Ejercicio colectivo

Escribamos un programa para, a partir de un cierto número de horas trabajadas y el precio por hora, calcular el salario bruto de un empleado. Las horas que sobrepasen 40 se considerarán extra y pagadas a 1,5 veces el precio de la hora regular.

Ejercicio colectivo

Escribamos un programa para, a partir de un cierto número de horas trabajadas y el precio por hora, calcular el salario bruto de un empleado. Las horas que sobrepasen 40 se considerarán extra y pagadas a 1,5 veces el precio de la hora regular.

```
horas = 46
ratio = 65
if (0 < horas < 40) and (ratio > 0):
   print ("Sueldo final: ", horas * ratio)
elif (horas > 40) and (ratio > 0) :
   print ("Sueldo final: ", 40*ratio + (horas-40)*1.5*ratio)
else:
   print("Error: los datos son incorrectos.")
```

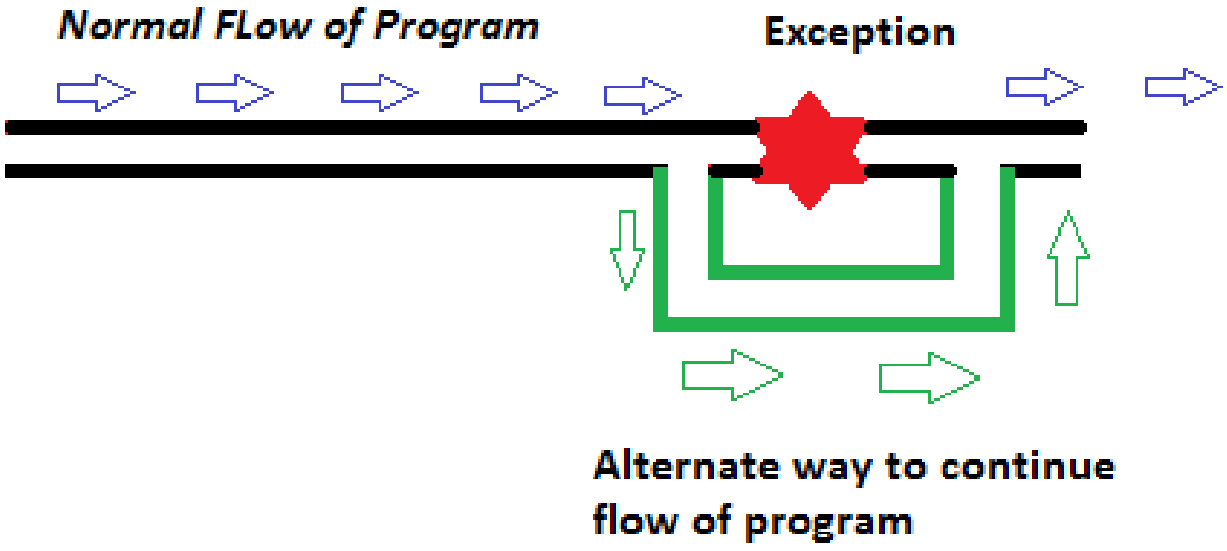
Excepciones

Definición

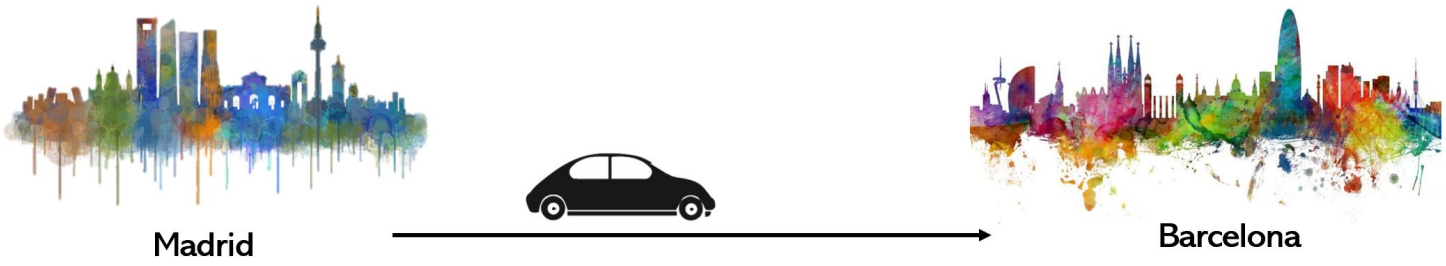
Una **excepción** es cualquier evento que provoca una disrupción en el flujo normal del programa.

- El tratamiendo de excepciones es un mecanismo que implementan algunos lenguajes para gestionar errores en tiempo de ejecución y mantener el flujo de control normal del programa.
- Las excepciones permiten separar el código "de negocio" del código de tratamiento de errores

Excepciones



Excepciones



1. Problema detectable en t. de compilación: si tráfico denso en A2 → usar R2
2. Problema detectable en t. de ejecución: reventón → arreglar y continuar
3. Problema irresoluble en t. de ejecución: hay un terremoto → no se puede continuar

Excepciones

- La detección y control de posibles errores con `if` resulta tediosa, pues modifica el aspecto del programa al llenarlo de comprobaciones.
- En Python, la sintaxis es:

```
try:
    acción potencialmente errónea
    acción potencialmente errónea
    ...
    acción potencialmente errónea
except:
    acción para tratar el error
```

Excepciones

```
anno_actual = 2020
x = input("Introduzca el anno en que nacio: ")
try:
    anno_nacimiento = int(x)
except:
    print("Algo ha fallado: no es posible calcular su edad")
else:
    edad = anno_actual - anno_nacimiento
    print("Su edad es: ", edad, " annos")
```

Resumen

- Las estructuras selectivas permiten ejecutar bloques de código diferentes dependiendo de una condición.
- La palabra reservada `if` puede emplearse en Python para construir estructuras selectivas simples, dobles (en combinación con `else` y múltiples (en combinación con `elif`.
- Las estructuras selectivas pueden anidarse.
- Las excepciones permiten separar el código de negocio del código de tratamiento de errores.

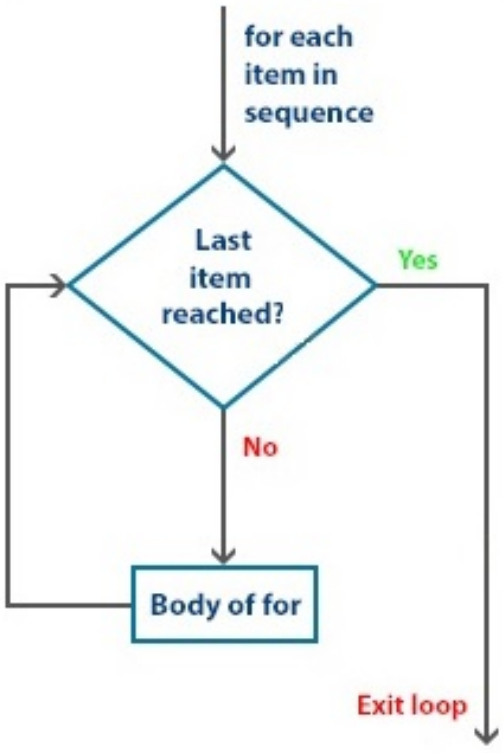
Bucles *para cada* (*for*)

- Permite ejecutar una sentencia o bloque de sentencias un número conocido de veces.
- La iteración se lleva a cabo de formas distintas:
  - Entre unos límites prefijados (Pascal)
  - Desde un valor inicial y con una condición de permanencia (C o Java)
  - Sobre una secuencia de valores conocidos (Python)

  ...pero siempre **para cada** valor de un conjunto conocido y limitado de valores.
- Una variable de control toma sucesivamente los valores que gobiernan la iteración.

El bucle `for` en Python

- La iteración recorre una secuencia u objeto iterable
- La variable de control va tomando uno a uno todos los valores y ejecutando el cuerpo del bucle.
- Tras la ejecución del cuerpo asociada al último elemento de la secuencia, el bucle termina.
- Si la secuencia no contiene elementos el cuerpo del bucle no llega a ejecutarse.



Iterables y secuencias en Python

- En Python, un objeto **iterable** es aquél capaz de ofrecer uno por uno sus elementos para su acceso
- Una **secuencia** es un tipo de iterable de uso común, con operaciones claramente definidas y formado por una serie de datos ordenados más simples.
  - Todas las secuencias son iterables pero no todos los iterables son secuencias
- Grupo heterogéneo que engloba listas, tuplas, diccionarios, conjuntos e incluso cadenas de texto.

Lista

Definición

Conjunto mutable y organizado de elementos.

- Se declara y representa encerrando los miembros entre corchetes.
- Los elementos de una lista son variables, pudiendo añadir, modificar y eliminar elementos en cualquier momento.
- Acceso a los elementos por posición mediante un índice entero: rango de 0 a longitud-1

String

Definición

Colección homogénea e inmutable cuyos elementos son caracteres de texto.

- Creación: texto entre comillas simples o dobles.
- Ejemplos:

```
texto1 = "Universidad Rey Juan Carlos"
texto2 = 'Alcala de Henares'
```

- Acceso a elementos (solo lectura) por índice entero (rango: 0 a longitud-1)

El bucle `for` en Python

- Dado que cada secuencia tiene un tipo de elementos base diferente, el modo en que un `for` opera sobre sus elementos es distinto.
- Ejemplo de iteración sobre una lista:

```
for amiga in ["Marta", "Elsa", "Ana", "Andrea"]:
    invitacion = amiga + ", te espero en mi fiesta."
    print(invitacion)
```

- Ejemplo de iteración sobre una cadena de texto:

```
for letra in "University of Cambridge":
    if es_vocal(letra): print(letra, end='')
    else: print("#", end='')
```

Función `range()` de Python

- Función que retorna un iterable de elementos enteros a menudo utilizados por una estructura `for`.
- Sintaxis: `range(ini,fin,inc)` con `ini` incluido, `fin` excluido e `inc` obligatoriamente entero.
- Ejemplos:
  - `range (5)` : [0, 1, 2, 3, 4] desde 0 hasta el número menos 1
  - `range (2,5)` : [2, 3, 4] de 2 a 5 - 1
  - `range (3, 10, 2)` : [3, 5, 7, 9] de 3 a 10 menos 1 de 2 en 2
  - `range (10, 5, -1)` : [10, 9, 8, 7, 6] de 10 a 5 + 1 de -1 en -1

Función `range()` de Python

- Se lee: *"para todo elemento de la serie, llevar a cabo una ejecución del cuerpo del bucle"*
- Ejemplo: "para cada valor de `i` entre 0 y 9 ambos incluidos, mostrar el valor de `i` por pantalla"

```
for i in range(10):
  print (i)
>>> 0 1 2 3 4 5 6 7 8 9
```

Ejemplo 1 *for*

¿Cuál es la salida de este código?

```
for i in range (1,20):
    print (i, end="")
```

Ejemplo 1 *for*

¿Cuál es la salida de este código?

```
    for i in range (1,20):
      print (i, end="")
```

>>>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Ejemplo 2 *for*

¿y de éste?

```
words = ['cat', 'dog', 'lion']
for w in words:
  print (w, end=''/'')
```

Ejemplo 2 *for*

¿y de éste?

```
words = ['cat', 'dog', 'lion']
for w in words:
  print (w, end=''/'')
```

>>>cat/dog/lion/

Ejemplo 3 *for*

¿y de este otro?

```
x = 0
for i in range (1, 20, 2):
   x += i
print(x)
```

Ejemplo 3 *for*

¿y de este otro?

```
x = 0
for i in range (1, 20, 2):
   x += i
print(x)
```

>>>100

Ejemplo 4 *for*

Generar aleatoriamente 10 números entre 0 y 100 y mostrar en pantalla su suma

Ejemplo 4 *for*

Generar aleatoriamente 10 números entre 0 y 100 y mostrar en pantalla su suma

```
import random
suma = 0
for i in range(10):
    numero_actual = random.randint(0,100)
    suma += numero_actual
print("La suma de los numeros introducidos es: ", suma)
```

Ejemplo 5 *for*

Obtener el factorial de un número

Ejemplo 5 *for*

Obtener el factorial de un número

```
def factorial(n):
    """ int -> int
        OBJ: Calcula el factorial de un entero
    """
    result = 1
    for i in range(1, n+1):
        result = result * i
    return result
```

Bucles anidados

- Un bucle anidado es un bucle (interno) dentro del cuerpo de otro bucle (externo).
- Funcionamiento:
  - La primera pasada del bucle externo activa el bucle interno, que se ejecuta hasta su finalización.
  - La segunda pasada del bucle externo activa el bucle interno nuevamente.
  - Esto se repite hasta que termina el bucle externo completamente.
- No hay límite en el número de niveles de anidamiento posibles.

Ejemplo de anidamiento *for*

Mostrar las tablas de multiplicar de los números pares

Ejemplo de anidamiento *for*

Mostrar las tablas de multiplicar de los números pares

```
for i in range(2,11,2):
    for j in range(1,11):
        print(i,"x",j,"=",i*j)
    print("---------------")
```

Ejercicio propuesto

Obtener la siguiente secuencia:

```
1 = 1
1+2 =3
1+2+3=6
1+2+3+4=10
....
```

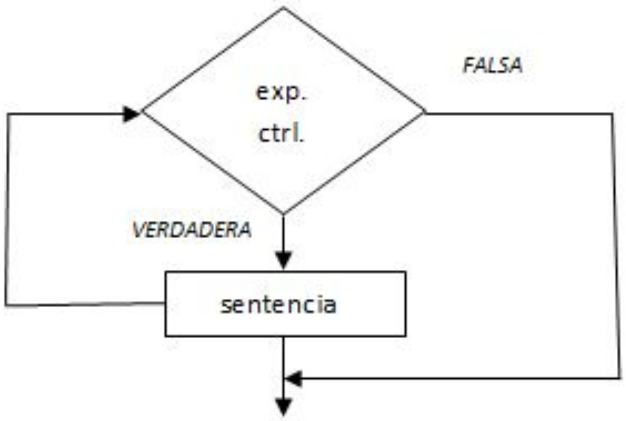
Bucles anidados

```
count = 5 # Numero de sumas
for i in range(2, count+1):
      sum = 1
      print(1, end="")
      # Calcula la suma de los enteros de 1 hasta i
      for j in range (2, i):
        sum = sum + j
        print(" + ", j, end="") # Escribir +j en
            misma linea
      print(" = ", sum)
```

Sentencia *while*

- Permite ejecutar una sentencia o bloque de sentencias 0 o más veces

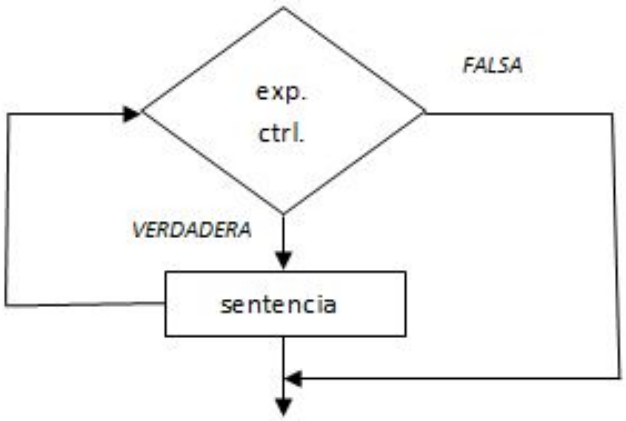
```
while condicion:
    sentencia(s)
```



Sentencia *while*

- Permite ejecutar una sentencia o bloque de sentencias 0 o más veces

```
while condicion:
    sentencia(s)
```



- Importante:
  - Diseñar cuidadosamente la expresión de control: el bucle debe tener fin.
  - Contemplar en el cuerpo del bucle la modificación de la(s) variable(s) que intervienen en la condición.

Bucles infinitos

- Ejecución continua y sin fin de un bucle
- Efecto normalmente no deseado, derivado de un error en la condición o en la modificación de las variables que gobiernan el bucle

```
n = 10
while n>0:
    print (n)
    n = n + 1
print ('Hemos terminado!')
```

Ejemplo *while*

Cruzar los valores enteros de dos variables.

Ejemplo *while*

Cruzar los valores enteros de dos variables.

```
import random
a = random.randint(1,100)
b = random.randint(1,100)
print(f'Valores iniciales: a= {a}, b={b}')
while a<b:
    a+=1
    b-=1
print(f'Valores finales: a= {a}, b={b}')
```

Ejemplo *while*

Invertir las cifras de un número entero.

Ejemplo *while*

Invertir las cifras de un número entero.

```
import random
inverso = 0
n = random.randint(1,1000)
while n % 10 == 0: n //= 10 # elimina ceros finales
print(f'Numero a invertir: {n}')
while (n > 0):
  inverso *= 10
  inverso += n % 10
  n //= 10
print("Resultado de invertir sus cifras: ", inverso)
```

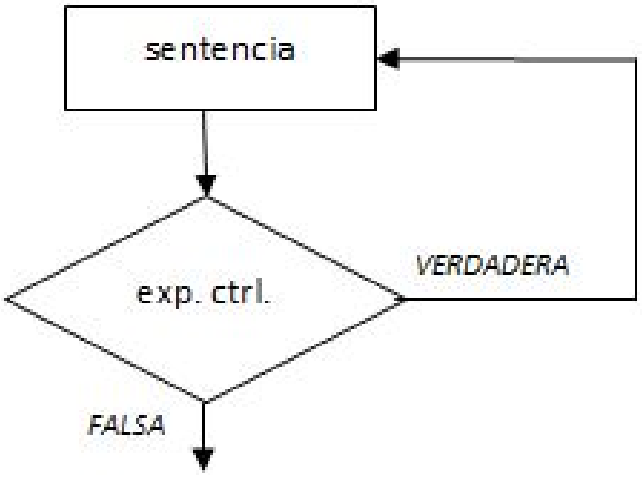
Resumen sentencia *while*

1. Se evalúa la condición, obteniendo `True` o `False`.
2. Si la condición es *False*, se sale de la sentencia *while* y el flujo de control continúa en la siguiente sentencia.
3. Si la condición es *True* ejecuta el cuerpo del bucle y vuelve al paso 1.

Este tipo de flujo se denomina *bucle* porque el tercer paso enlaza directamente con el primero estableciendo un mecanismo de repetición.

El bucle *repetir*

- Representa la repetición de una sentencia o bloque de sentencias 1 o más veces



- Python no ofrece una construcción específica para este concepto.

Implementación del concepto “repetir” en Python

Calcular la media de todos los números entre 0 y 100 aleatoriamente generados hasta que salga un cero

Implementación del concepto "repetir" en Python

Calcular la media de todos los números entre 0 y 100 aleatoriamente generados hasta que salga un cero

```
import random
CENTINELA = 0
cuantos = 0
suma = 0
valor = random.randint(0,100) # iteracion pre-bucle
while (valor != CENTINELA):
  suma += valor
  cuantos += 1
  valor = random.randint(0,100)
if (cuantos>0):
  print ("La media es: ", suma/cuantos)
```

Ejemplo repetir *while*

Encontrar un elemento en una lista eligiendo al azar...

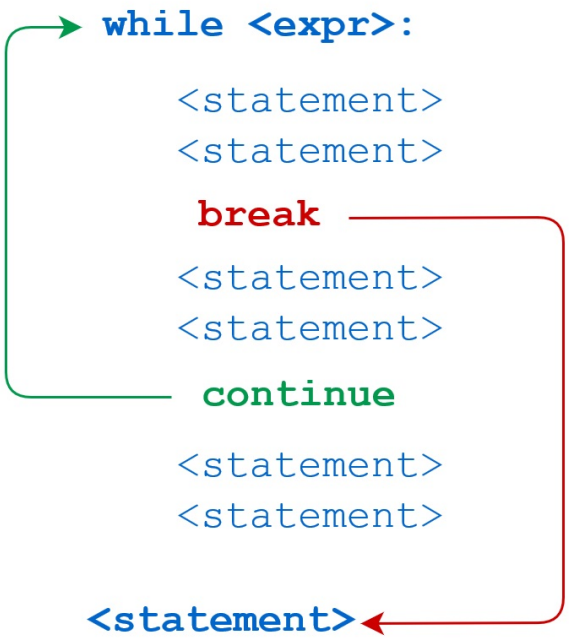
Ejemplo repetir *while*

Encontrar un elemento en una lista eligiendo al azar...

```
import random
my_list = ["One", "Two", "Three", "Four"]
intentos = 1
my_choice = random.choice(my_list)
while (my_choice != "Three"):
    intentos += 1
    my_choice = random.choice(my_list)
print(f'Ha costado {intentos} intentos dar con el 3')
```

Salida forzada de un bucle

- Existen dos instrucciones controvertidas (`break` y `continue`) que permiten salir de una iteración en curso subvirtiendo el normal comportamiento del bucle



Resumen

- Las estructuras selectivas permiten ejecutar bloques de código diferentes dependiendo de una condición.
- Las estructuras iterativas permiten ejecutar bloques de código repetidas veces.
- Existen varios tipo de bucles: 0-n, 1-n (repetir), for.
- Las estructuras de control pueden anidarse
- Aunque poco ortodoxas, existen formas de salir de un bucle a mitad de una iteración (`break` y `continue`)

Referencias y reconocimientos

- Algunos contenidos de esta presentación han sido adaptados de los materiales del curso de "Programming for Everybody (Python)", creado por Charles Severance y disponible en `https://www.coursera.org/course/pythonlearn`.
- También hemos tomado explicaciones del libro "Introducción a la programación con Python 3" de Marzal y su equipo de la Univ. Jaume I: `http://issuu.com/universitatjaumei/docs/sapientia_93`
- Algunos contenidos de esta presentación han sido adaptados de los materiales del curso "Programming for Everybody (Python)", creado por Charles Severance.
- También se han incluído explicaciones y ejemplos del libro "Introducción a la programación con Python 3" de A. Marzal y su equipo de la Univ. Jaume I.
- Como en otros temas, esta presentación es una adaptación de la original elaborada por el autor junto con Elena García y Miguel A. Sicilia para la asignatura "Fundamentos de la programación" de la Universidad de Alcalá.

Tema II: Estructuras de control

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Octubre de 2025



Tema III: Subprogramación

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Octubre de 2025



Licencia

© 2025.

Algunos derechos reservados.

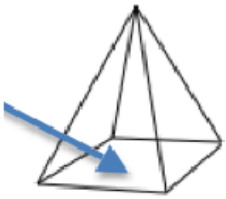
- Isaac Lozano Osorio (isaac.lozano@urjc.es)
- Salvador Sánchez Alonso (salvador.sanchez@urjc.es)

Este documento se distribuye bajo la licencia según la licencia Compartir Igual 4.0 (CC BY-SA 4.0)

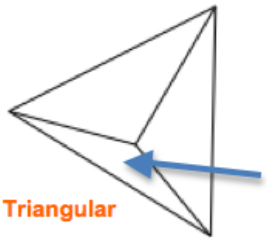


Problema: cálculo del área de una pirámide

- $area = area_{base} + area_{lateral}$
- $area_{base} = ladobase^2$
- $area_{lateral} = \frac{perimetro \times apotema}{2}$
- $apotema = \sqrt{altura^2 + (\frac{ladobase}{2})^2}$
- En el caso de una pirámide de base triangular, $area_{base} = \frac{ladobase \times altura_{base}}{2}$
- Y para obtener la altura de la base: $altura_{base} = \sqrt{ladobase^2 - (\frac{ladobase}{2})^2}$



Square



Triangular

Calculando el área de pirámides

```
# Calculo del areas de varios tipos de piramide

# Piramide regular cuadrangular
lado_base = float(input("Introduzca la base de la piramide: "))
h_piramide = float(input("Introduzca ahora la altura: "))
apotema = pow((pow(h_piramide, 2) + pow((lado_base/2),2)),0.5)
perimetro = lado_base * 4
area_lateral = (perimetro * apotema) / 2
area_base = pow(lado_base, 2)
area_total = area_lateral + area_base
print("Area de la piramide cuadrangular: ", area_total);

# Piramide regular triangular
lado_base = float(input("Introduzca la base de la piramide: "))
h_piramide = float(input("Introduzca ahora la altura: "))
apotema = pow((pow(h_piramide, 2) + pow((lado_base/2),2)),0.5)
perimetro = lado_base * 3
h_base = pow(pow(lado_base,2) - pow((lado_base / 2), 2), 0.5)
area_lateral = (perimetro * apotema) / 2
area_base = (lado_base * h_base) / 2
area_total = area_lateral + area_base
print("Area de la piramide triangular: ", area_total);
```

Algunos problemas del programa anterior

- Poco legible

Algunos problemas del programa anterior

- Poco legible
- Repetición del código: Cálculo del perímetro, área lateral, área total, muestra de resultados, etc.

Algunos problemas del programa anterior

- Poco legible
- Repetición del código: Cálculo del perímetro, área lateral, área total, muestra de resultados, etc.
- Dificultad de depuración:
  - ¿Qué ocurre –por ejemplo– si hemos implementado mal la fórmula para calcular el área lateral?

Algunos problemas del programa anterior

- Poco legible
- Repetición del código: Cálculo del perímetro, área lateral, área total, muestra de resultados, etc.
- Dificultad de depuración:
  - ¿Qué ocurre –por ejemplo– si hemos implementado mal la fórmula para calcular el área lateral?
  - Se nos puede olvidar modificar la fórmula en alguna línea

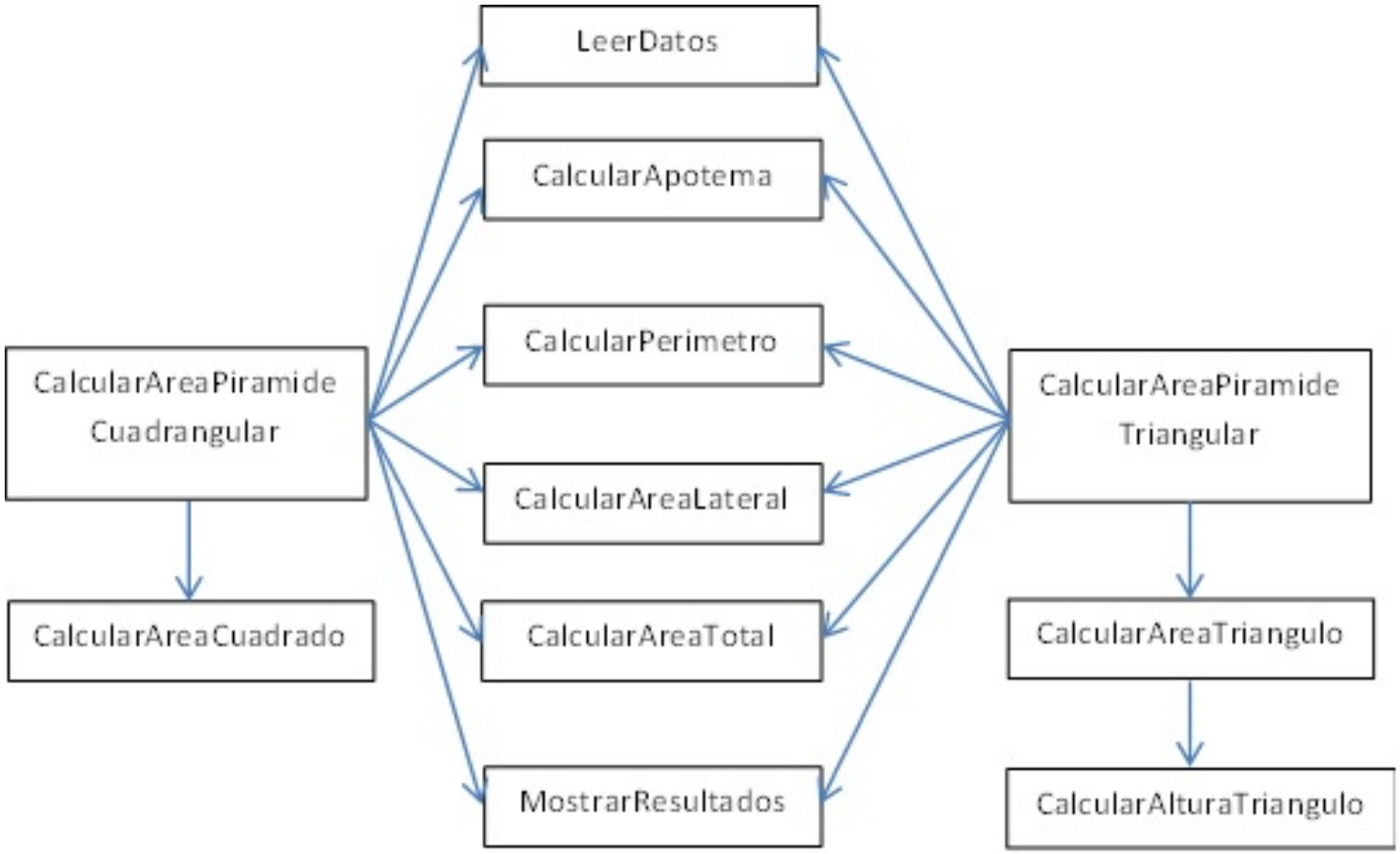
Algunos problemas del programa anterior

- Poco legible
- Repetición del código: Cálculo del perímetro, área lateral, área total, muestra de resultados, etc.
- Dificultad de depuración:
  - ¿Qué ocurre –por ejemplo– si hemos implementado mal la fórmula para calcular el área lateral?
  - Se nos puede olvidar modificar la fórmula en alguna línea
- Dificultad de mantenimiento

Algunos problemas del programa anterior

- Poco legible
- Repetición del código: Cálculo del perímetro, área lateral, área total, muestra de resultados, etc.
- Dificultad de depuración:
  - ¿Qué ocurre –por ejemplo– si hemos implementado mal la fórmula para calcular el área lateral?
  - Se nos puede olvidar modificar la fórmula en alguna línea
- Dificultad de mantenimiento
- Mayor coste de desarrollo

Pasos para calcular el área de una pirámide regular



Subprograma

Definición

Un subprograma es una porción de código relativamente independiente que puede ser **llamado**, **enviándole (o no) datos** para que realice una determinada **tarea** y/o proporcione una serie de **resultados**.

- Cuando el subprograma retorna valor(es) al código que lo invocó hablamos de una **función**
- Si por el contrario no retorna valores, es un **procedimiento**

Ventajas de usar subprogramas

- **Reutilización** de código.
  - El mismo cálculo del perímetro, área lateral, etc. se reutiliza para calcular el área de la pirámide triangular y cuadrangular.

Ventajas de usar subprogramas

- **Reutilización** de código.
  - El mismo cálculo del perímetro, área lateral, etc. se reutiliza para calcular el área de la pirámide triangular y cuadrangular.
  - Un mismo subprograma puede ser utilizado por diferentes programas o en diferentes partes del programa.

Ventajas de usar subprogramas

- **Reutilización** de código.
  - El mismo cálculo del perímetro, área lateral, etc. se reutiliza para calcular el área de la pirámide triangular y cuadrangular.
  - Un mismo subprograma puede ser utilizado por diferentes programas o en diferentes partes del programa.
- **Legibilidad**.

Ventajas de usar subprogramas

- **Reutilización** de código.
  - El mismo cálculo del perímetro, área lateral, etc. se reutiliza para calcular el área de la pirámide triangular y cuadrangular.
  - Un mismo subprograma puede ser utilizado por diferentes programas o en diferentes partes del programa.
- **Legibilidad**.
- Mejora de los procesos de **depuración y pruebas**.
  - Ambos se llevan a cabo por subprogramas, no globalmente.

Ventajas de usar subprogramas

- **Reutilización** de código.
  - El mismo cálculo del perímetro, área lateral, etc. se reutiliza para calcular el área de la pirámide triangular y cuadrangular.
  - Un mismo subprograma puede ser utilizado por diferentes programas o en diferentes partes del programa.
- **Legibilidad**.
- Mejora de los procesos de **depuración y pruebas**.
  - Ambos se llevan a cabo por subprogramas, no globalmente.
- Mayor facilidad de **mantenimiento**.
  - Las modificaciones se realizan una sola vez: en el subprograma.

Subprogramas: Características

- Fuerte **cohesión**:
  - Debe existir una clara relación funcional entre las sentencias o grupos de sentencias de un mismo subprograma.
  - Un subprograma fuertemente cohesionado ejecutará una única tarea sencilla.

Subprogramas: Características

- Fuerte **cohesión**:
  - Debe existir una clara relación funcional entre las sentencias o grupos de sentencias de un mismo subprograma.
  - Un subprograma fuertemente cohesionado ejecutará una única tarea sencilla.
- **Idea**: para ver si es cohesivo un subprograma, analizar si puede describirse con una oración simple, con un solo verbo activo. Si hay más de un verbo activo en su descripción, sopesar su descomposición en más de un subprograma.

Subprogramas: Características

- Débil **acoplamiento**:

Subprogramas: Características

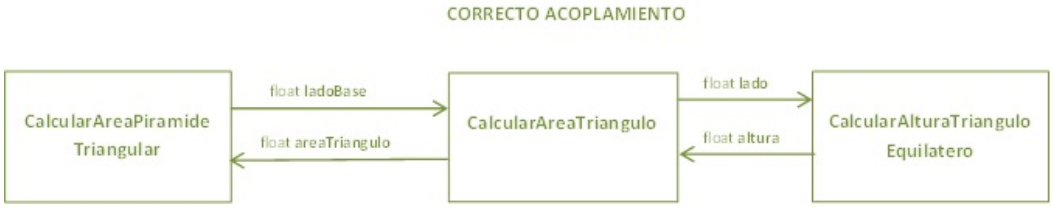
- Débil **acoplamiento**:
  - Debe existir poca interdependencia entre los subprogramas de un programa.

Subprogramas: Características

- Débil **acoplamiento**:
  - Debe existir poca interdependencia entre los subprogramas de un programa.
  - Un subprograma se comunicará con otros únicamente a través de los parámetros de entrada y de los valores de salida.

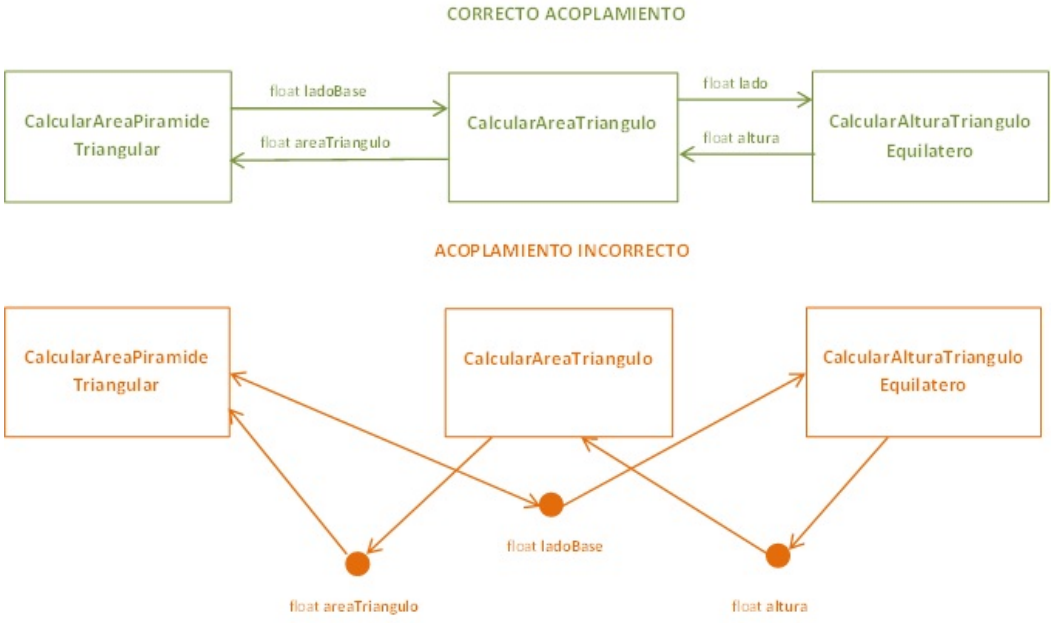
Subprogramas: Características

- Débil **acoplamiento**:
  - Debe existir poca interdependencia entre los subprogramas de un programa.
  - Un subprograma se comunicará con otros únicamente a través de los parámetros de entrada y de los valores de salida.
  - Evitar el uso de variables globales



Subprogramas: Características

- Débil **acoplamiento**:
  - Debe existir poca interdependencia entre los subprogramas de un programa.
  - Un subprograma se comunicará con otros únicamente a través de los parámetros de entrada y de los valores de salida.
  - Evitar el uso de variables globales



Implementación de subprogramas

Definición

La definición de un **subprograma** consiste en la escritura del código necesario para que éste realice las tareas para la que ha sido previsto.

Implementación de subprogramas

Definición

La definición de un **subprograma** consiste en la escritura del código necesario para que éste realice las tareas para la que ha sido previsto.

```
# (1) Encabezamiento o cabecera
def nombre_subprograma(lista_argumentos):
# (2) Documentacion  (docstrings)
# (3) Instrucciones del subprograma
 return expresion # (4) Retorno [opcional]
```

Documentación de los subprogramas

- La documentación es transparente para la máquina: sin ella el subprograma funciona igual, pero es muy importante para su reutilización
- Debe ser clara y concisa.
- Orientada a los programadores: puede incluir términos técnicos.
- En Python se escribe con docstrings: entre tres símbolos de dobles comillas (“ “ “ documentación aquí ” ” ”) detrás de la instrucción `def` y antes el cuerpo del subprograma.
  - No hay un modelo único: PEP484, alternativas PEP484 compatibles con Python 2, otras...

Docstrings estilo PEP 484

- Las anotaciones de tipo propuestas en PEP 484 permiten incluir información sobre los tipos de los elementos utilizando una sintaxis familiar a programadores de otros lenguajes como C o Pascal.

```
def function_pep484(param1: int, param2: str) -> bool:
    """Funcion con anotaciones de tipo estilo PEP 484.

    Args:
        param1: Primer parametro.
        param2: Segundo parametro.

    Returns:
        Valor de retorno. True cuando tiene exito, False
            en caso contrario.

    """
```

Ejemplo estilo PEP484

```
def bigger_than(a:int, b:int) -> bool:
    """
    Args:
        a: The first parameter.
        b: The second parameter.

    Returns:
        True if a > b, False otherwise.
    """
    return a > b
```

```
In [17]: help(bigger_than)
Help on function bigger_than in module __main__:

bigger_than(a:int, b:int) -> bool
    Args:
        a: The first parameter.
        b: The second parameter.

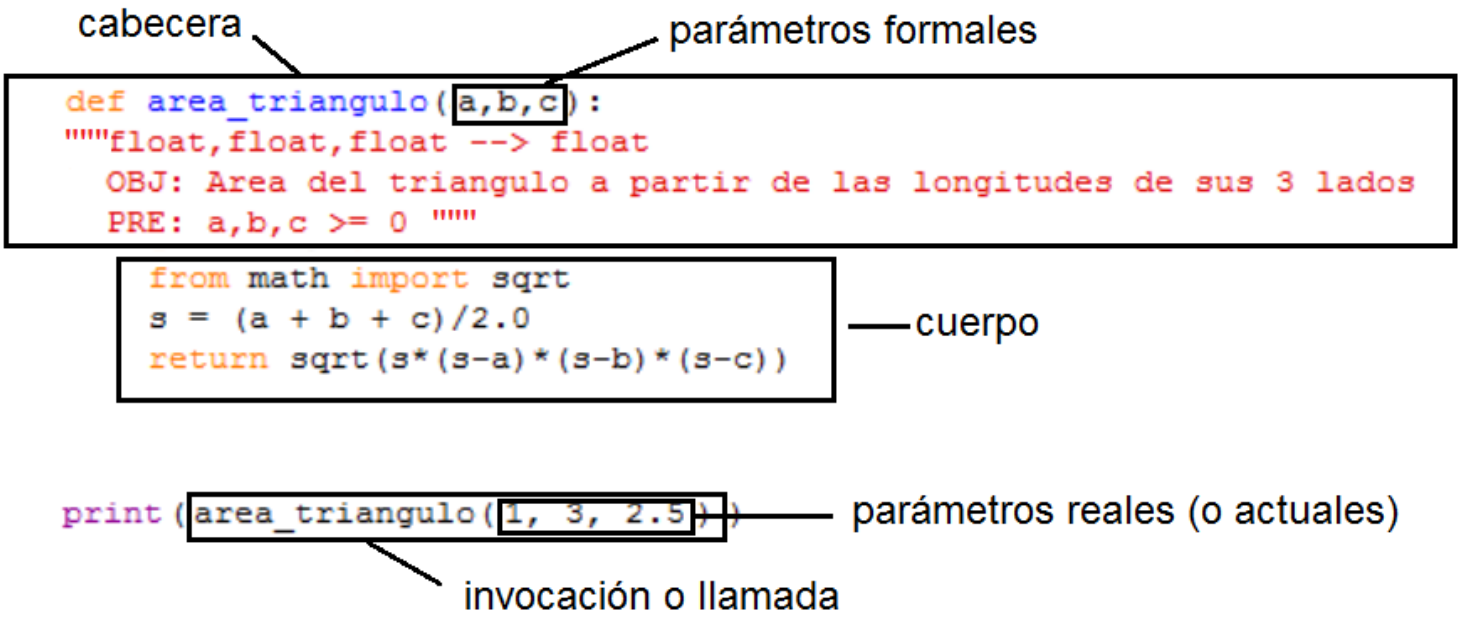
    Returns:
        True if a > b, False otherwise.
```

Enfoque en la asignatura: cabecera como contrato

Proponemos una alternativa sencilla y compacta que sigue el enfoque de cabecera como contrato: "si Vd. me da argumentos que satisfagan lo especificado en PRE, le garantizo que haré lo que dice OBJ".

- Línea 1: contiene los tipos de los argumentos y de la salida
- Línea 2: tras "OBJ:" (objetivo) una frase sintetiza el objetivo del subprograma, el papel que desempeña cada argumento y lo que devuelve el subprograma ('None' si es un procedimiento).
- Línea 3: Solo si existen condiciones previas para que el subprograma haga su trabajo, deben indicarse tras la etiqueta "PRE:" (precondición).

Estructura de un subprograma



Ejemplos de subprogramas

```
def euros_to_pesetas(amount):
    """ float -> float
        OBJ: Transforms euros to ES pesetas
    """
    pesetas = amount * 166.386
    return pesetas

def euros_to_dollars(amount, exchange_rate):
    """ float, float -> float
        OBJ: Euros to US dollars using a given exchange rate
    """
    return amount * exchange_rate

print(euros_to_pesetas(euros_to_dollars(40,2)))
```

Ejemplos de subprogramas

```
def bilingual_hi(name):
  """ str -> None
      OBJ: Says hi in english and spanish
  """
  print("Hola,", name)
  print("Hello,", name)

bilingual_hi('joe')
```

Implementación de los subprogramas

- Un subprograma se puede llamar desde otra parte del programa posterior a su definición.

```
# Llamada o invocacion:
nombre(lista_arg_reales)
```

Implementación de los subprogramas

- Un subprograma se puede llamar desde otra parte del programa posterior a su definición.

```
# Llamada o invocacion:
nombre(lista_arg_reales)
```

- Dependiendo del tipo de retorno, las llamadas se pueden realizar:
  - Dentro de una expresión (funciones): El tipo de retorno debe coincidir con los tipos válidos de la expresión.
  - En una sentencia que únicamente contenga la llamada, si no retorna ningún valor (procedimientos).

Ejemplo: Función que calcula el cuadrado de un número

```
# Declaracion de la funcion:
def cuadrado (x):
""" float -> float
    OBJ: Calcula el cuadrado de un numero x    """
    return x * x

# Usos de la funcion:
...
y = cuadrado (5)
z = 45 + cuadrado (a+b) / 2
print("El cuadrado de 13 es: ", cuadrado(13))
```

Subprogramas sin implementar (aún)

- Python permite postergar la implementación de un subprograma permitiendo utilizarlo en llamadas, etc.

Subprogramas sin implementar (aún)

- Python permite postergar la implementación de un subprograma permitiendo utilizarlo en llamadas, etc.
- Tal vez su implementación aún no se conoce o no se desea especificar

Subprogramas sin implementar (aún)

- Python permite postergar la implementación de un subprograma permitiendo utilizarlo en llamadas, etc.
- Tal vez su implementación aún no se conoce o no se desea especificar
- Sentencia `pass`:

```
def cuadrado (x): pass
""" Aun no lo tengo claro """

# Codigo de prueba:
print(cuadrado(5))

>>> None
```

Comunicación

- Los subprogramas se comunican con otras partes del código mediante el paso de argumentos y el retorno de valores

Comunicación

- Los subprogramas se comunican con otras partes del código mediante el paso de argumentos y el retorno de valores
- Un parámetro es un valor que el subprograma espera recibir cuando sea invocado, a fin de ejecutar acciones según dicho valor.

Comunicación

- Los subprogramas se comunican con otras partes del código mediante el paso de argumentos y el retorno de valores
- Un parámetro es un valor que el subprograma espera recibir cuando sea invocado, a fin de ejecutar acciones según dicho valor.
- Un subprograma puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

Comunicación

- Los subprogramas se comunican con otras partes del código mediante el paso de argumentos y el retorno de valores
- Un parámetro es un valor que el subprograma espera recibir cuando sea invocado, a fin de ejecutar acciones según dicho valor.
- Un subprograma puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.
- Al concluir su cometido el subprograma puede (o no) devolver uno o más valores al módulo que los invocó.

Tipo de argumentos

- Los argumentos o parámetros pueden ser:
  - Formales. Los que se definen en la cabecera del subprograma.
  - Reales o actuales. Los que se especifican en la llamada al subprograma.

Tipo de argumentos

- Los argumentos o parámetros pueden ser:
  - Formales. Los que se definen en la cabecera del subprograma.
  - Reales o actuales. Los que se especifican en la llamada al subprograma.
- Hay correspondencia biunívoca entre los parámetros reales y formales: Deben coincidir en número, semántica y tipo esperado.

Tipo de argumentos

- Los argumentos o parámetros pueden ser:
  - Formales. Los que se definen en la cabecera del subprograma.
  - Reales o actuales. Los que se especifican en la llamada al subprograma.
- Hay correspondencia biunívoca entre los parámetros reales y formales: Deben coincidir en número, semántica y tipo esperado.
- Pueden no existir (en Python se conservan los paréntesis)

Ejemplo de subprogramas sin argumentos

```
# Declaracion del subprograma
def muestra_fecha_hora():
  """ None -> None
      OBJ : Muestra en pantalla la fecha y hora actuales
  """
  import time
  print ('Fecha : ', time.strftime ("%x"))
  print ('Hora  : ', time.strftime ("%H:%M:%S"))

# Uso del subprograma:
muestra_fecha_hora()
```

Parámetros no posicionales

En Python es posible invocar subprogramas especificando el nombre y valor de los parámetros aunque no estén posicionalmente colocados (parámetros de tipo keyword):

```
def ejemplo(a,b):
    """ int, int -> int
        OBJ : Funcion tonta para ejemplos
    """
    return a-b

#probador
print(ejemplo(8,9))
>> -1
print(ejemplo(b=90, a=9))
>> -81
```

Comunicación entre módulos: retorno

- Una **función** es un subprograma que realiza un cómputo y retorna un valor:

```
def hipotenusa(a,b):
 """ float,float -> float
     OBJ: hipotenusa triangulo rectangulo, catetos: a,b
     PRE: a,b > 0
 """
 import sqrt
 return math.sqrt(a**2 + b**2)
```

Comunicación entre módulos: retorno

- Los **procedimientos** imprimen, muestran, ordenan, etc. pero no se espera de ellos un valor como retorno:

```
# Declaracion:
def saludar (nombre):
    """
       str -> None
       OBJ : Saludo personalizado a una persona
    """
    print('Hola, ', nombre)

# Uso del procedimiento:
saludar('Maria')
```

Separación de la interfaz

- En general deben separarse las acciones de interfaz de los cómputos (bajo acoplamiento):

```
def mal_ejemplo(a,b):
""" None -> None
    OBJ: Separacion incorrecta de computos e interfaz
"""
    media = a/b
    print('La media es: ', media)

mal_ejemplo(2,4)
```

Elegir bien el nombre de los subprogramas

- El nombre debe describir claramente lo que hace (si es un procedimiento) o lo que devuelve (si es una función)

Elegir bien el nombre de los subprogramas

- El nombre debe describir claramente lo que hace (si es un procedimiento) o lo que devuelve (si es una función)
- El nombre de un procedimiento debe ser un verbo de acción: `imprimir`, `mostrar`, `ordenar`, `centrar`...

Elegir bien el nombre de los subprogramas

- El nombre debe describir claramente lo que hace (si es un procedimiento) o lo que devuelve (si es una función)
- El nombre de un procedimiento debe ser un verbo de acción: `imprimir`, `mostrar`, `ordenar`, `centrar`...
- En el nombre de las funciones booleanas encajan bien verbos de estado como ser, estar o tener. Ej: `es_bisiesto`, `esta_llena`, `tiene_huecos`

Elegir bien el nombre de los subprogramas

- El nombre debe describir claramente lo que hace (si es un procedimiento) o lo que devuelve (si es una función)
- El nombre de un procedimiento debe ser un verbo de acción: `imprimir`, `mostrar`, `ordenar`, `centrar`...
- En el nombre de las funciones booleanas encajan bien verbos de estado como ser, estar o tener. Ej: `es_bisiesto`, `esta_llena`, `tiene_huecos`
- En aquellas funciones que devuelven la magnitud de una propiedad o una entidad, el nombre será el sustantivo correspondiente a lo que devuelve: `hipotenusa`, `cuadrado`, `raiz_cubica`...

El ámbito de las variables

- Variables **locales**: Se emplean dentro de un subprograma.
  - Sólo son accesibles en el subprograma en que se utilizan.

El ámbito de las variables

- Variables **locales**: Se emplean dentro de un subprograma.
  - Sólo son accesibles en el subprograma en que se utilizan.
- Variables **globales**: Se definen fuera de cualquier subprograma.
  - Accesibles desde cualquier lugar salvo que su nombre sea ocultado por otra variable dentro de un subprograma.

El ámbito de las variables

- Variables **locales**: Se emplean dentro de un subprograma.
  - Sólo son accesibles en el subprograma en que se utilizan.
- Variables **globales**: Se definen fuera de cualquier subprograma.
  - Accesibles desde cualquier lugar salvo que su nombre sea ocultado por otra variable dentro de un subprograma.
- El **acceso global a una variable** consiste en acceder a una variable perteneciente al subprograma llamante.
  - Especialmente desaconsejado si es en escritura.
  - ¡Pueden generar efectos laterales indeseados!

Ejemplo: uso no recomendado de variables globales

```
def funcion_mal_ejemplo():
    """ None -> None
        OBJ: Ejemplo de uso no recomendado de variables
            globales
    """
    media = (a + b)/2
    print('Media: ', media)

a=20
b=10
funcion_mal_ejemplo()
```

Ejemplo: uso recomendado de variables globales

```
def media_aritmetica(x,y):
    """ int, int -> float
        OBJ: Calculo de la media aritmetica de dos enteros
    """
    media = (x + y)/2
    return media

# probador
a = int(input('Introduce el valor para a (entero): '))
b = int(input('Introduce el valor para b (entero): '))
print(f'La media aritmetica de {a} y {b} es: {
    media_aritmetica(a,b)} ')
```

El paso de argumentos en los lenguajes de programación

- En general hay 2 enfoques: por valor (entrada) y por referencia (entrada-salida).

El paso de argumentos en los lenguajes de programación

- En general hay 2 enfoques: por valor (entrada) y por referencia (entrada-salida).
- El paso **por valor** implica copia: el subprograma hace una copia local del argumento real y lo utiliza. Si modifica su copia local, el original no se altera.

El paso de argumentos en los lenguajes de programación

- En general hay 2 enfoques: por valor (entrada) y por referencia (entrada-salida).
- El paso **por valor** implica copia: el subprograma hace una copia local del argumento real y lo utiliza. Si modifica su copia local, el original no se altera.
- En el paso **por referencia** el subprograma usa el valor original mediante un nombre (referencia) local. Si se modifica el objeto a través de su referencia, el original se modifica.

Paso de argumentos en Python

- En Python, los subprogramas reciben parámetros que pueden ser tratados como de entrada o de entrada-salida.

Paso de argumentos en Python

- En Python, los subprogramas reciben parámetros que pueden ser tratados como de entrada o de entrada-salida.
- Si queremos que sean de **entrada**, únicamente aparecerán en la lista de argumentos.

Paso de argumentos en Python

- En Python, los subprogramas reciben parámetros que pueden ser tratados como de entrada o de entrada-salida.
- Si queremos que sean de **entrada**, únicamente aparecerán en la lista de argumentos.
- Si son de **entrada-salida**, aparecerán también en el retorno del subprograma y recogeremos su resultado en el código llamante.

Paso de argumentos en Python

- En Python, los subprogramas reciben parámetros que pueden ser tratados como de entrada o de entrada-salida.
- Si queremos que sean de **entrada**, únicamente aparecerán en la lista de argumentos.
- Si son de **entrada-salida**, aparecerán también en el retorno del subprograma y recogeremos su resultado en el código llamante.

```
def incrementar(x,n):
    """
       int, int -> int
       OBJ: Incrementa el entero x en n unidades
    """
    return x+n;

#Probador
a = 10
print (a)
a = incrementar(a,2)
print (a)
```

Paso de argumentos en Python: otro ejemplo

```
def funcion_pythoniana_correcta(a,c):  # TODO LO QUE ENTRA
    """ int,int -> int,int
    """
    b = 2*a+c              # usa todo lo que entra
    c = 2+c                # modifico todo lo que sale
    return b,c             # TODO LO QUE SALE

#Probador
a=2
b=3
c=4
print('\t E \tS\t E/S')
print('antes   a=',a,'\t b=',b,'\t c=',c)
b,c = funcion_pythoniana_correcta(a,c)
print('despues a=',a,'\t b=',b,'\t c=',c)
```

Bibliotecas

- Permiten agrupar funciones y otros elementos para que las puedan utilizar otros programas.

Bibliotecas

- Permiten agrupar funciones y otros elementos para que las puedan utilizar otros programas.
- Agrupación por funcionalidad relacionada: operaciones con cadenas, operaciones matemáticas, funciones gráficas, etc.

Bibliotecas

- Permiten agrupar funciones y otros elementos para que las puedan utilizar otros programas.
- Agrupación por funcionalidad relacionada: operaciones con cadenas, operaciones matemáticas, funciones gráficas, etc.
- En Python es tan sencillo como almacenarlas juntas en un archivo .py

Bibliotecas

- Permiten agrupar funciones y otros elementos para que las puedan utilizar otros programas.
- Agrupación por funcionalidad relacionada: operaciones con cadenas, operaciones matemáticas, funciones gráficas, etc.
- En Python es tan sencillo como almacenarlas juntas en un archivo .py
- Se traducen por separado

Bibliotecas

- Permiten agrupar funciones y otros elementos para que las puedan utilizar otros programas.
- Agrupación por funcionalidad relacionada: operaciones con cadenas, operaciones matemáticas, funciones gráficas, etc.
- En Python es tan sencillo como almacenarlas juntas en un archivo .py
- Se traducen por separado
- Las bibliotecas se importan en los programas que las van a usar:

```
import math #pone disponible los elementos de math
print(math.pi)
# otro enfoque (menos recomendable)...
from math import pi #hace disponible solo un elemento
print(pi)
```

Resumen

- Los diseños modulares tiene muchas e importantes ventajas frente a los no modulares.
- Es deseable que los subprogramas tengan poco acoplamiento y mucha cohesión.
- Resulta imprescindible documentar adecuadamente los subprogramas, para lo cual existen varias alternativas.
- En función de dónde se definan las variables y de cómo se puede acceder a ellas son globales o locales.
- Los subprogramas se comunican con el resto del programa a través de los argumentos y los valores de retorno (concepto de contrato).
- Las bibliotecas permiten agrupar subprogramas, constantes y otros elementos reutilizables.

Referencias y reconocimientos

- Algunos de los contenidos de esta presentación han sido adaptados de los materiales del curso "Programming for Everybody (Python)", creado por Charles Severance y disponible en `https://www.coursera.org/learn/python`.
- Otros contenidos se basan en los apuntes elaborados por el autor junto con el profesor Miguel A. Sicilia para la asignatura Fundamentos de la Programación que se imparte en las titulaciones de grado de la Universidad de Alcalá.

Tema III: Subprogramación

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Octubre de 2025



Tema IV: Estructuras de datos

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Noviembre de 2025



Licencia

© 2025.

Algunos derechos reservados.

- Isaac Lozano Osorio (isaac.lozano@urjc.es)
- Salvador Sánchez Alonso (salvador.sanchez@urjc.es)

Este documento se distribuye bajo la licencia según la licencia Compartir Igual 4.0 (CC BY-SA 4.0)



Colecciones de datos

Definición

Tipos de datos que representan grupos de elementos.

- Abstracciones que facilitan el manejo y acceso a los datos.
- Cada tipo de colección posee características específicas en lo relativo a la homogeneidad o heterogeneidad de los elementos, orden, acceso por posición, tamaño fijo o variable, y otros.
- Las colecciones disponibles varían de un lenguaje a otro.
  - A veces la colección que necesitamos está directamente disponible en el lenguaje. Ej.: arrays en C.
  - Otras veces el programador debe adaptar y modelar los tipos disponibles. Ej.: arrays en Python.

Colecciones de tamaño fijo

- **Arrays:** Colección cuyos elementos, todos del mismo tipo, pueden ser accedidos por posición.
  - En los lenguajes que los implementan es necesario definir de antemano su tamaño y el tipo de sus elementos.
  - Pueden tener una o más dimensiones: vectores, matrices, etc.

Colecciones de tamaño fijo

- **Arrays:** Colección cuyos elementos, todos del mismo tipo, pueden ser accedidos por posición.
  - En los lenguajes que los implementan es necesario definir de antemano su tamaño y el tipo de sus elementos.
  - Pueden tener una o más dimensiones: vectores, matrices, etc.
- **Cadenas:** Array unidimensional cuyo elemento base es un carácter.

Colecciones de tamaño fijo

- **Arrays:** Colección cuyos elementos, todos del mismo tipo, pueden ser accedidos por posición.
  - En los lenguajes que los implementan es necesario definir de antemano su tamaño y el tipo de sus elementos.
  - Pueden tener una o más dimensiones: vectores, matrices, etc.
- **Cadenas:** Array unidimensional cuyo elemento base es un carácter.
- **Registros:** Colección formada por un conjunto de campos a cuyos elementos, potencialmente de tipos distintos, se accede por nombre de campo.
  - Dan soporte al concepto de "ficha" sobre un elemento: libro (título, autor, editorial), coche (matrícula, color, marca, modelo), etc.
  - Frecuentemente utilizada como dato base de colecciones más estructuradas: array de registros, pila de registros, etc.

Colecciones de tamaño variable

- **Pilas / Colas:** Colecciones donde lo más importante es el orden de entrada y salida de los elementos.
  - En una pila, el último elemento en entrar es el primero en salir (LIFO).
  - En una cola, el primer elemento en entrar es el primero en salir (FIFO).
  - Variantes: cola con prioridades, cola multinivel, etc.

Colecciones de tamaño variable

- **Pilas / Colas:** Colecciones donde lo más importante es el orden de entrada y salida de los elementos.
  - En una pila, el último elemento en entrar es el primero en salir (LIFO).
  - En una cola, el primer elemento en entrar es el primero en salir (FIFO).
  - Variantes: cola con prioridades, cola multinivel, etc.
- **Lista:** Colección de elementos a los que se accede por posición mediante un índice (a menudo entero).
  - En la mayoría de lenguajes es una estructura homogénea.
  - Variantes: tupla, lista ordenada.

Colecciones de tamaño variable

- **Pilas / Colas:** Colecciones donde lo más importante es el orden de entrada y salida de los elementos.
  - En una pila, el último elemento en entrar es el primero en salir (LIFO).
  - En una cola, el primer elemento en entrar es el primero en salir (FIFO).
  - Variantes: cola con prioridades, cola multinivel, etc.
- **Lista:** Colección de elementos a los que se accede por posición mediante un índice (a menudo entero).
  - En la mayoría de lenguajes es una estructura homogénea.
  - Variantes: tupla, lista ordenada.
- **Conjunto:** Agrupación no ordenada de elementos únicos.

Colecciones complejas de tamaño variable

- **Diccionario:** Colección formada por pares clave-valor en la que el acceso a los elementos se realiza por clave.

Colecciones complejas de tamaño variable

- **Diccionario:** Colección formada por pares clave-valor en la que el acceso a los elementos se realiza por clave.
- **Archivo:** Almacenamiento externo (no en memoria principal) de datos bien homogéneos o bien heterogéneos.
  - Debe ser abierto antes de su manejo y cerrado al término del mismo.
  - Variantes: archivo de texto, binario, indizado, etc.

Colecciones complejas de tamaño variable

- **Diccionario:** Colección formada por pares clave-valor en la que el acceso a los elementos se realiza por clave.
- **Archivo:** Almacenamiento externo (no en memoria principal) de datos bien homogéneos o bien heterogéneos.
  - Debe ser abierto antes de su manejo y cerrado al término del mismo.
  - Variantes: archivo de texto, binario, indizado, etc.
- **Otros:** Tablas hash, arrays dinámicos (arraylists), etc.

Colecciones de datos en Python

- Al no tener un sistema estricto de tipos de datos, en Python las colecciones son heterogéneas: pueden contener elementos de tipos distintos.
- Cada posición puede referenciar un elemento de tipo simple (entero, real, booleano, string) o compuesto (otras secuencias, instancias de una clase, archivos, etc.).

Mutabilidad

Definición

Aquellos objetos cuyo valor puede cambiar se dice que son **mutables**, a diferencia de aquellos cuyo valor no puede cambiar una vez creados, los cuales se denominan **inmutables**.

- Son inmutables los tipos de datos cuyas variables solo pueden cambiar el valor de todo el objeto (`var = valor`).
- Los tipos compuestos mutables, permiten cambiar el valor de los componentes, de forma independiente, o añadir nuevos elementos
- Mutables en Python: listas, diccionarios y conjuntos.
- Inmutables: números (int, float, complex), cadenas y tuplas.

Listas

Definición

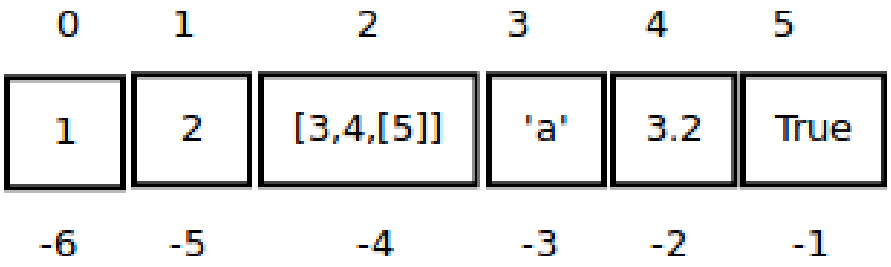
Conjunto mutable y organizado de elementos.

- Se representa encerrando los miembros entre corchetes.
- Los elementos de una lista son variables, pudiendo añadir, modificar y eliminar elementos en cualquier momento.
- Se acceder a los elementos por posición mediante un índice entero
- Rango de los elementos: 0 a longitud-1
- Puede haber elementos repetidos

Lista

```
lista = [1, 2, [3,4,[5]], 'a', 3.2, True]
```

- Representación interna:



- El acceso por posición puede hacerse de izquierda a derecha o de derecha a izquierda, utilizando índices enteros positivos o negativos.

Operaciones enfocadas a elementos

- `lista.index(elem)` - Retorna la posición del elemento en la lista. Si no está se produce una excepción.

Operaciones enfocadas a elementos

- `lista.index(elem)` - Retorna la posición del elemento en la lista. Si no está se produce una excepción.
- `lista[i]` - Accede al elemento en la posición i.

Operaciones enfocadas a elementos

- `lista.index(elem)` - Retorna la posición del elemento en la lista. Si no está se produce una excepción.
- `lista[i]` - Accede al elemento en la posición i.
- `elem in lista` - Determina si un elemento está o no en la lista.

Operaciones enfocadas a elementos

- `lista.index(elem)` - Retorna la posición del elemento en la lista. Si no está se produce una excepción.
- `lista[i]` - Accede al elemento en la posición i.
- `elem in lista` - Determina si un elemento está o no en la lista.
- `lista.count(elem)` - Número de veces que aparece un elemento.

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.
- `lista.insert(pos,elem)` - inserta un elemento en una posición.

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.
- `lista.insert(pos,elem)` - inserta un elemento en una posición.
- `lista.remove(elem)` - Elimina la primera aparición del elemento, si no está, produce una excepción.

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.
- `lista.insert(pos,elem)` - inserta un elemento en una posición.
- `lista.remove(elem)` - Elimina la primera aparición del elemento, si no está, produce una excepción.
- `lista.sort()` - Ordena la lista

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.
- `lista.insert(pos,elem)` - inserta un elemento en una posición.
- `lista.remove(elem)` - Elimina la primera aparición del elemento, si no está, produce una excepción.
- `lista.sort()` - Ordena la lista
- `lista.reverse()` - Invierte el orden de los elementos de la lista

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.
- `lista.insert(pos,elem)` - inserta un elemento en una posición.
- `lista.remove(elem)` - Elimina la primera aparición del elemento, si no está, produce una excepción.
- `lista.sort()` - Ordena la lista
- `lista.reverse()` - Invierte el orden de los elementos de la lista
- `lista.pop()` - Elimina y retorna el último elemento

Operaciones sobre listas

- `len(lista)` - retorna el tamaño de la lista (num. elementos).
- `lista.append(elemento)` - añade un elemento al final de la lista.
- `L1.extend(L2)` - añade a L1 todos los elementos de L2.
- `lista1 + lista2` - produce una lista con los elementos de ambas.
- `lista.insert(pos,elem)` - inserta un elemento en una posición.
- `lista.remove(elem)` - Elimina la primera aparición del elemento, si no está, produce una excepción.
- `lista.sort()` - Ordena la lista
- `lista.reverse()` - Invierte el orden de los elementos de la lista
- `lista.pop()` - Elimina y retorna el último elemento
- + (concatenar) y * (repetir): Ej. lista1 = lista2 * 3

Iteración

- Las secuencias son el elemento base de la iteración con `for`.

```
frutero = ['pera', 'higo', 'fresa', 'caqui']
for fruta in frutero:
    print ("Una pieza de fruta: ", fruta)
```

Slicing

- Forma abreviada de obtener subsecuencias a partir de secuencias existentes
- Sintaxis: `secuencia [inicio_incluido:fin_excluido:salto]`
- Si no se especifica inicio, se entiende desde 0, si no se especifica fin, se entiende hasta el final de la secuencia

Slicing

- Forma abreviada de obtener subsecuencias a partir de secuencias existentes
- Sintaxis: `secuencia [inicio_incluido:fin_excluido:salto]`
- Si no se especifica inicio, se entiende desde 0, si no se especifica fin, se entiende hasta el final de la secuencia
- Ejemplos:

```
preposiciones = ['a', 'ante', 'bajo', 'cabe', 'con', '
    contra', 'de', 'desde', 'en']
a = preposiciones [1:5]  # copia de la posicion 1 a la 4
     a=['ante', 'bajo', 'cabe', 'con']
b = preposiciones [4:]  # copia de la posicion 4 al final
     b=['con','contra', 'de', 'desde', 'en']
c = preposiciones [:3]  # copia de la posicion 0 a la 2
    c=['a', 'ante', 'bajo']
d = preposiciones[:] # hace un duplicado
```

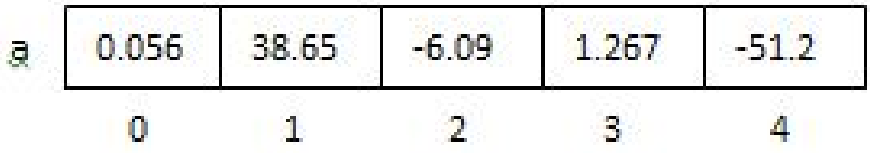
Tupla

Definición

Conjunto inmutable y ordenado de elementos.

- Se declara y representa encerrando los miembros entre paréntesis.
- Número fijo de elementos: sólo se pueden añadir y eliminar reasignando la tupla completa.
- Ejemplo:

$$a = (0.056,\ 38.65, -6.09, 1.267, -51.2)$$



Operaciones con tuplas

- Sus métodos son todos de *solo-lectura*.
  - No pueden añadirse elementos (no tienen métodos `append` ni `extend`).
  - No pueden eliminarse elementos (no tienen `remove` ni `pop`).
- Pueden utilizarse como claves en un diccionario (las listas no).
- Pueden 'convertirse' en listas: `list()` toma una tupla y devuelve una lista con los mismos elementos
  - Es posible también 'convertir' una lista a tupla: `tuple()`
  - Se dice que `tuple` "congela" una lista, y que a su vez `list` "descongela" una tupla.

String

Definición

Colección homogénea cuyos elementos son caracteres de texto.

- Número fijo de elementos: no se pueden añadir ni eliminar.
- Sus elementos no se pueden modificar.
- Creación: texto entre comillas simples o dobles.
- Ejemplo:

```
cadena1 = "Universidad Rey Juan Carlos"
cadena2 = 'Campus de Mostoles'
```

Operaciones con cadenas

- `cadena.upper()` - Genera una copia de la cadena donde las letras que aparezcan ahora van en mayúsculas
- `cadena.lower()` - Genera una copia en minúsculas
- `cadena.find(car)` - Obtiene la posición en la cadena del carácter `car` en la cadena. Si no lo encuentra devuelve -1
- `cadena.replace(car,car2)` - Genera una copia de la cadena donde se ha sustituido car por car2

Operaciones con cadenas

- `cadena.strip()` - Elimina espacios iniciales y finales de una cadena
  - `rstrip()`: variante para eliminar solo los caracteres al final de la cadena (lado derecho)
  - `lstrip()` elimina espacios iniciales (lado izquierdo).
  - Con cualquiera de las 3 variantes puede indicarse que elimine cualquier símbolo o conjunto de ellos: `cadena.strip('!')`, `cadena.rstrip('!. ')`
- `cadena.split()` - Retorna una lista con las partes de la cadena separadas (usando espacios)
  - Se puede usar `split()` con cualquier separador si se indica: `split('.')`, `split(':')`, etc.
- Otras operaciones:
  - cad1 + cad2: genera una cadena con la concatenación de ambas cadenas
  - cad `*i`: genera una cadena donde se repite i veces la cadena original

Diccionarios

Definición

Estructura de datos sin orden entre los elementos, y en la que el acceso viene determinado por una clave única que se asocia a cada valor.

- La clave es a menudo una cadena de texto, si bien puede ser cualquiera de los tipos inmutables de Python.
- Tipos de claves: boolean, integer, float, tupla, string ...
- Los diccionarios son mutables: se pueden agregar, eliminar y cambiar sus elementos.

Crear diccionarios

- Para crear un diccionario se emplean corchetes ({}) alrededor de pares `clave:valor` separados por comas.
- El diccionario más simple es un diccionario vacío, el cual no contiene ninguna clave o valor en absoluto:

```
>>> empty_dict = {}
>>> empty_dict
{}
```

Ejemplo de diccionarios

- Ejemplo preliminar de diccionario con citas del Diccionario de la Real Academia de la Lengua:

```
>>> rae = {
    "colapso": "Destruccion, ruina de una institucion, sistema, estructura, etc.",
    "desinformacion": "Accion y efecto de desinformar.",
    "todologo": "Persona que cree saber y dominar varias especialidades.",
   }
>>>
```

- Al escribir el nombre del diccionario en el intérprete se imprimirán sus claves y valores:

```
>>> rae
{'colapso': 'Destruccion, ruina de una institucion, sistema, estructura, etc.', '
     desinformacion': 'Accion y efecto de desinformar.', 'todologo': 'Persona que
     cree saber y dominar varias especialidades.'}
>>>
```

Operaciones con diccionarios

- **Agregar** un elemento: basta con referirse al ítem por su clave y asignarle un valor.
  - Si la clave ya estaba en el diccionario, el valor existente se reemplaza por el nuevo, si la clave es nueva, se agrega al diccionario con su valor.
  - Imposible errar por especificar un índice fuera de rango.
- **Eliminar** un elemento: utilizar `del`. Ejemplo: `del rae['colapso']`
- **Reiniciar** eliminando todos los elementos: `clear`. Ejemplo: `rae.clear()`
- **Pertenencia** de una clave: `in`. Ejemplo: `'colapso' in rae`
- **Número de elementos:** `len(diccionario)`
- **Réplica:** `nuevo_diccionario = diccionario.copy()`

Más operaciones con diccionarios

- **Lista de claves**: `keys`. Ejemplo: `rae.keys()`
  - Retorna un objeto de tipo `dict_keys`, que es una forma iterable de lista. Puede convertirse a lista con `list()`.
- **Lista de valores**: `values`. Ejemplo: `rae.values()`
  - Retorna un objeto de tipo `dict_values`, una forma iterable de lista.
  - Ejemplo: `for item in x.values(): print item`
- **Lista de items**: lista de tuplas (clave,valor) que al igual que `values` y `keys`, es iterable: `diccionario.items()`

Resumen

- Las colecciones son abstracciones que facilitan el manejo de datos agrupados.
- Todo lenguaje de programación implementará unicamente algunas colecciones, otras deben ser modeladas por el programador
- Una lista en Python es una secuencia de elementos mutable (número de elementos variable + elementos modificables)
- Una tupla en Python es una colección de elementos inmutable (número de elementos fijo + elementos no modificables)
- en Python el acceso a las posiciones y muchas otras operaciones sobre secuencias son comunes a todas las secuencias
- Las cadenas de texto son secuencias inmutables de elementos homogéneos
- Los diccionarios contienen parejas clave-valor

Tema IV: Estructuras de datos

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Noviembre de 2025



Tema V: Recursividad y algoritmos básicos

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Noviembre de 2025



Licencia

© 2025.

Algunos derechos reservados.

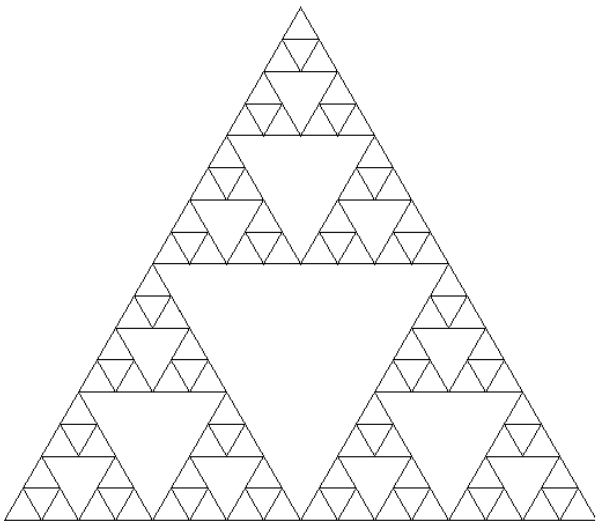
- Isaac Lozano Osorio (isaac.lozano@urjc.es)
- Salvador Sánchez Alonso (salvador.sanchez@urjc.es)

Este documento se distribuye bajo la licencia según la licencia Compartir Igual 4.0 (CC BY-SA 4.0)



Concepto de recursividad

- Definir un objeto o proceso en función de sí mismo.
- Es útil para resolver problemas por descomposición en subproblemas.
  - Los subproblemas tienen idéntica estructura al problema original, pero el caso que resuelven está reducido o es más pequeño.



Definiciones

Definición

Un **objeto** recursivo es aquel que forma parte de sí mismo o se define en función de sí mismo

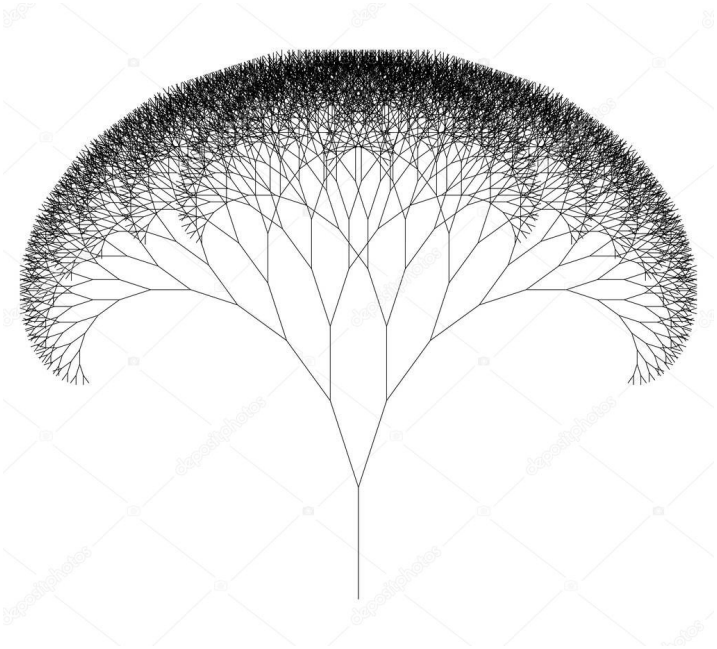
Definición

Un **subprograma** recursivo es aquél que contiene llamadas a sí mismo.

```
n! = n * (n-1)!
```

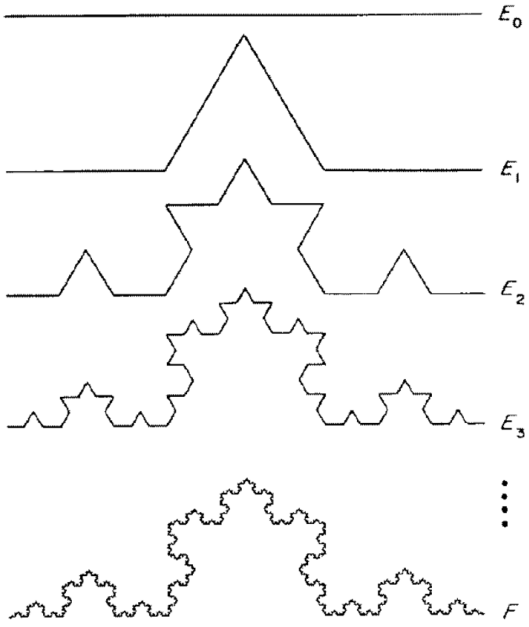
Resolución de un problema mediante recursividad

- Descomponer el problema en subproblemas de más fácil solución.
- Descomponer a su vez cada subproblema hasta alcanzar un problema con solución trivial: el caso básico.
- El subproblema y el problema deben tener la misma estructura.
- La descomposición debe permitir alcanzar el caso básico en un número finito de pasos.



Ejecución de un subprograma recursivo

- Cada vez que se invoca una resolución de un subproblema, el problema actual queda suspendido y pendiente de resolución (apilado)
- La resolución efectiva comienza una vez se alcanza el caso básico.
- Cada vez que se termina de resolver un subproblema, se retorna el resultado al problema que lo invocó (desapilado)



Esquema básico de una función recursiva

```
funcion R (problema)

  si el problema es suficientemente simple # caso basico
    resolverlo directamente

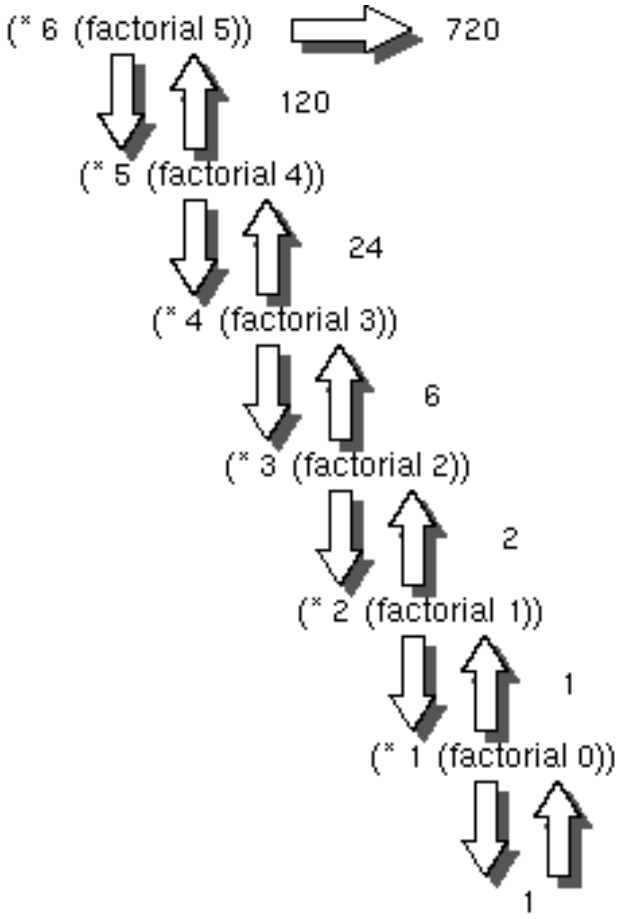
  si_no # descomponer el problema en subproblemas

    # llamada recursiva
    por cada subproblema s, invocar R(s)
```

Cómo diseñar algoritmos recursivos

- Identificar el problema a resolver.
- Identificar el/los casos que se resuelven directamente, de manera sencilla: caso o problema básico.
- Identificar cómo se puede descomponer un caso cualquiera no básico para reducir el problema a uno o varios subproblemas más pequeños que tiendan al caso básico.
    - Si no tienden al caso básico ¡se desbordará la pila de invocaciones en tiempo de ejecución!

Ejemplo: Factorial

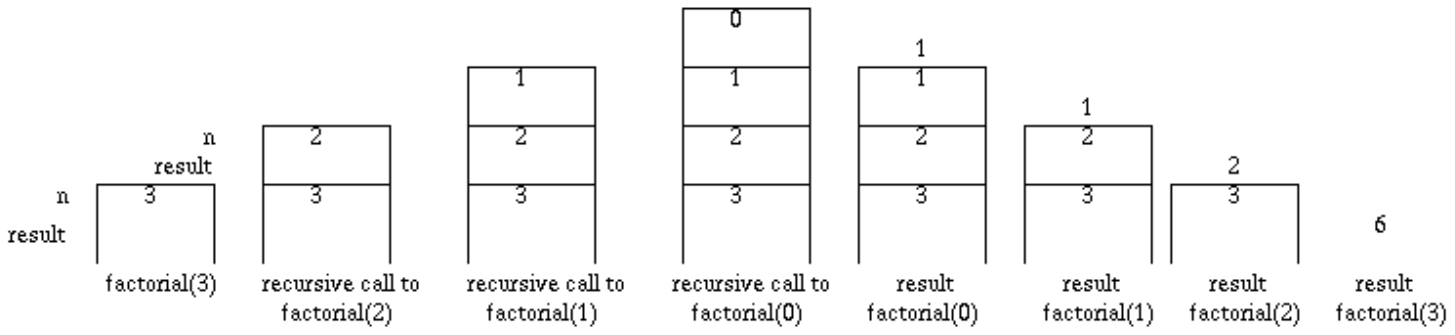


Ejemplo: Factorial

```
def factorial (n):
    """ int -> int
        OBJ: Factorial de n
        PRE: n >= 0
    """
    if (n==0):  resultado = 1
    else:       resultado = n * factorial (n-1)
    return resultado
```

Ejemplo: Factorial

```
def factorial (n):
    """ int -> int
        OBJ: Factorial de n
        PRE: n >= 0
    """
    if (n==0):  resultado = 1
    else:       resultado = n * factorial (n-1)
    return resultado
```



Análisis de un problema recursivo

Problema: Multiplicar dos números por sumas sucesivas.

Análisis de un problema recursivo

Problema: Multiplicar dos números por sumas sucesivas.

- Cualquier número multiplicado por 1 es ese número $\rightarrow$ caso básico

Análisis de un problema recursivo

Problema: Multiplicar dos números por sumas sucesivas.

- Cualquier número multiplicado por 1 es ese número $\rightarrow$ caso básico
- Para multiplicar 6 por 4 podemos hacer la siguiente descomposición: «Sumamos 6 a la multiplicación de 6 por 3»

Análisis de un problema recursivo

Problema: Multiplicar dos números por sumas sucesivas.

- Cualquier número multiplicado por 1 es ese número $\rightarrow$ caso básico
- Para multiplicar 6 por 4 podemos hacer la siguiente descomposición: «Sumamos 6 a la multiplicación de 6 por 3»
- El problema original es 6*4 y lo hemos reducido a una operación de suma y a la resolución del subproblema 6*3

Análisis de un problema recursivo

Problema: Multiplicar dos números por sumas sucesivas.

- Cualquier número multiplicado por 1 es ese número $\rightarrow$ caso básico
- Para multiplicar 6 por 4 podemos hacer la siguiente descomposición: «Sumamos 6 a la multiplicación de 6 por 3»
- El problema original es 6*4 y lo hemos reducido a una operación de suma y a la resolución del subproblema 6*3
- Es posible también descomponer 6*3 en 6 + (6*2) y 6*2 en 6+(6*1)

Ejemplo: Código en Python

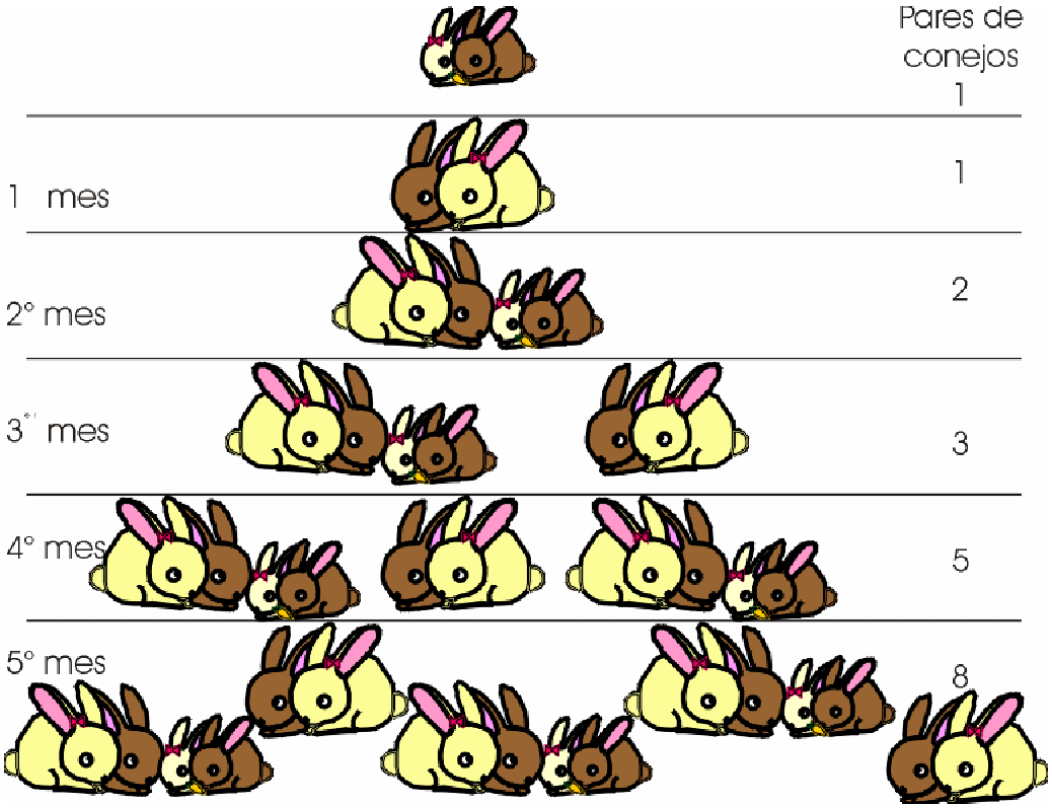
```
def multiplicar (n, m):
  """ int, int -> int
      OBJ: Multiplicacion n x m mediante sumas sucesivas
  """
  if (m==1):
     multiplicacion = n
  else:
     multiplicacion = n + multiplicar(n, m-1)
  return multiplicacion
```

Ejemplo: Serie de Fibonacci

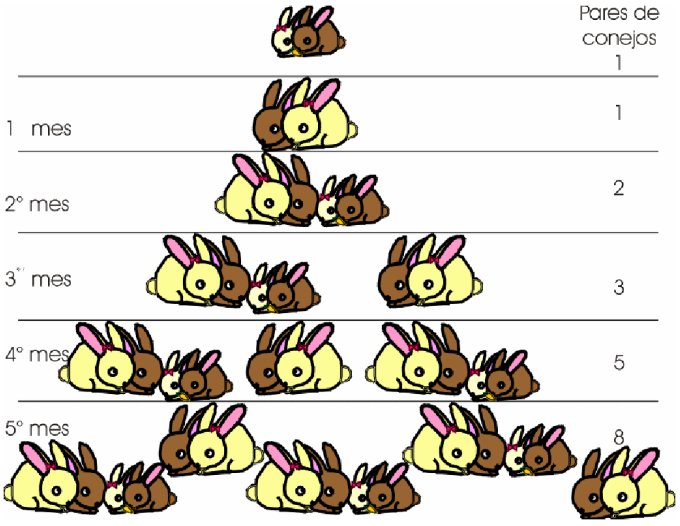
Fibonacci se planteó el número de parejas de conejos en una granja al cabo de varios meses, si se parte de una sola pareja y asumiendo que...

- Los conejos alcanzan la madurez sexual a la edad de un mes.
- En cuanto alcanzan la madurez sexual los conejos se aparean.
- El periodo de gestación de los conejos es de un mes.
- Los conejos nunca mueren.
- Los conejos adultos se aparean regularmente cada mes.
- La hembra siempre pare una pareja de conejos de sexos opuestos.
- Los conejos ni tienen moral ni instinto de variedad genética, por lo que se aparean entre parientes.

Ejemplo: Serie de Fibonacci



Ejemplo: Serie de Fibonacci




```
def fibonacci(n):
    """ int -> int
        OBJ: Valor de la serie de Fibonacci de orden n
        PRE: n>=0
    """
    if n <= 1: f = 1
    else:  f = fibonacci(n-2) + fibonacci(n-1)
    return f
```

Ejemplo: Qué hace el siguiente código?

```
def misterio (n):
  if (n==1):
    c = input('Introduce un caracter: ')
    print(c)
  else:
    c = input('Introduce un caracter: ')
    misterio(n-1)
    print(c)
```

Ejemplo: Qué hace el siguiente código?

```
def misterio (n):
  if (n==1):
    c = input('Introduce un caracter: ')
    print(c)
  else:
    c = input('Introduce un caracter: ')
    misterio(n-1)
    print(c)
```

Entrada: A S D F G
Salida: G F D S A

Ejemplo: Palíndromo



Ejemplo: Palíndromo




```
def es_palindromo (pal, inicio, fin):
  """ str, int, int -> bool
      OBJ: Calcula si un texto entre 2 indices es palindromo
      PRE: inicio, fin >= 0
  """
  if inicio >= fin: result = True
  else:
    if pal[inicio]!=pal[fin]: result = False
    else: result = es_palindromo(pal,inicio+1,fin-1)
  return result
```

Ejemplo: Palíndromo (con *slicing*)

```
def es_palindromo (frase):
    """ str -> bool
        OBJ : Determina si una frase puede leerse igual al
            derecho o al reves ( palindromo )
        PRE: La frase no contiene signos de puntuacion"""
    frase = frase.replace(' ','')    # elimina espacios
    if len(frase) <=1: result = True
    elif frase[0] != frase[-1]: result = False
    else: result = es_palindromo(frase [1:-1])
    return result

# Probador
print(es_palindromo('campeon')) # False
print(es_palindromo('somos o no somos')) # True
```

Ejemplo: Suma del contenido de una lista

Ejemplo: Suma del contenido de una lista

```
def suma_elementos(lista, ini, fin):
  """ list, int, int -> float
      OBJ: suma elementos de una lista entre dos posiciones
      PRE: ini, fin >= 0
  """
  if len(lista) == 0:  suma = 0
  elif    (fin==ini):  suma = lista[ini]
  else: suma = lista[ini] + suma_elementos(lista,ini+1,fin)
  return suma
```

Ejemplo: Suma del contenido de una lista

```
def suma_elementos(lista, ini, fin):
  """ list, int, int -> float
      OBJ: suma elementos de una lista entre dos posiciones
      PRE: ini, fin >= 0
  """
  if len(lista) == 0:  suma = 0
  elif    (fin==ini):  suma = lista[ini]
  else: suma = lista[ini] + suma_elementos(lista,ini+1,fin)
  return suma
```

Usando *slicing*:

```
def suma_todos_elementos(lista):
  """ lista -> float
      OBJ: suma todos los elementos de una lista
  """
  if len(lista) == 0: suma = 0
  else: suma = lista[0] + suma_todos_elementos (lista[1:])
  return suma
```

Ejemplo: Mostrar los elementos de una lista

Ejemplo: Mostrar los elementos de una lista

```
def imprimir(lista, ini, fin):
  """ list, int, int -> None
      OBJ: muestra por pantalla los elementos de una lista
      PRE: ini, fin >= 0
  """
  print(lista[ini])
  if (fin!=ini): imprimir(lista,ini+1,fin)
```

Ejemplo: Mostrar los elementos de una lista

```
def imprimir(lista, ini, fin):
  """ list, int, int -> None
      OBJ: muestra por pantalla los elementos de una lista
      PRE: ini, fin >= 0
  """
  print(lista[ini])
  if (fin!=ini): imprimir(lista,ini+1,fin)
```

- Si queremos que imprima el contenido al revés...

Ejemplo: Mostrar los elementos de una lista

```
def imprimir(lista, ini, fin):
  """ list, int, int -> None
      OBJ: muestra por pantalla los elementos de una lista
      PRE: ini, fin >= 0
  """
  print(lista[ini])
  if (fin!=ini): imprimir(lista,ini+1,fin)
```

- Si queremos que imprima el contenido al revés...

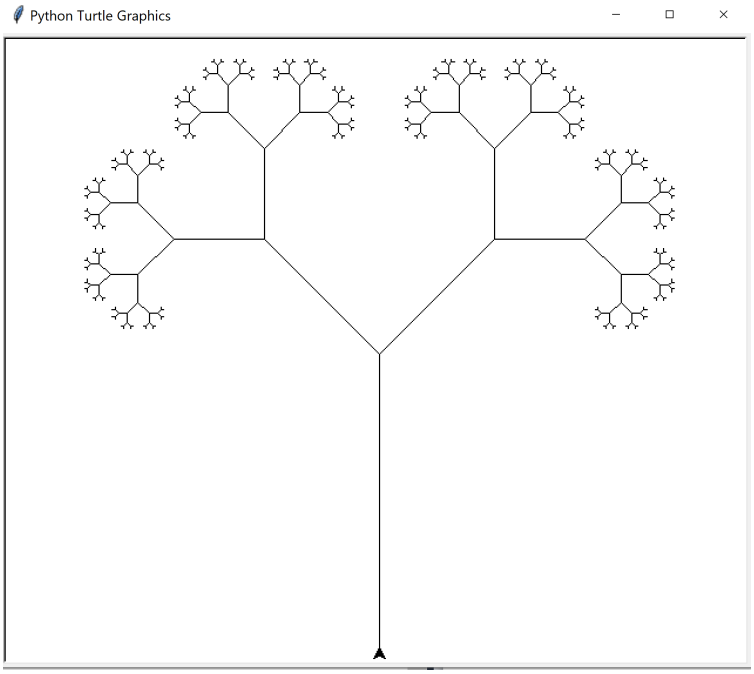
```
def imprimir(lista, ini, fin):
  """ list, int, int -> None
      OBJ: muestra por pantalla los elementos de una lista
      PRE: ini, fin >= 0
  """
  if (fin!=ini): imprimir(lista,ini+1,fin)
  print (lista[ini])
```

Ejercicios propuestos

- Codifica un subprograma recursivo que calcule la división entera entre dos números A y B sin utilizar multiplicaciones ni divisiones.
- Implementa un subprograma recursivo que calcule x^n, siendo x un número real y n un entero.
- Elabora un programa recursivo que tome una cadena como parámetro y devuelva otra cadena que sea la original pero con sus caracteres invertidos.
- Implementa una función recursiva en Python que cuente el número de vocales que hay en una frase.

Tarea de investigación

- La biblioteca `turtle` de Python permite simular un antiguo lenguaje gráfico de programación llamado Logo que fue creado con fines didácticos. Investiga por tu cuenta y aprende a dibujar un árbol fractal con recursividad, Python y la tortuga de Logo.



Resumen

- La recursividad es una técnica de programación que facilita la codificación de programas descomponiendo el problema a resolver en problemas más pequeños.
- Un subprograma recursivo contiene llamadas a sí mismo.
- Los subprogramas recursivos deben tener una condición de parada cuando se llegue al caso (o casos) básico(s) y una o más llamadas recursivas con parámetros que converjan al caso base.
- Es importante comprender cómo se apilan las llamadas para entender el resultado de la recursividad.

Algoritmia

Definición

Ciencia que estudia los algoritmos, con un énfasis especial en su eficiencia.

- **Eficiencia**: menor consumo posible de tiempo, espacio de memoria y operaciones del procesador.
- **Complejidad algoritmica**: mide la eficiencia de un algoritmo determinando el número de recursos necesario para su ejecución

Medidas de complejidad

- Complejidad **temporal**: consiste en contar el número de operaciones básicas (ej. sumas, multiplicaciones) que realiza el algoritmo.
- Complejidad **espacial**: consiste en determinar la cantidad de memoria que emplea el algoritmo al ejecutarse.
- Empleamos siempre medidas:
  - Relativas al tamaño del problema (no absolutas ej. segundos, bytes...)
  - Independientes de la máquina, la plataforma y el lenguaje utilizado.

Complejidad temporal

- El tiempo de ejecución de un algoritmo puede variar entre diferentes entradas del mismo tamaño
- Consideraremos siempre el caso más desfavorable: comportamiento asintótico (notación O).
- La complejidad temporal permite comparar diferentes algoritmos que resuelven una misma tarea:
  - Preferiremos siempre el algoritmo que realice la tarea en el menor número de operaciones (más eficiente).

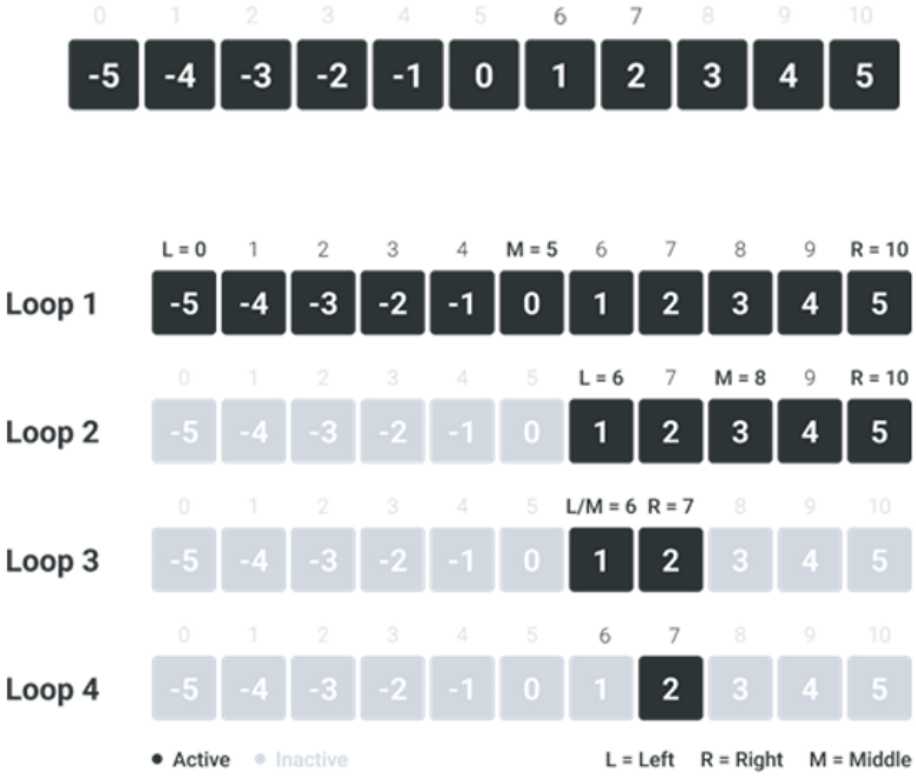
Notación O grande (Big-O)

- Notación para comparar la complejidad de diferentes algoritmos.
- Establece una cota superior asintótica, i.e. una función que hace de cota superior de la función "número de operaciones de nuestro algoritmo"
- Aplicación práctica:
  - Todos los algoritmos básicos de ordenación son $O(n^2)$.
  - Las diferencias entre los métodos básicos radica en la cuenta de algunas operaciones, por ejemplo:
    - Número de comparaciones entre claves: Relevante cuando la comparación es costosa.
    - Número de movimientos de elementos: Relevante cuando el elemento es grande y hay que optimizar el uso de memoria.

Complejidad temporal de las búsquedas

- **Búsqueda lineal**: En el peor caso (buscar un elemento que no esté) hace tantos accesos como elementos tiene el array, es decir: n
- **Búsqueda binaria**: En el peor caso (elemento que no está) hace muchas menos operaciones, 4 en nuestro ejemplo: log2(10)=4
- Se demuestra que el número de operaciones de la búsqueda binaria es O(log(n))

Complejidad temporal de las búsquedas



Búsqueda secuencial

- Permite buscar un elemento en una colección.
- Los elementos de la colección no están ordenados o el criterio de ordenación se desconoce.
- Se recorre la colección desde el principio, terminando cuando se encuentra el elemento
- Debe establecerse un criterio para cuando llega al final sin encontrarlo

```
def posicion(lista,elemento):
    """ list, object -> int
        OBJ: Posicion en una lista, None si no esta
    """
    pos = None
    i = 0
    while pos == None and i<len(lista):
        if lista[i] == elemento: pos = i
        i += 1
    return pos
```

Búsqueda secuencial

- La complejidad de los algoritmos se calcula teniendo en cuenta el peor caso posible.
- Para el caso de una b'usqueda en una lista de n elementos, el peor caso es que el elemento no se encuentre en la lista: hay que recorrerla entera (n operaciones).
- A la complejidad de esta operación se la denomina de orden lineal: O(n).

Búsqueda binaria

- Permite buscar un elemento en una colección (v) de manera más eficiente que con una búsqueda secuencial.
- Es imprescindible que la colección tenga los elementos ordenados.
- Se basa en buscar sólo en la parte de la colección que puede contener el elemento:
  - Obtener la posición media de la colección (centro).
  - Si el elemento a buscar es menor que v[centro], se busca en la primera mitad de v.
  - Si el elemento a buscar es mayor que v[centro], se busca en la segunda mitad de v.

Búsqueda binaria

Implementación iterativa:

```
def posicion(A,elemento):
  """ list, object -> int
      OBJ: Posicion en una lista, None si no esta.
           Utiliza el algoritmo de busqueda binaria
      PRE: lista ordenada
  """
  pos = None
  ini = 0
  fin = len(A)-1
  while ini <= fin and pos == None:
    centro = (ini + fin)//2
    if lista[centro] == elemento:  pos = centro
    elif elemento < lista[centro]: fin = centro-1
    else: ini = centro + 1
  return pos
```

Búsqueda binaria

Implementación recursiva:

```
def posicion(A,elem,inicio,fin):
  """ list, object, int, int -> int
      OBJ: Posicion en una lista entre 2 limites, None si no esta
           Utiliza el algoritmo de busqueda binaria recursiva
      PRE: lista ordenada
  """
  if inicio > fin: pos = None
  else:
    centro = (inicio + fin) // 2
    if A[centro] == elem: pos= centro
    elif elem < A[centro]: pos = posicion(A,elem,inicio,centro-1)
    else: pos = posicion(A,elem,centro+1,fin)
  return pos
```

Búsqueda binaria

Implementación recursiva con *slicing*:

```
def posicion (v,elem):
  """ lista, obj -> int
   OBJ: Posicion de un elemento en una lista, None si no esta
        PRE: La lista debe estar ordenada """
  if len(v) == 0: pos = None
  else:
    centro = len(v) // 2
    if (v[centro] == elem):     pos = centro
    else:
      if (len(v) == 1):         pos = None
      elif (elem < v[centro]): pos = posicion(v[:centro], elem)
      else:
          pos = posicion(v[centro+1:], elem)
          # la posicion en la sublista + centro+1 sera la pos.
          # en la lista original, pero solo si el elemento estaba
          if pos != None: pos += centro + 1
  return pos
```

Ejemplo para una lista de clientes

```
def posicion(lista, buscado):
  """ list, int -> int
      OBJ: Posicion en una lista de clientes, None si no esta.
      PRE: lista ordenada por num_cliente
  """
  pos = None
  ini = 0
  fin = len(lista)-1
  while (ini <= fin and pos == None):
    centro = (ini + fin)//2
    if lista[centro]['num_cliente'] == buscado:  pos = centro
    elif buscado < lista[centro]['num_cliente']: fin = centro-1
    else: ini = centro + 1
  return pos

clientes = [
        {
            'nombre':'Angel Perez',
            'num_cliente': 1
        },
        {   'nombre':'Angela Perez',
            'num_cliente': 2
        },
        ...
]
#probador
print(posicion(clientes,2))
```

Introducción: Ordenación

- Se desea ordenar una colección de elementos que poseen clave.
- Clasificar u ordenar consiste en reorganizar una colección de n elementos, de modo que:
  $clave[a_{p(1)}] < clave[a_{p(2)}] < ... < clave[a_{p(n)}]$

Métodos de ordenación básicos

- Poco eficientes ($O(n^2)$): Adecuados para ejemplares de tamaño pequeño.
- Clasificación:
  - Métodos de *intercambio*: Ordenan el ejemplar intercambiando pares de elementos.
  - Métodos de *inserción*: Ordenan el ejemplar introduciendo de manera ordenada cada elemento en la posición que le corresponde dentro de una subsecuencia previamente ordenada.
  - Métodos de *selección*: Ordenan el ejemplar escogiendo de entre los elementos no ordenados el menor (o mayor), y colocándolo a continuación de los que ya están en orden.

Métodos de ordenación avanzados

- Aplican técnicas de diseño de algoritmos a los esquemas básicos (p.e. Merge-sort, Quick-sort)
- Generalización de algoritmos básicos (p.e. Shell)
- Se sustentan en estructuras de datos avanzadas (p.e. Heap-sort)
- Mejoran la eficiencia de los algoritmos básicos.
- Adecuados para ejemplares de gran tamaño.

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5
```

```
22, 10, 12, 10, 1, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 12, 1, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
                                    22, 10, 1, 12, 10, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 12, 1, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 12, 1, 10, 5
```

```
22, 10, 1, 12, 10, 5

22, 10, 1, 12, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 12, 1, 10, 5

22, 10, 1, 12, 10, 5

22, 10, 1, 12, 10, 5

22, 1, 10, 12, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 12, 1, 10, 5

22, 10, 1, 12, 10, 5

22, 10, 1, 12, 10, 5

22, 1, 10, 12, 10, 5

22, 1, 10, 12, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 12, 1, 10, 5

22, 10, 1, 12, 10, 5

22, 1, 10, 12, 10, 5

1, 22, 10, 12, 10, 5
```

Método de la burbuja: Traza (i)

- Es un método de *intercambio*: Ordena intercambiando parejas de elementos
- El elemento de menor clave «flota».
- Se intercambian los elementos para que el menor vaya «ascendiendo».

```
22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

22, 10, 12, 1, 10, 5

22, 10, 1, 12, 10, 5

22, 1, 10, 12, 10, 5

1, 22, 10, 12, 10, 5

1, 22, 10, 12, 10, 5
```

Método de la burbuja

Habrá que hacer tantas ejecuciones de esta función como elementos desordenados tengamos:

```
def ascender_menor(v,primero,ultimo):
  """ lista, int, int -> None
      OBJ: Encuentra el menor en un rango y lo lleva a la
           primera posicion. Modifica la lista.
  """
  for i in range (ultimo,primero,-1):
    if (v[i] < v[i-1]):
      v[i], v[i-1] = v[i-1], v[i]
```

Método de la burbuja: Traza (ii)

22, 10, 12, 10, 1, 5

Método de la burbuja: Traza (ii)

22, 10, 12, 10, 1, 5

1, 22, 10, 12, 10, 5

Método de la burbuja: Traza (ii)

22, 10, 12, 10, 1, 5

1, 22, 10, 12, 10, 5

1, 5, 22, 10, 12, 10

Método de la burbuja: Traza (ii)

22, 10, 12, 10, 1, 5

1, 22, 10, 12, 10, 5

1, 5, 22, 10, 12, 10

1, 5, 10, 22, 10, 12

Método de la burbuja: Traza (ii)

22, 10, 12, 10, 1, 5

1, 22, 10, 12, 10, 5

1, 5, 22, 10, 12, 10

1, 5, 10, 22, 10, 12

1, 5, 10, 10, 22, 12

Método de la burbuja: Traza (ii)

22, 10, 12, 10, 1, 5

1, 22, 10, 12, 10, 5

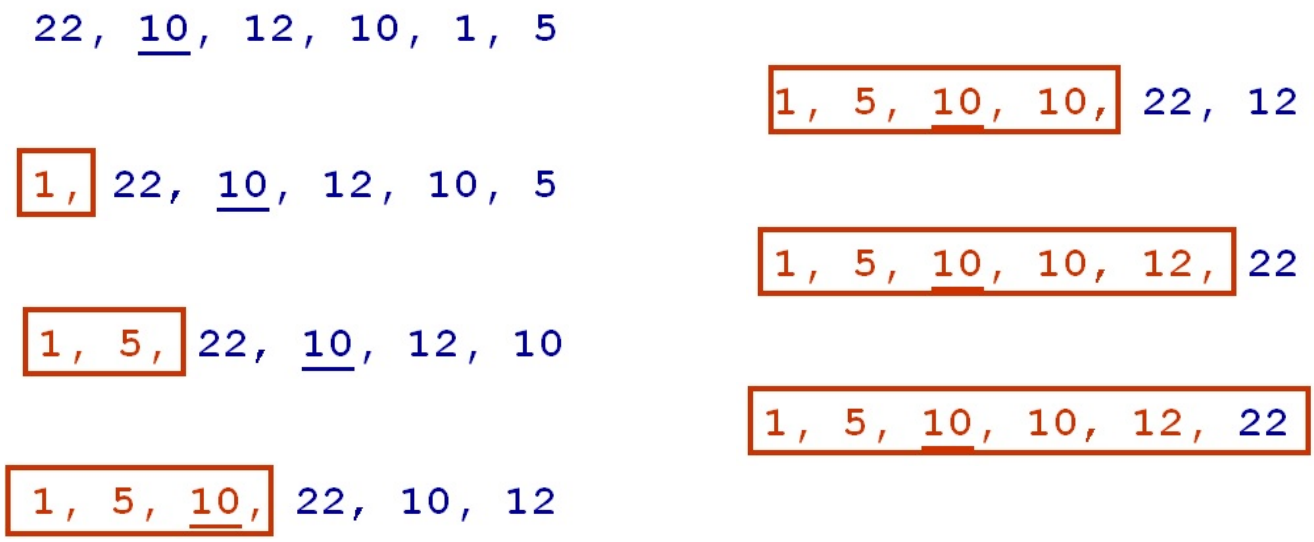
1, 5, 22, 10, 12, 10

1, 5, 10, 22, 10, 12

1, 5, 10, 10, 22, 12

1, 5, 10, 10, 12, 22

Método de la burbuja: Traza (ii)



Método de la burbuja

El algoritmo completo realiza varias pasadas (n-1) invocando la función ascender:

```
def ordenar_burbuja(v,inicio,fin):
  """ lista, int, int -> None
      OBJ: Ordena ascendentemente una lista entre dos
           posiciones. Metodo burbuja.  """
  for pasada in range(inicio,fin-1):
    ascender_menor(v,pasada,fin)
```

Método de la burbuja

El algoritmo completo realiza varias pasadas (n-1) invocando la función ascender:

```
def ordenar_burbuja(v,inicio,fin):
  """ lista, int, int -> None
      OBJ: Ordena ascendentemente una lista entre dos
           posiciones. Metodo burbuja.  """
  for pasada in range(inicio,fin-1):
    ascender_menor(v,pasada,fin)
```

recordando que...

```
def ascender_menor(v,primero,ultimo):
  """ lista, int, int -> None
      OBJ: Encuentra el menor en un rango y lo lleva a la
           primera posicion. Modifica la lista. """
  for i in range (ultimo,primero,-1):
    if (v[i] < v[i-1]):
      v[i], v[i-1] = v[i-1], v[i]
```

Método de la burbuja

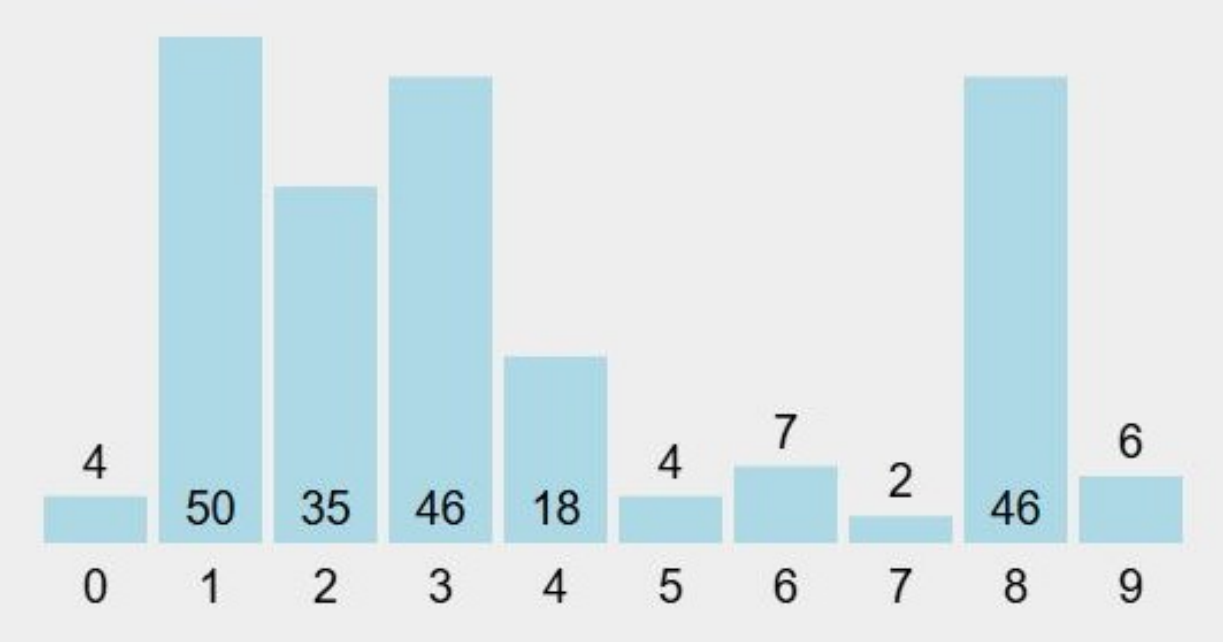
Una implementación más compacta para ordenar una secuencia completa:

```
def ordenar_burbuja(v):
    """ lista -> None
        OBJ: Ordena una lista completa ascendentemente.
             Metodo burbuja. Modifica la lista.
    """
    long = len(v) - 1
    # haremos long - 1 pasadas: 10 elementos -> 9 pasadas
    for pasada in range(0,long):
        # en cada pasada comparamos el elemento i con el i+1
        for i in range (0,long-pasada):
        # cambiamos de posicion si el izquierdo es mayor,
            para ordenar ascendente
            if (v[i] > v[i+1]): v[i], v[i+1] = v[i+1], v[i]
```

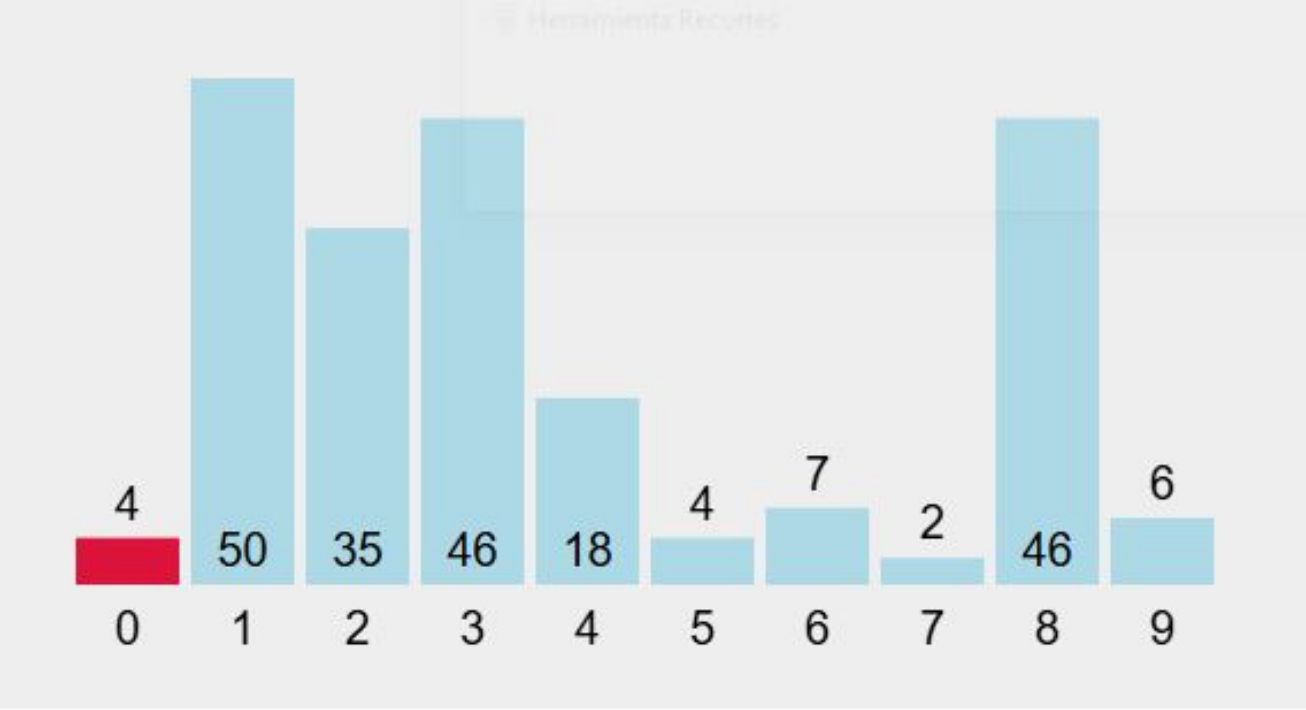
Método de Selección: Traza

- Selecciona el menor elemento de la lista y lo coloca justo a continuación de todos los que ya han sido ordenados.
- Al iniciar el algoritmo, el primer elemento (uno cualquiera de los dos extremos) se considera ordenado.
- El algoritmo de ordenación comienza propiamente con el segundo elemento a ordenar.
- Mejora el rendimiento del burbuja ya que solo hace un intercambio en cada pasada.

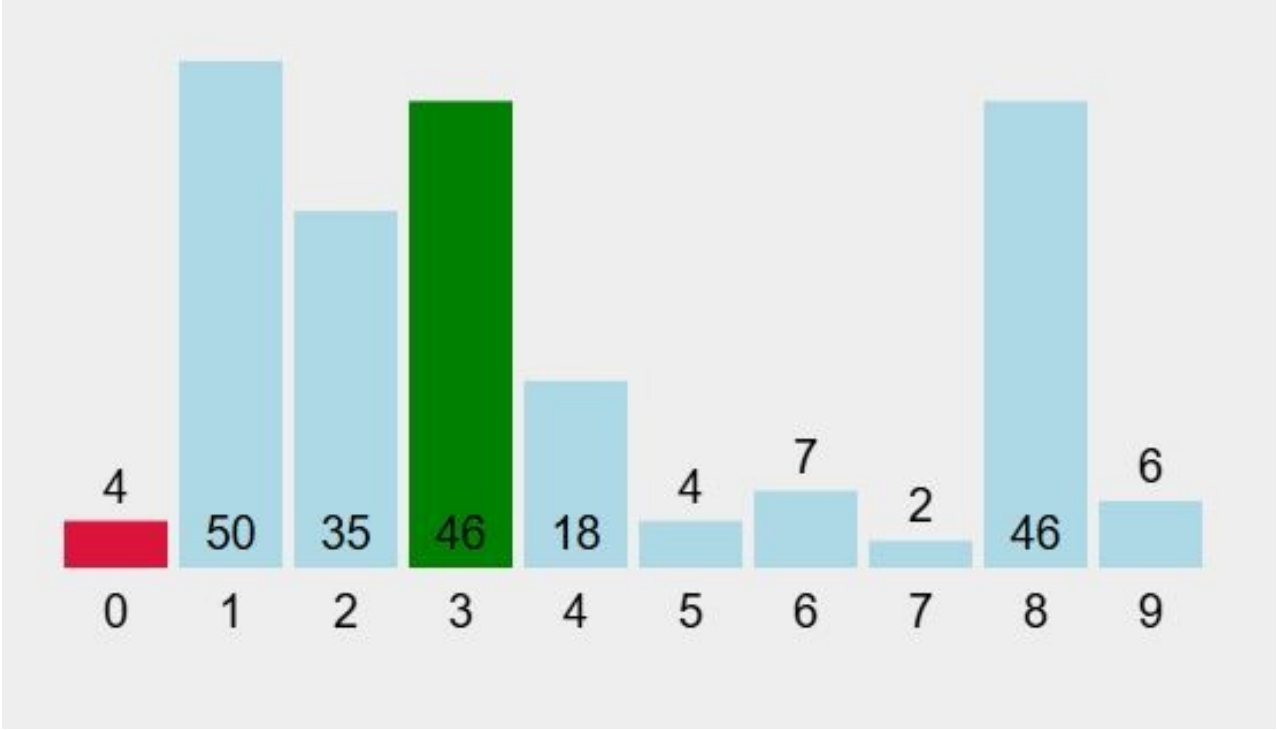
Método de Selección: Animación



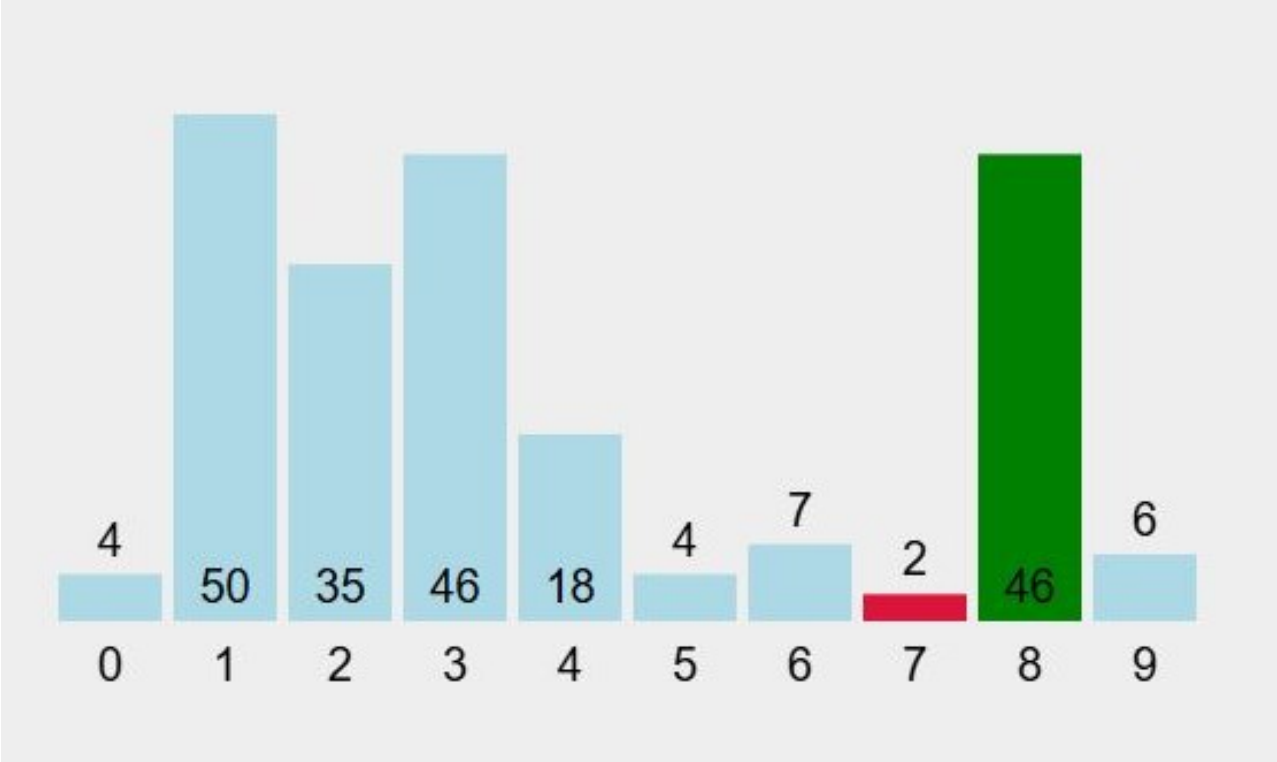
Método de Selección: Animación



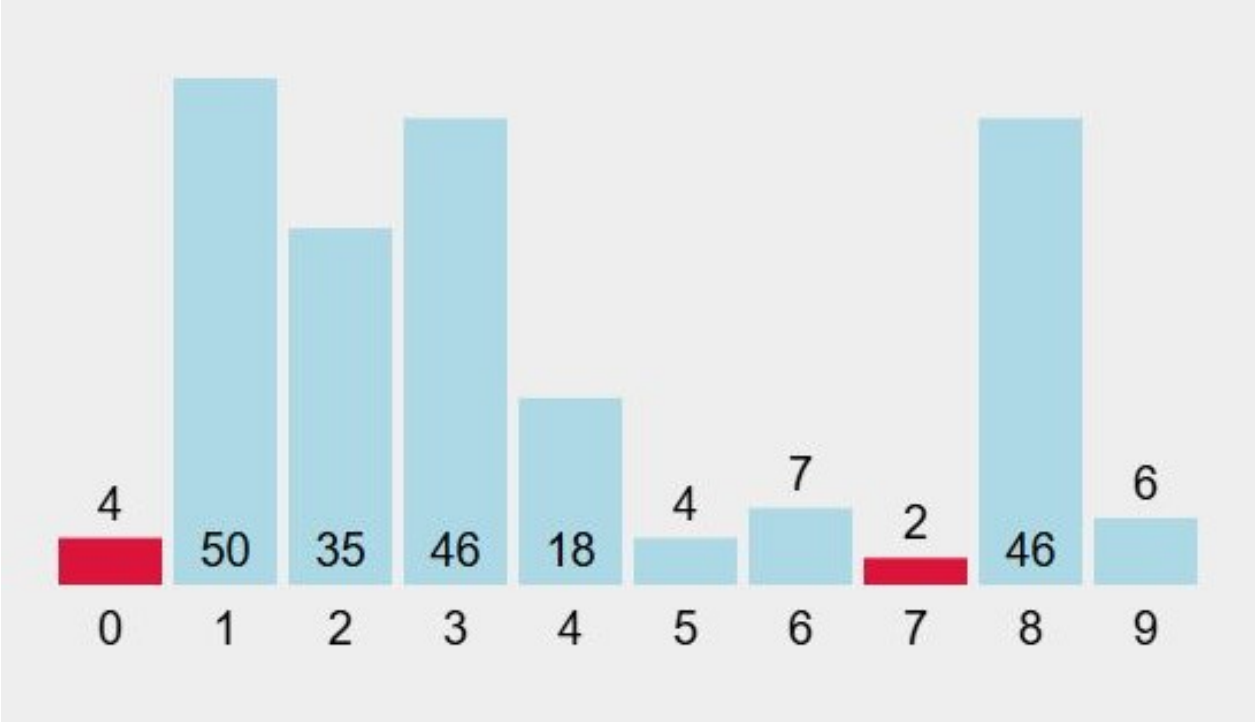
Método de Selección: Animación



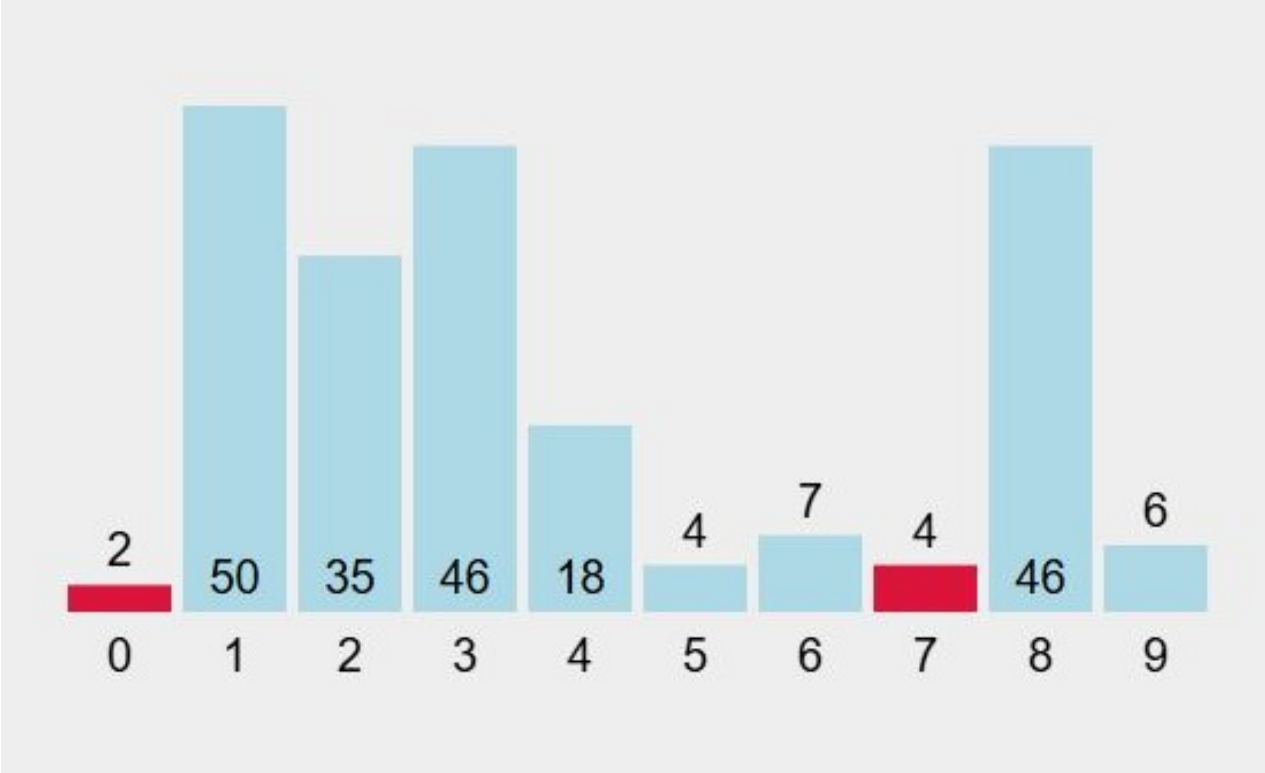
Método de Selección: Animación



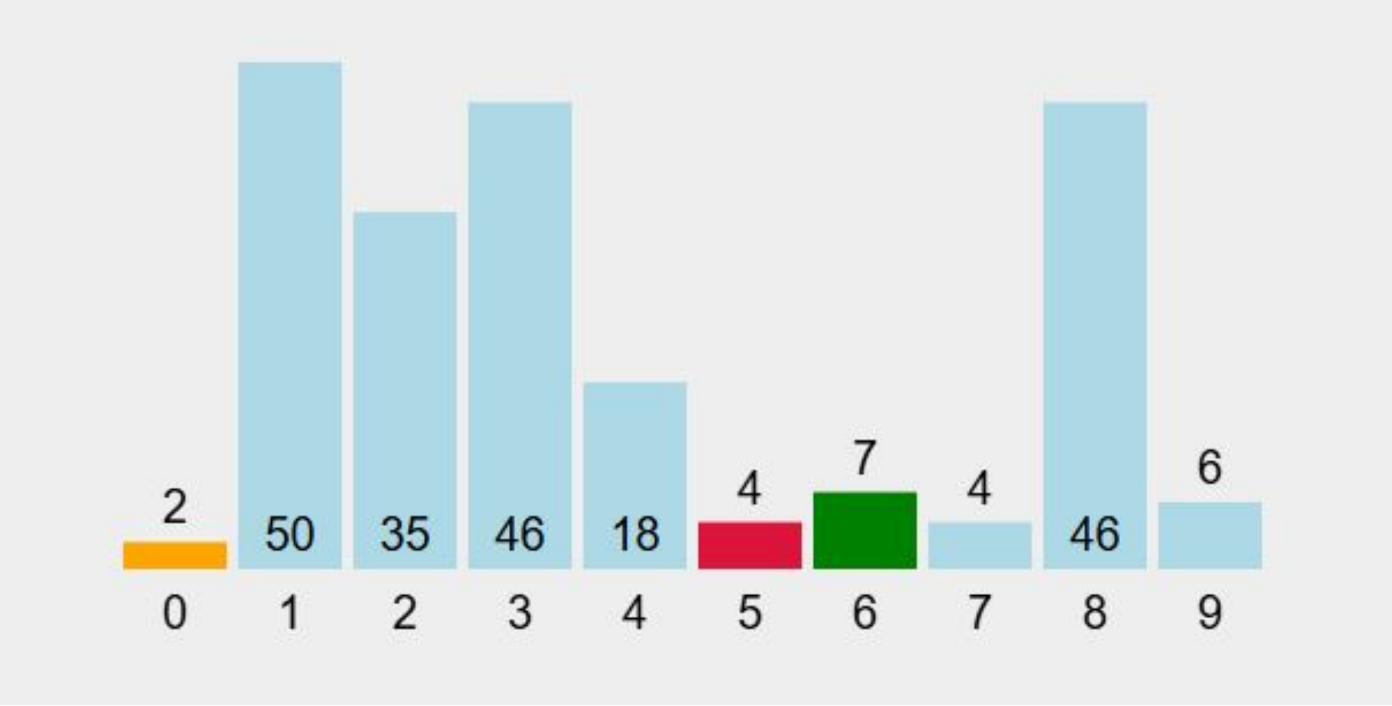
Método de Selección: Animación



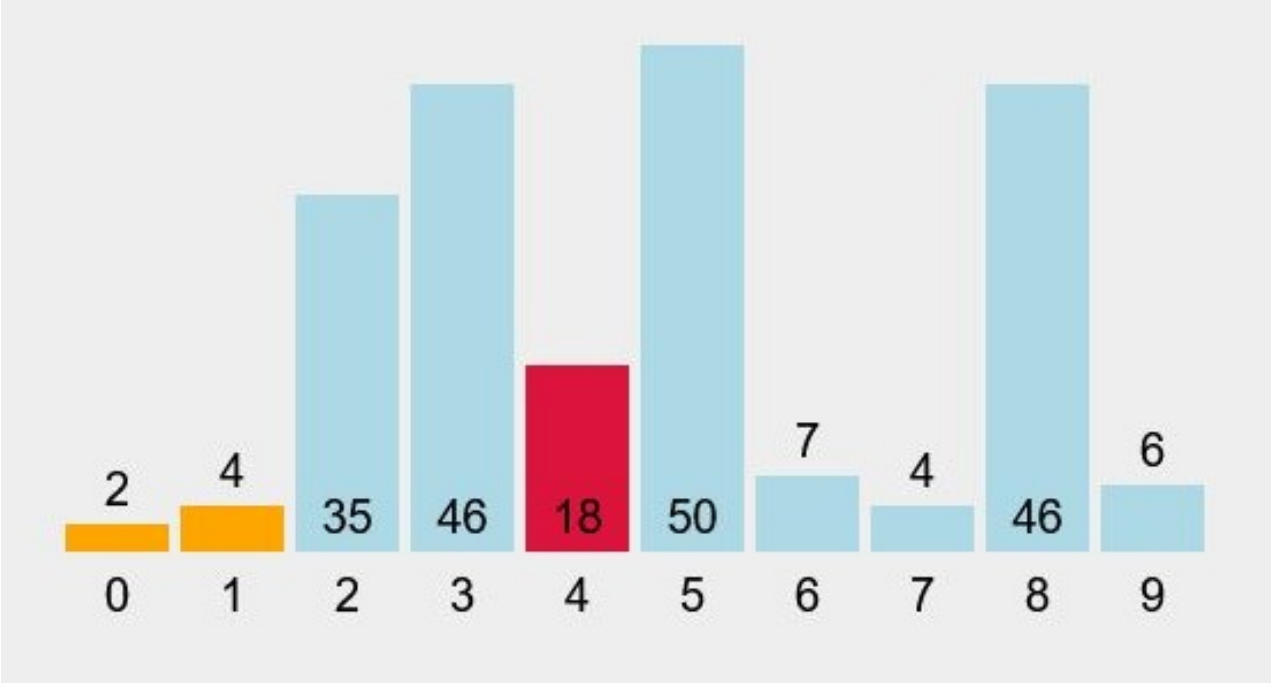
Método de Selección: Animación



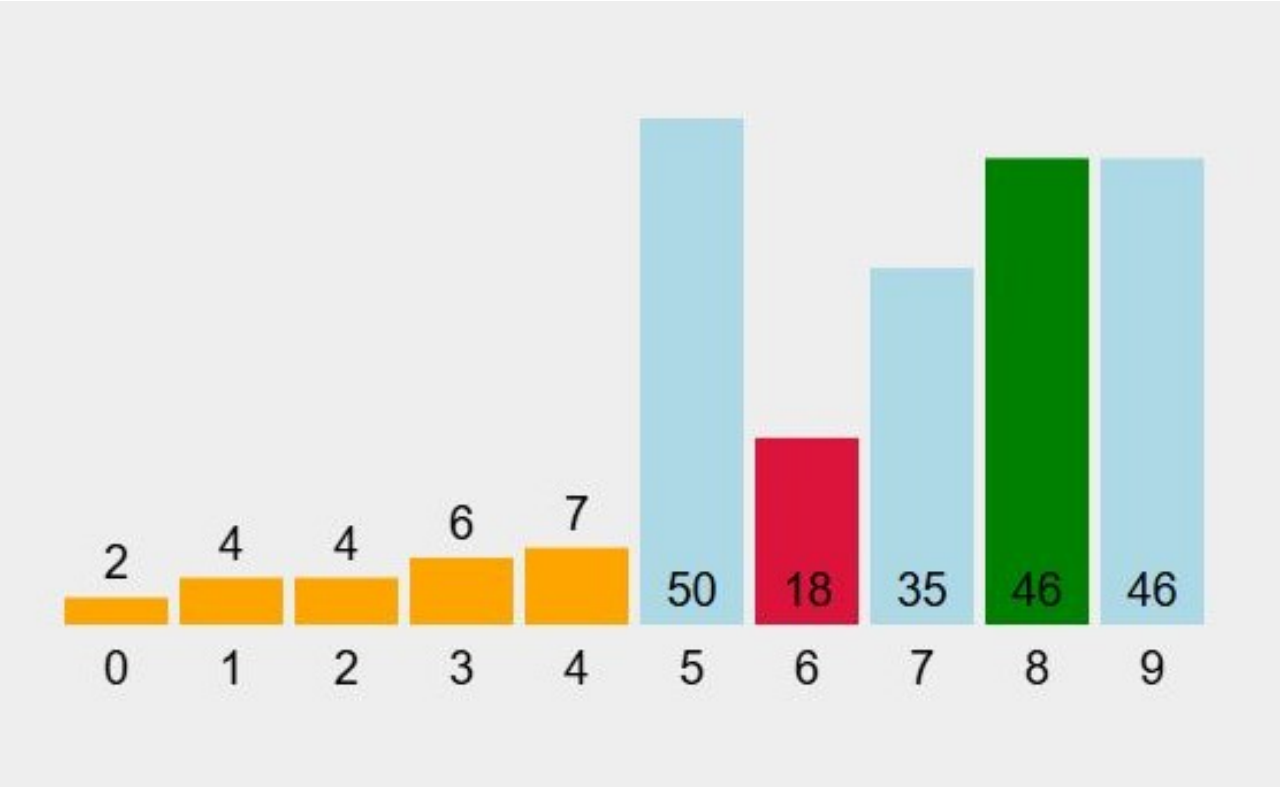
Método de Selección: Animación



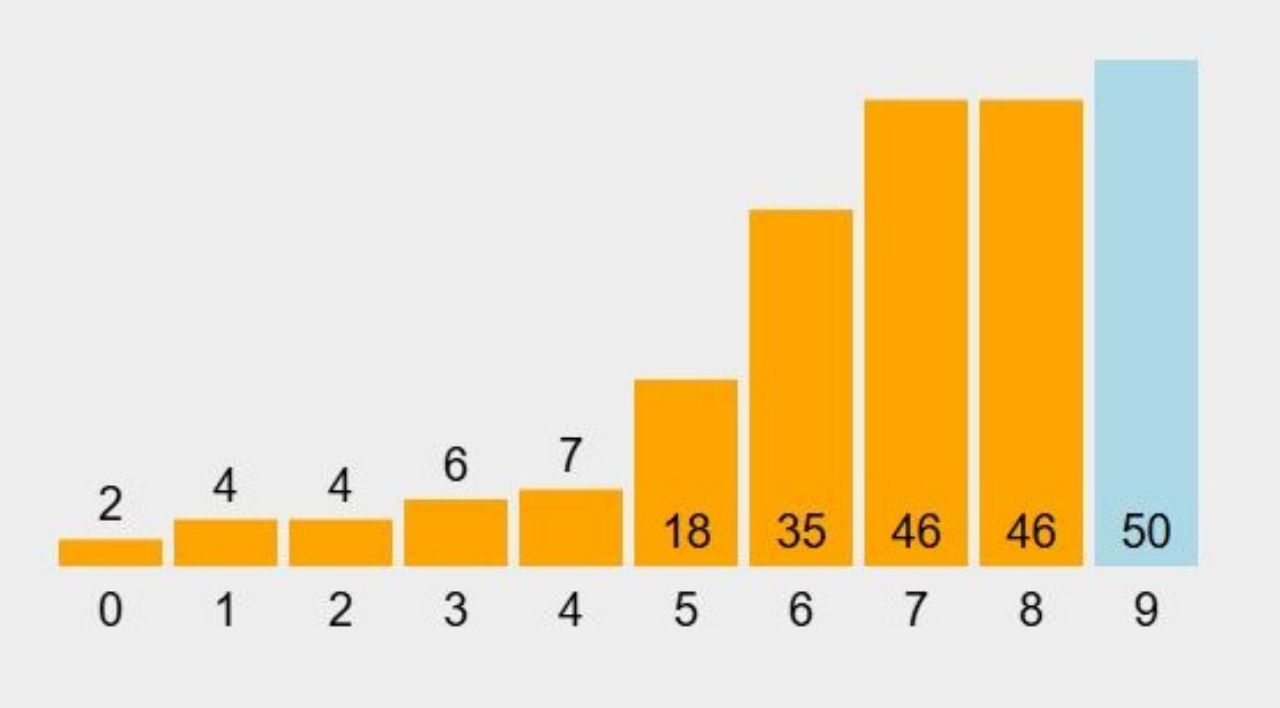
Método de Selección: Animación



Método de Selección: Animación



Método de Selección: Animación



Método de Selección: Traza

```
22, 10, 12, 10, 1, 5
```

Método de Selección: Traza

```
22, 10, 12, 10, 1, 5
```

```
22, 10, 12, 10, 1, 5
```

Método de Selección: Traza

```
22, 10, 12, 10, 1, 5
```

```
22, 10, 12, 10, 1, 5
```

```
1, 10, 12, 10, 22, 5
```

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 10, 12, 22, 10

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 10, 12, 22, 10

1, 5, 10, 12, 22, 10

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 10, 12, 22, 10

1, 5, 10, 12, 22, 10

1, 5, 10, 10, 22, 12

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 10, 12, 22, 10

1, 5, 10, 12, 22, 10

1, 5, 10, 10, 22, 12

1, 5, 10, 10, 22, 12

Método de Selección: Traza

22, 10, 12, 10, 1, 5

22, 10, 12, 10, 1, 5

1, 10, 12, 10, 22, 5

1, 10, 12, 10, 22, 5

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 12, 10, 22, 10

1, 5, 10, 12, 22, 10

1, 5, 10, 10, 22, 12

1, 5, 10, 10, 22, 12

1, 5, 10, 10, 12, 22

Método de selección

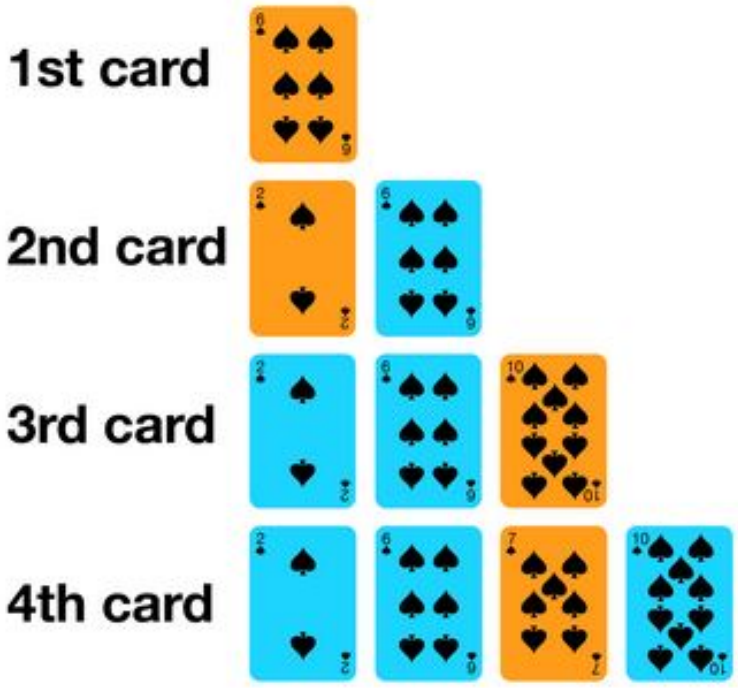
Implementación del método para ordenar una secuencia completa:

```
def ordenar_seleccion(A):
    """ lista -> None
        OBJ: Ordena ascendentemente una lista.
             Metodo de seleccion. Modifica la lista.  """
     for inicio_pasada in range(len(A)):
         # Encuentra el menor de la parte desordenada
         posicion_menor = inicio_pasada
         for i in range(inicio_pasada + 1, len(A)):
             if A[i] < A[posicion_menor]:
                 posicion_menor = i
         # Intercambia el primero con el menor
         A[inicio_pasada], A[posicion_menor] =
                     A[posicion_menor], A[inicio_pasada]
```

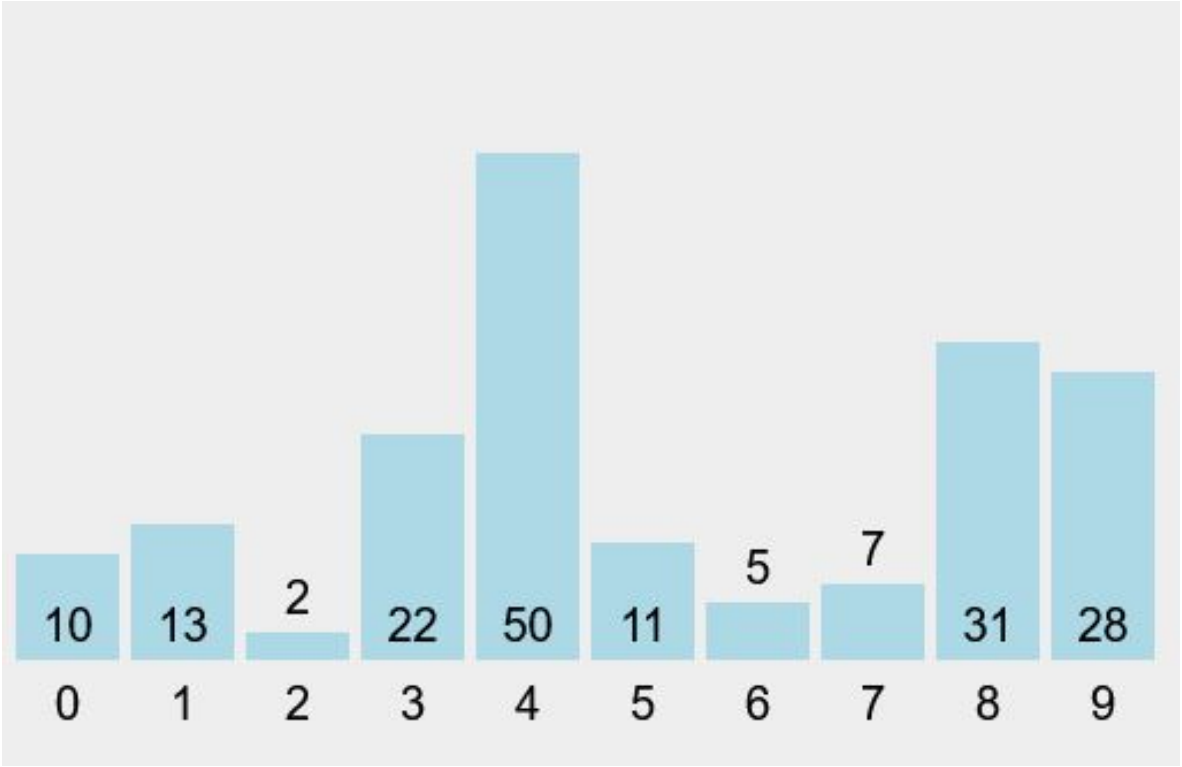
Método de inserción directa

- Cada elemento se desplaza hasta encontrar su lugar en la lista. Cuando todos han encontrado su lugar, la lista está ordenada.
- Para ello...
  - Se inserta en orden cada uno de los elementos del vector dentro de una subsecuencia ordenada.
  - Tras la inserción, la subsecuencia debe quedar ordenada de nuevo.
  - En la iteración i se inserta el elemento de la posición i + 1 de manera ordenada entre las posiciones 0 e i, creando una secuencia mayor

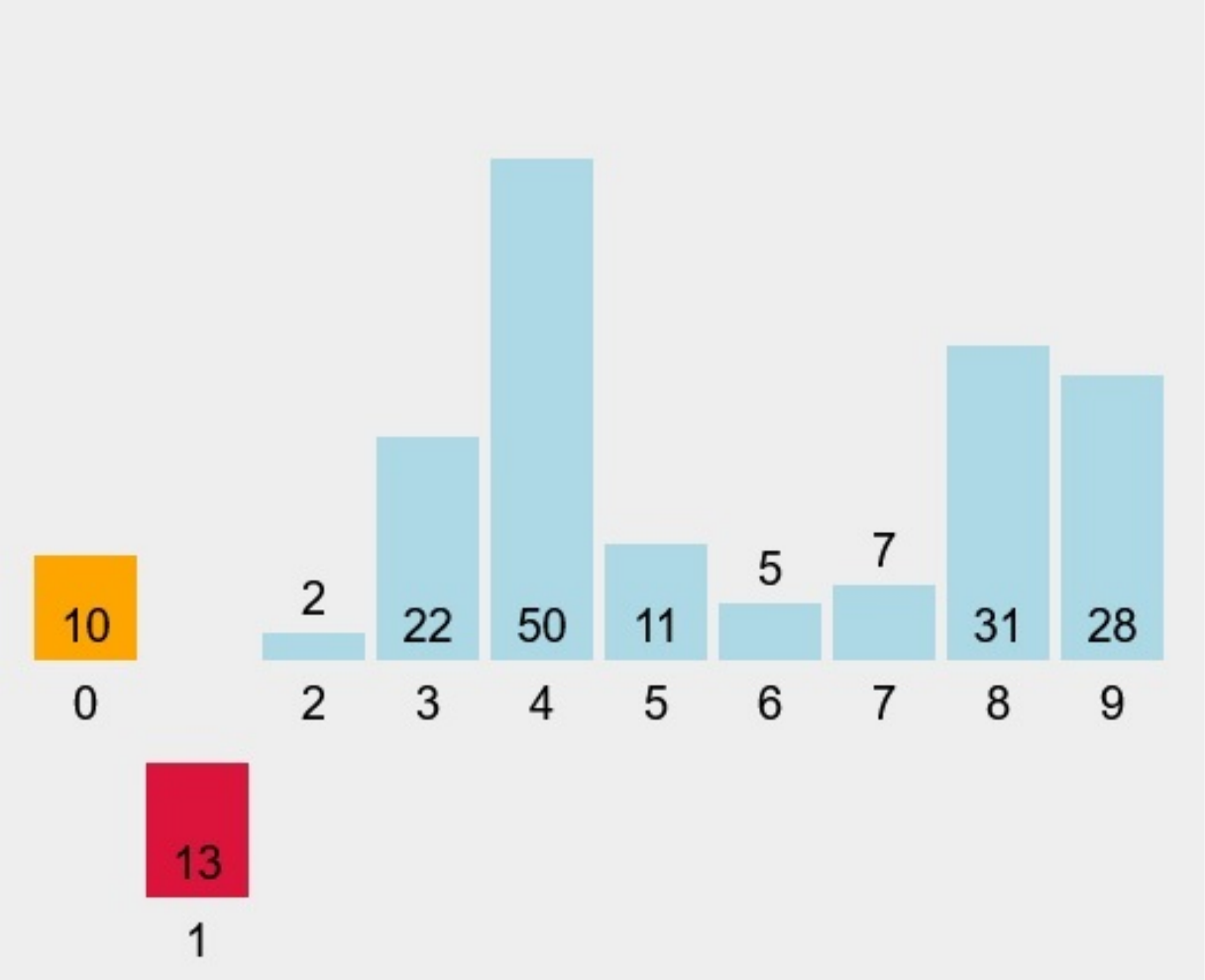
Método de Inserción directa: Animación



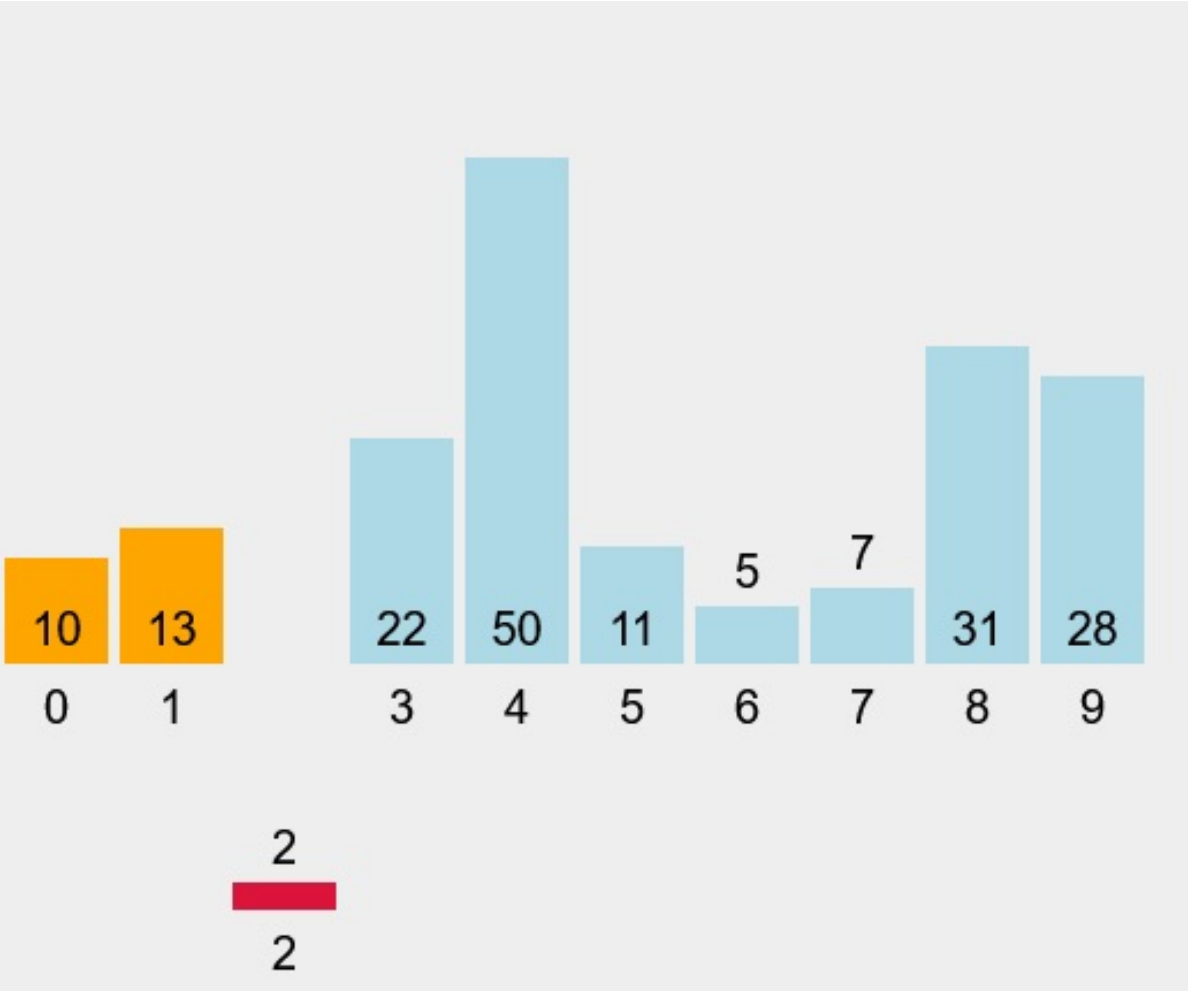
Método de Inserción directa: Animación



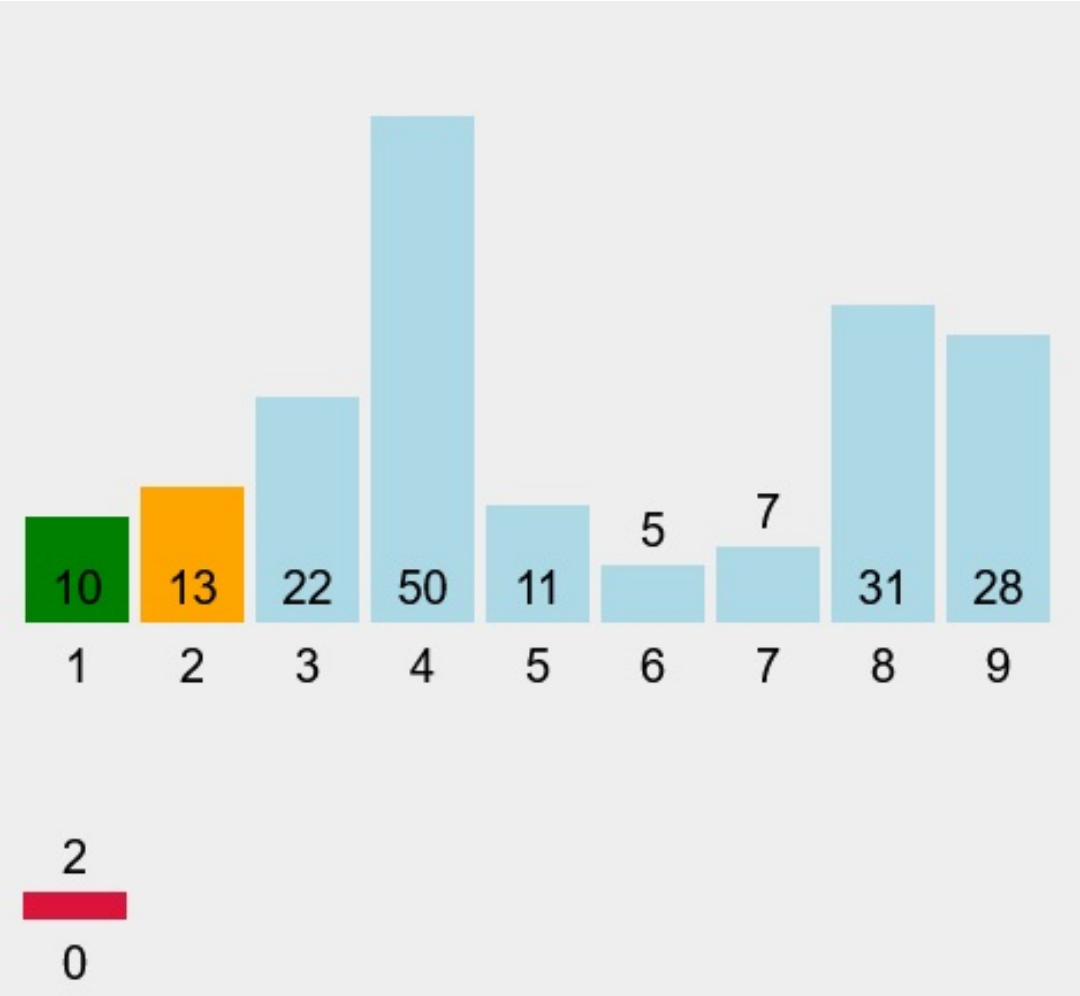
Método de Inserción directa: Animación



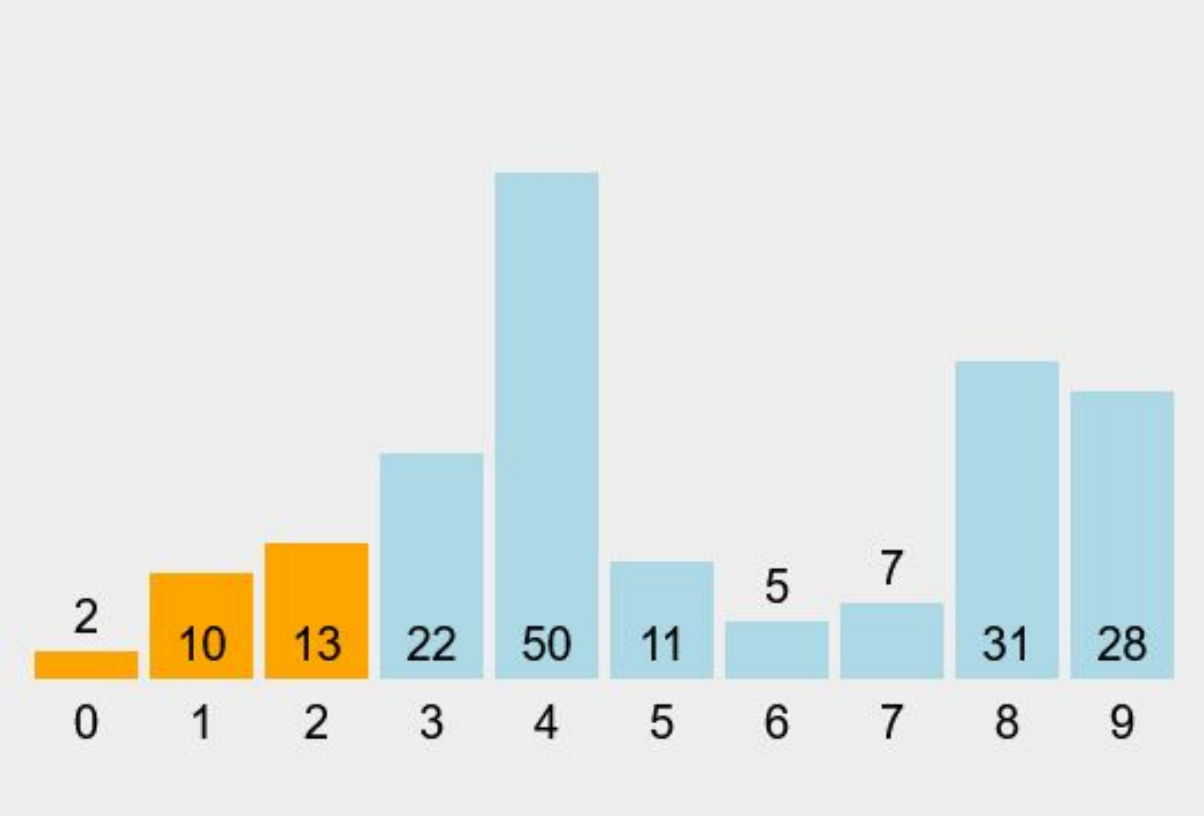
Método de Inserción directa: Animación



Método de Inserción directa: Animación



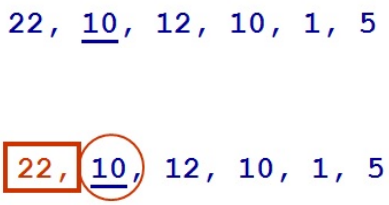
Método de Inserción directa: Animación



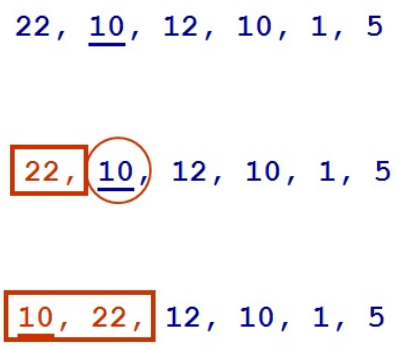
Método de Inserción directa: Traza

22, 10, 12, 10, 1, 5

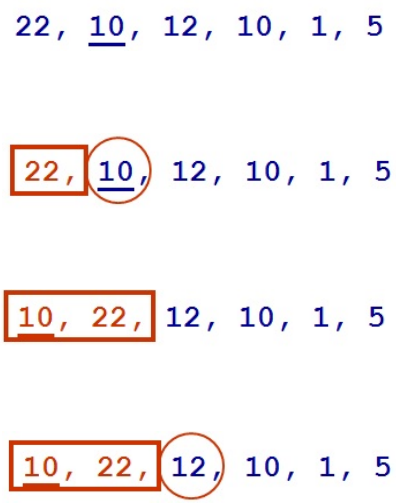
Método de Inserción directa: Traza



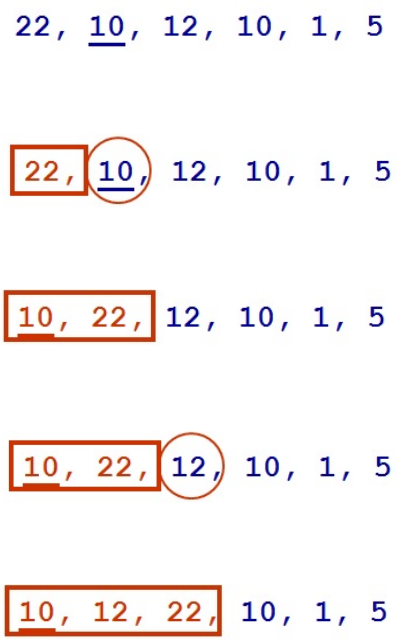
Método de Inserción directa: Traza



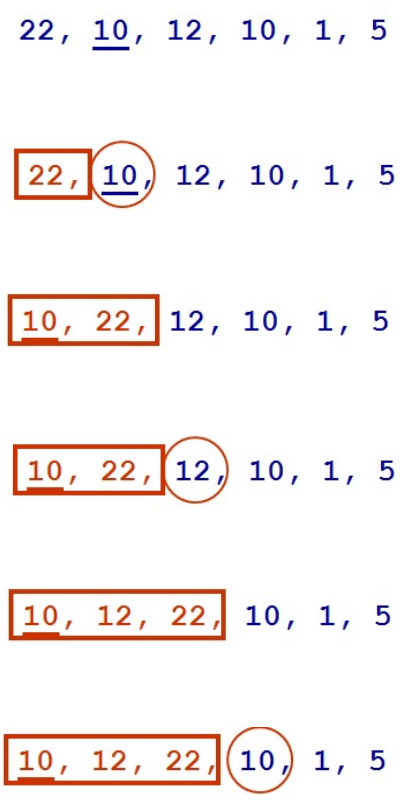
Método de Inserción directa: Traza



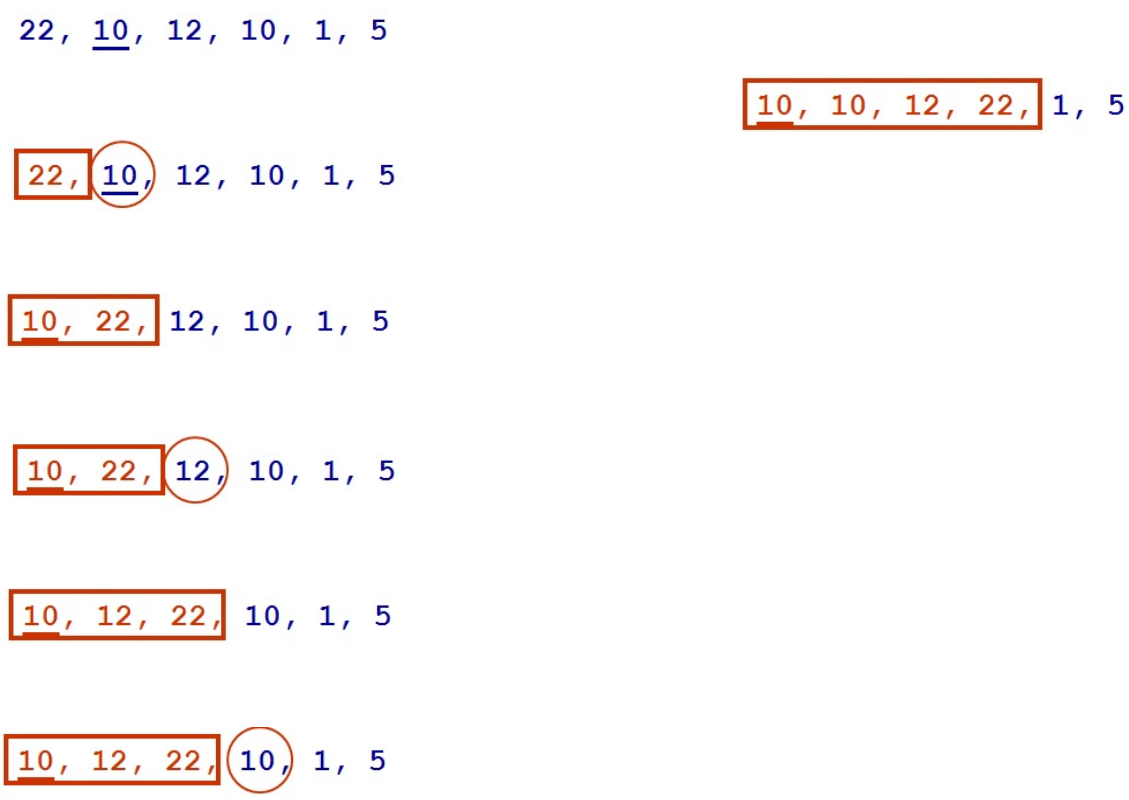
Método de Inserción directa: Traza



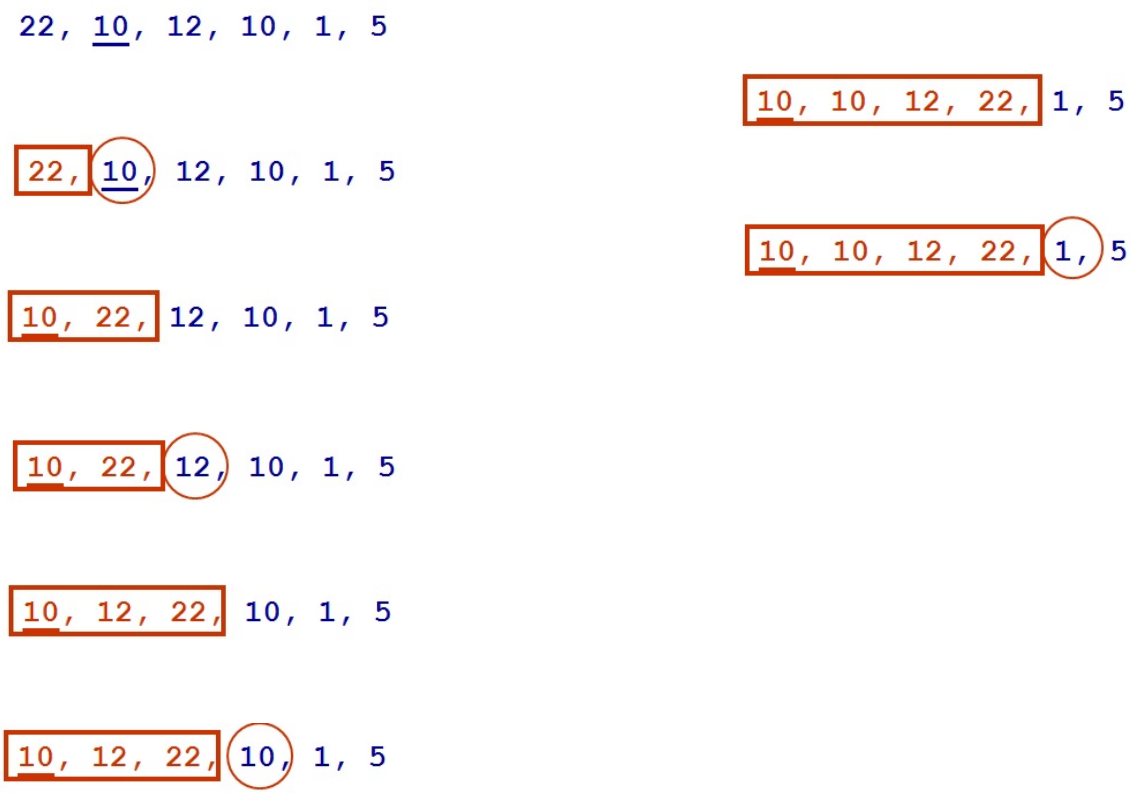
Método de Inserción directa: Traza



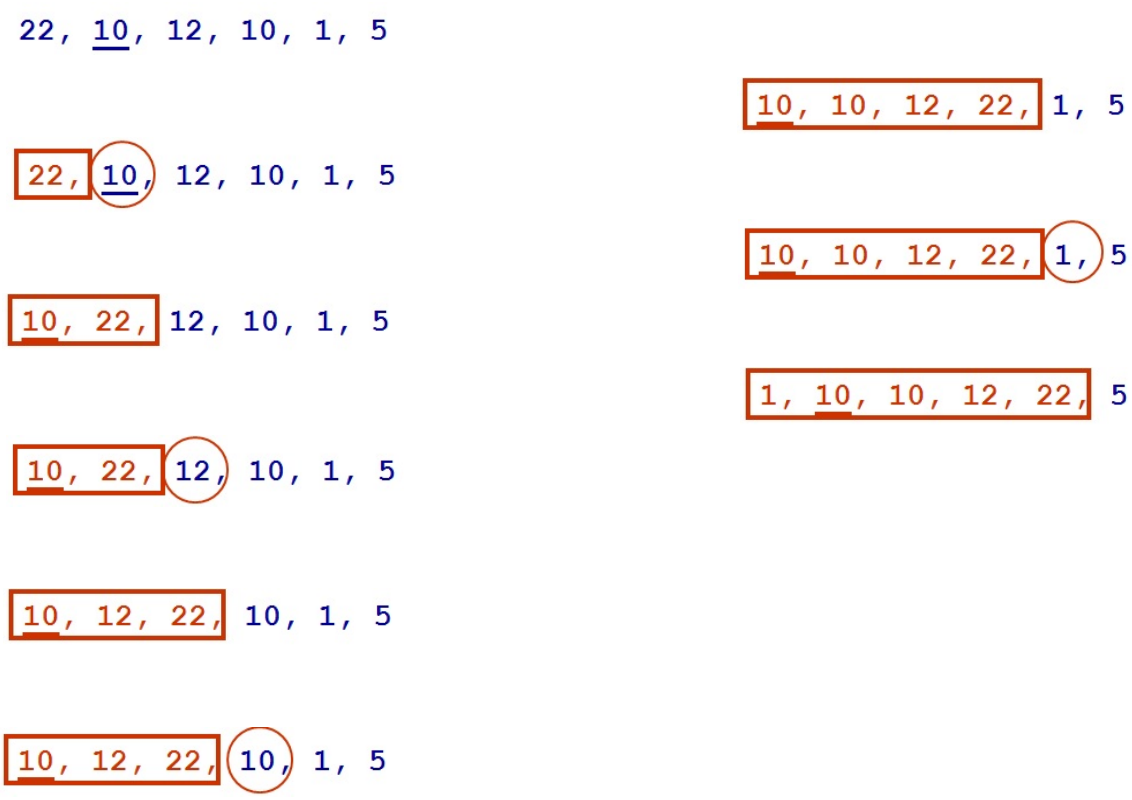
Método de Inserción directa: Traza



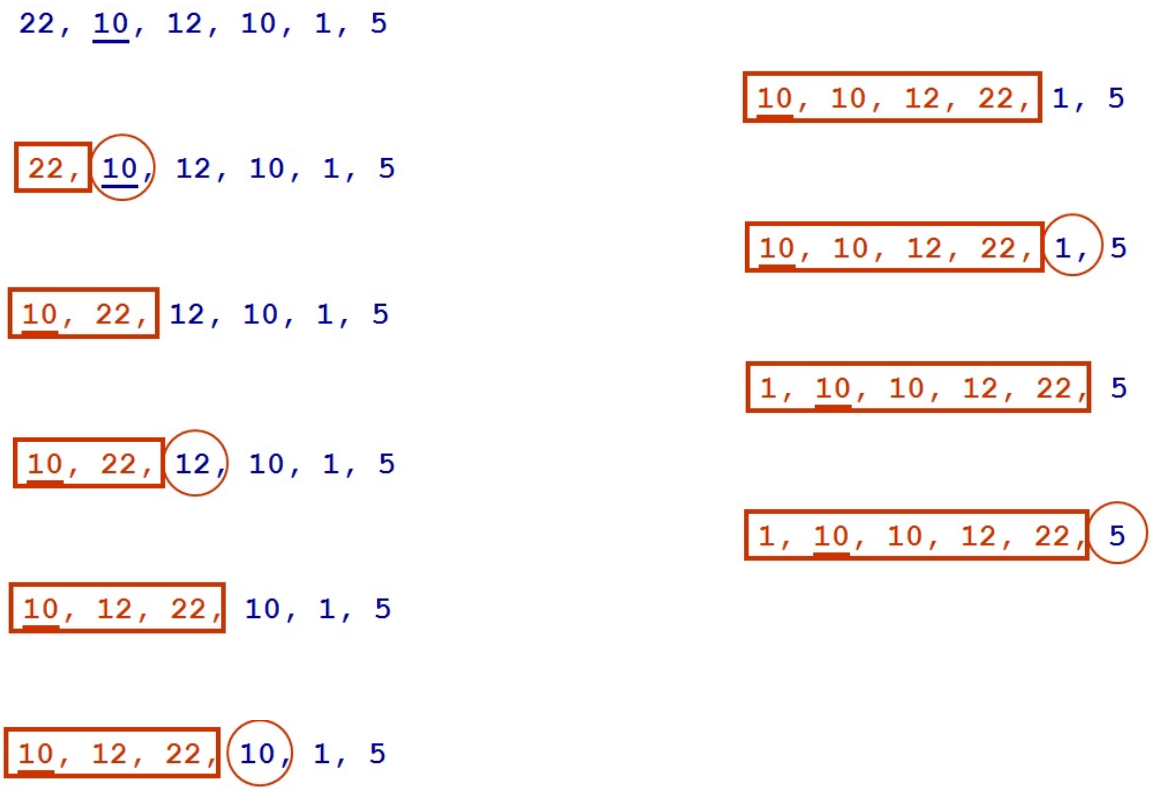
Método de Inserción directa: Traza



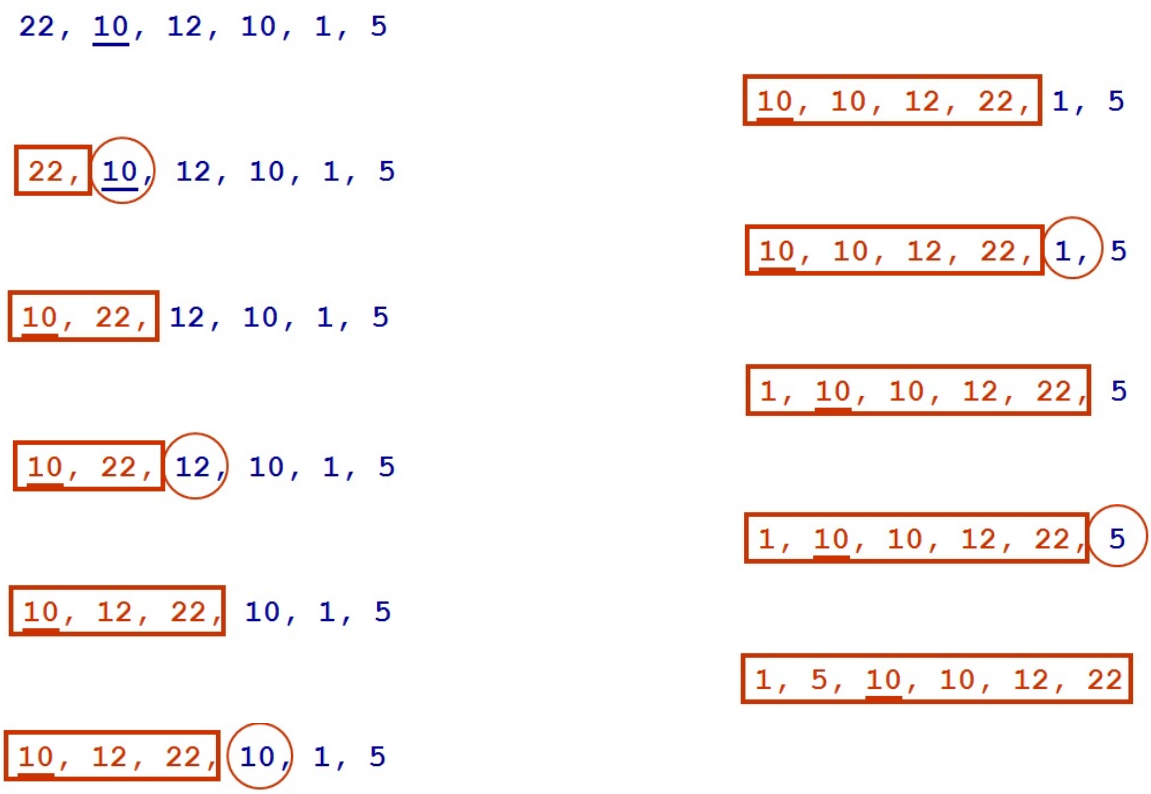
Método de Inserción directa: Traza



Método de Inserción directa: Traza



Método de Inserción directa: Traza



Método de Inserción directa

Implementación para ordenar una secuencia completa:

```
def insercion(A):
  """ lista -> None
      OBJ: Ordena ascendentemente una lista.
           Metodo de insercion. Modifica la lista.  """
  for i in range(1, len(A)):
      clave = A[i]
      # Mover elementos de la zona A[0..i-1], mayores que
      # la clave una posicion mas a la derecha
      j = i-1
      while j >=0 and clave < A[j] :
              A[j+1] = A[j]
              j -= 1
      # insertar el elemento 'clave' en la posicion que le
          corresponde
      A[j+1] = clave
```

Resumen

- Existen dos métodos para buscar elementos en una colección: búsqueda lineal y binaria.
- La búsqueda binaria es más rápida que la lineal, pero requiere que los elementos de la colección se encuentren previamente ordenados.
- Se puede ordenar una colección utilizando métodos básicos o avanzados de ordenación.
- Los métodos básicos son menos eficientes pero muy útiles cuando la colección es pequeña.
- Los métodos básicos son: Burbuja, Inserción directa y Selección.

Referencias

- Iteresantes animaciones que muestran la evolución de una lista de números a medida que se ordena en función de los distintos métodos `https://visualgo.net/en/sorting`

Tema V: Recursividad y algoritmos básicos

Grado en Inteligencia Artificial

Salvador Sánchez Alonso

Material docente en abierto de la Universidad Rey Juan Carlos

Noviembre de 2025



Programación I

DOMjudge:
¿Cómo subir problemas?

DOMjudge – Grado en Inteligencia Artificial

Isaac Lozano Osorio

Material docente en abierto de la Universidad Rey Juan Carlos

Universidad
Rey Juan Carlos

CC BY SA

Licencia

- **© 2025**

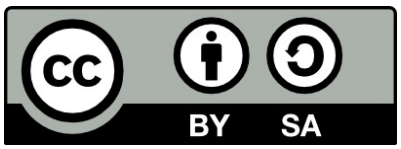
  Algunos derechos reservados

Isaac Lozano Osorio

isaac.lozano@urjc.es

Salvador Sánchez Alonso

salvador.sanchez@urjc.es



- *Este documento se distribuye bajo la licencia "**Atribución - Compartir Igual 4.0**" de Creative Commons*
- *Para más información, ver:* https://creativecommons.org/licenses/by-sa/4.0/

Contenido de la carpeta

Archivo o carpeta	Descripción
problem.yaml	Metadatos sobre el problema (por ejemplo, fuente, licencia, límites).
problem.pdf	Archivos del enunciado del problema.
submissions/	Soluciones correctas e incorrectas del juez para el problema.
data/sample/	Datos de prueba de ejemplo.
data/secret/	Datos de prueba secretos.

Contenido de la carpeta submissions

Carpeta	Descripción
accepted	Soluciones correctas.
wrong_answer	Soluciones con algún tipo de error.
time_limit_exceeded	Soluciones de complejidad alta.
run_time_error	Soluciones que tengan algún error.

No hacen falta, pero son recomendables.

Contenido de la carpeta sample

Carpeta	Descripción
X.in	Entrada por formato estándar.
X.ans	Salida esperada.
X.desc*	Es optativo, muestra la descripción del caso de prueba (sirve para responder dudas si el caso es muy grande).

Tanto la secret como sample tendrán el mismo contenido.
No debe repetirse el nombre del archivo (juntando ambas carpetas).
El X debe ser igual, es la forma de identificar quien es la entrada y salida del programa.

Consejos Varios

1. Pon nombres descriptivos a cada archivo X.zip únicos (si se repiten dará fallo al subir).
2. Sube soluciones de todos los tipos para comprobar el problema.
3. Una vez soluciones un fallo en un problema, hazlo tanto en el zip como en el juez.

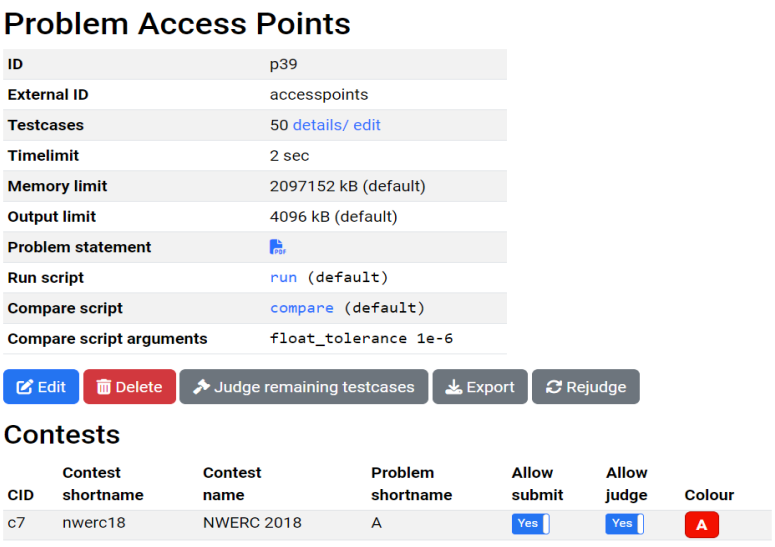
Importar al juez

Desde import/export encontramos la forma de subir el zip.



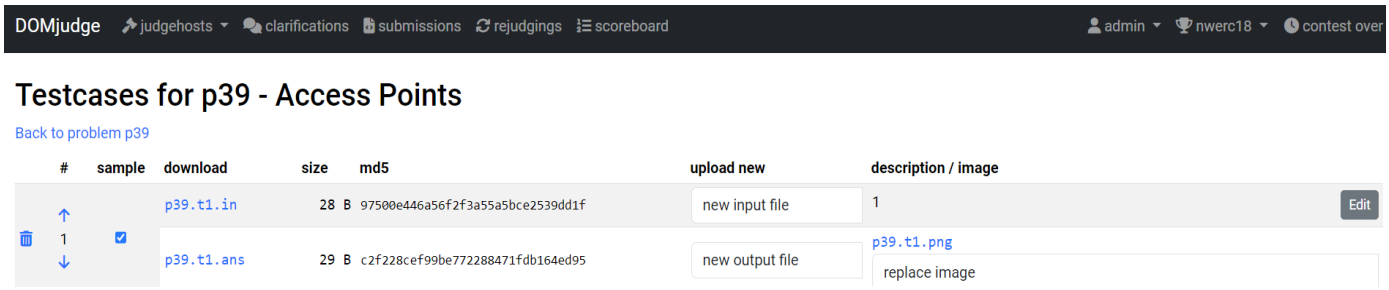
Modificar en el juez

Si pulsas en problems y en el propio problema, tienes un panel para cambiar los casos de prueba y enunciado.



Modificar en el juez

Los casos de prueba, se modifican uno por uno (eliminar, editar, etc). Si se marca el clic de sample, significa que el caso es público y el alumno puede verlo.



Programación I

DOMjudge:
¿Cómo subir problemas?

DOMjudge – Grado en Inteligencia Artificial

Isaac Lozano Osorio

Material docente en abierto de la Universidad Rey Juan Carlos

Universidad
Rey Juan Carlos

CC BY SA

Programación I

Prácticas:

Escritorio de desarrollo

GRADO EN INTELIGENCIA ARTIFICIAL

Isaac Lozano Osorio

Material docente en abierto de la Universidad Rey Juan Carlos

Universidad
Rey Juan Carlos

CC BY SA

Licencia

- **© 2025**

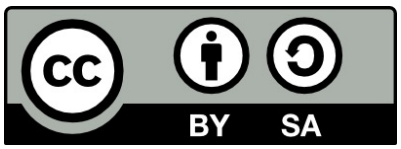
  Algunos derechos reservados

Isaac Lozano Osorio

isaac.lozano@urjc.es

Salvador Sánchez Alonso

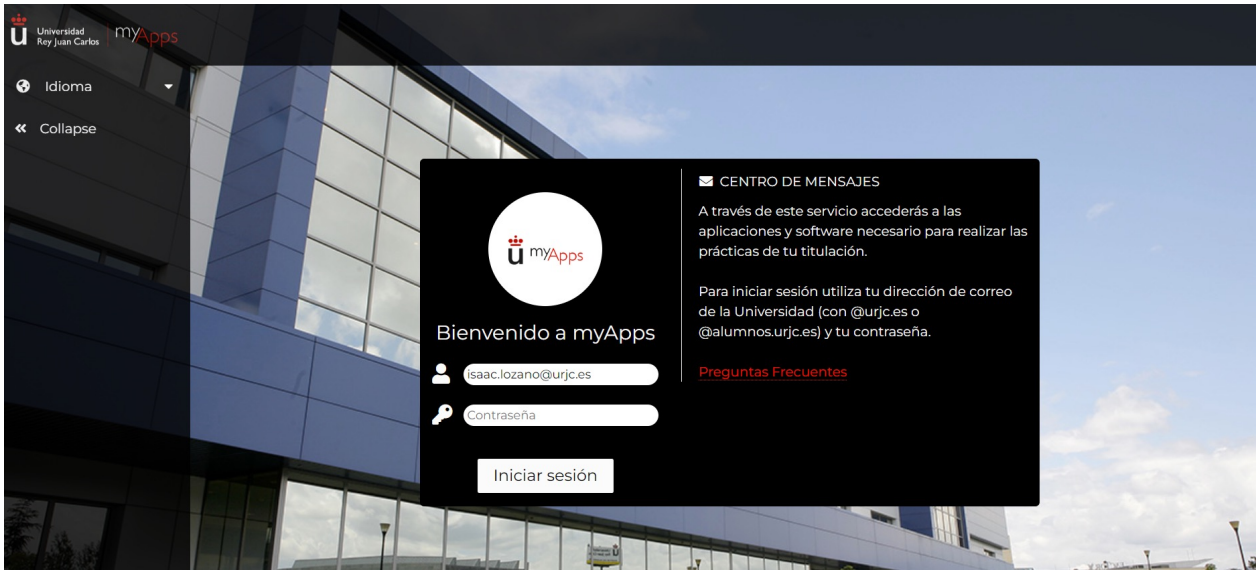
salvador.sanchez@urjc.es



- *Este documento se distribuye bajo la licencia "**Atribución - Compartir Igual 4.0**" de Creative Commons*
- *Para más información, ver:* https://creativecommons.org/licenses/by-sa/4.0/

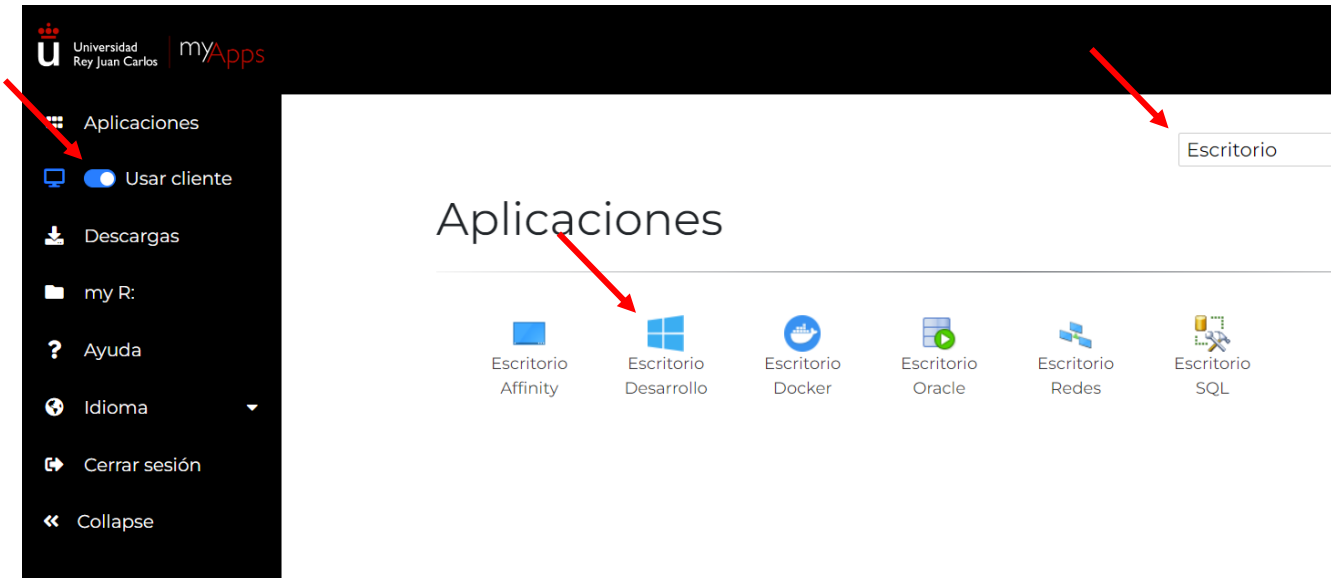
Acceso al escritorio de desarrollo

https://myapps.urjc.es/myapps/

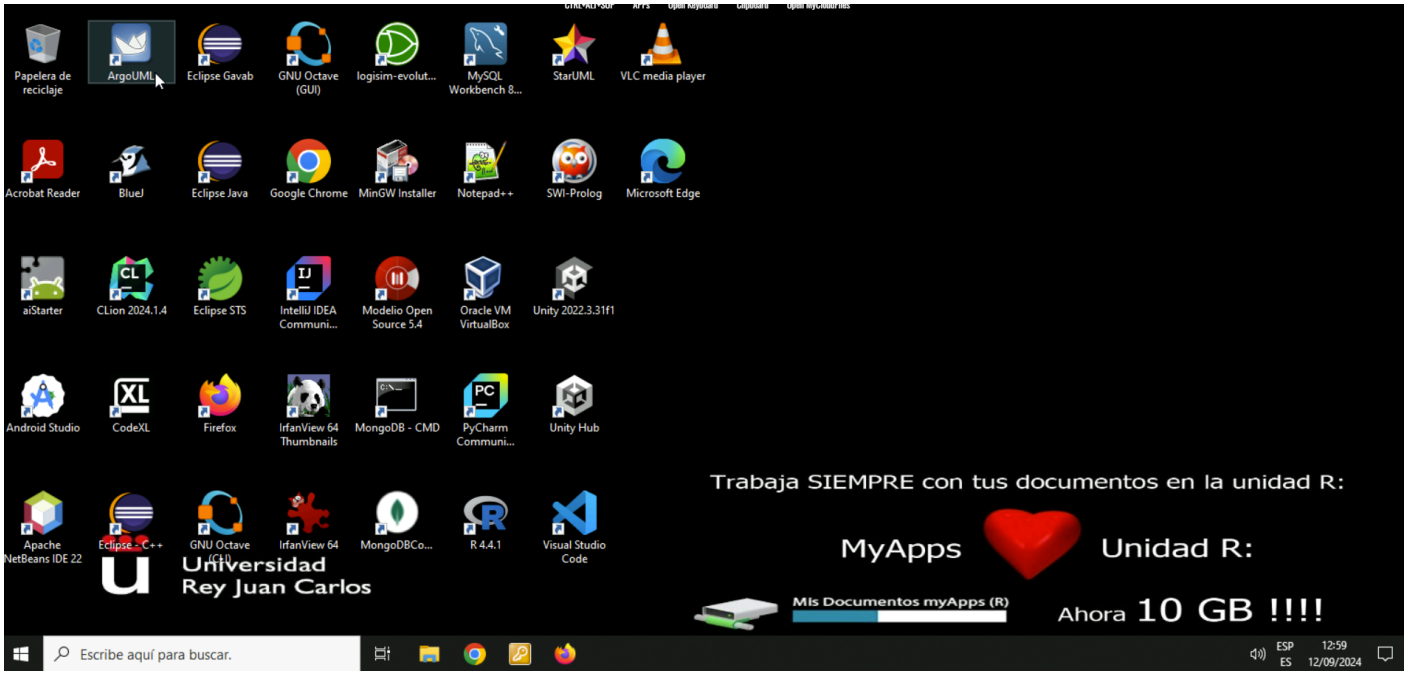


Acceso al escritorio de desarrollo

Debe estar **desactivado** usar cliente.

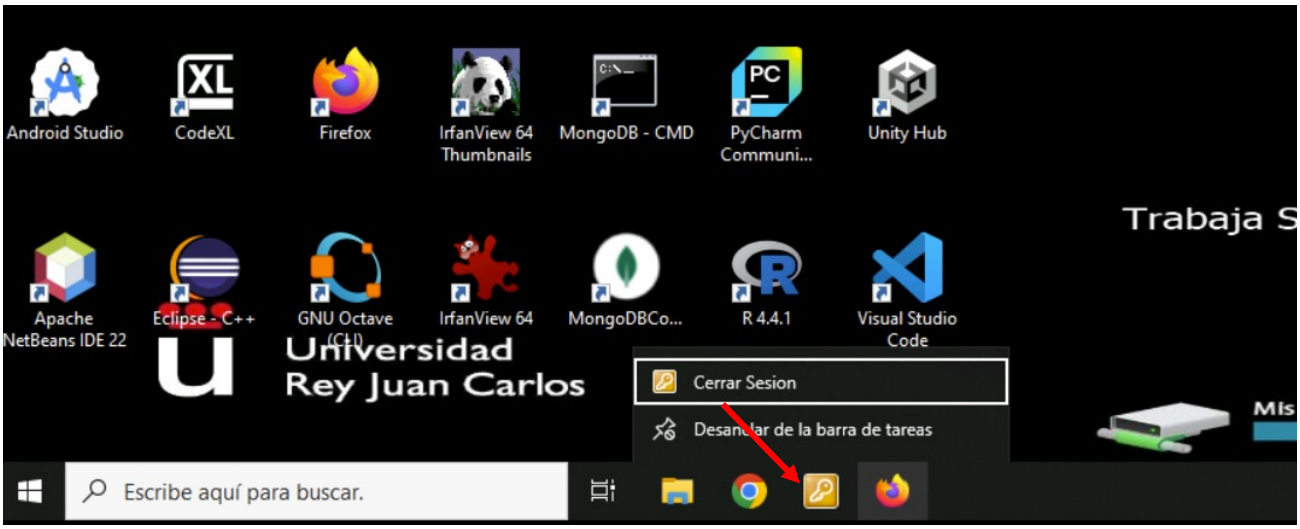


Escritorio de desarrollo personal



Escritorio de desarrollo personal

Al entrar, pulsad **click derecho en la llave** (cerrar sesión) y pulsar en **desanclar** de la barra de tareas, en otro caso podéis cerrar sesión sin querer y perder algunos datos.

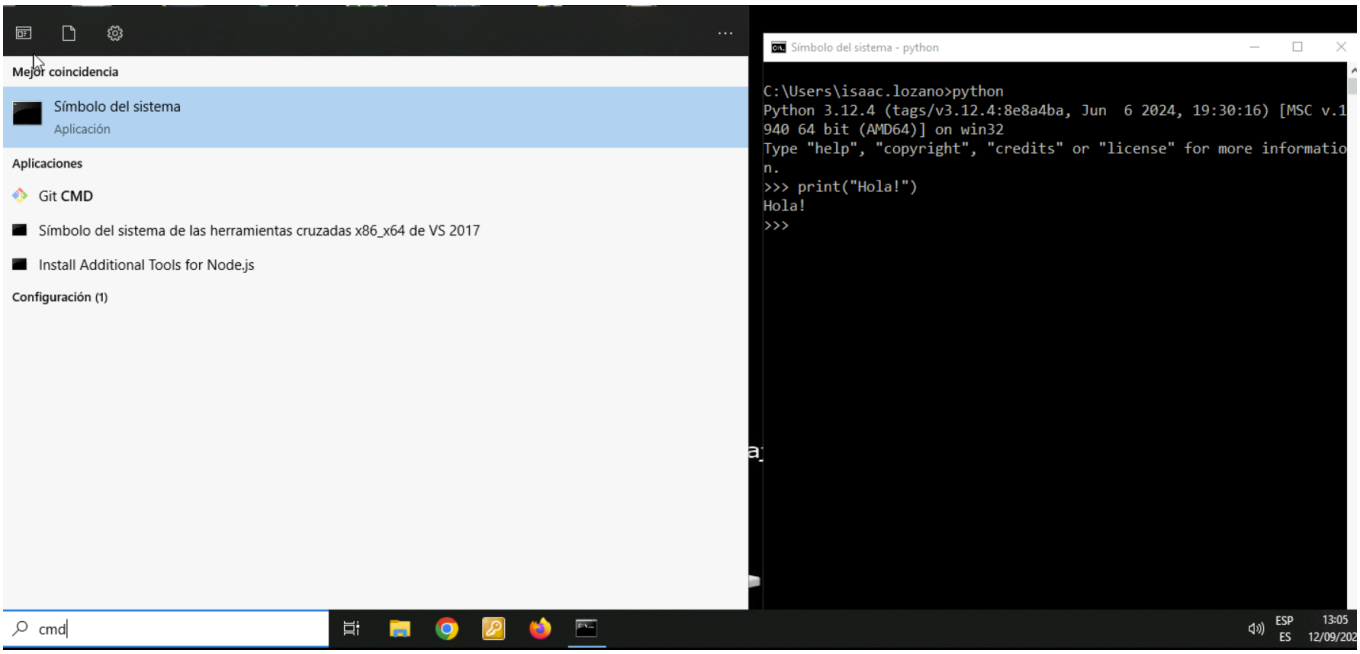


Escritorio de desarrollo personal

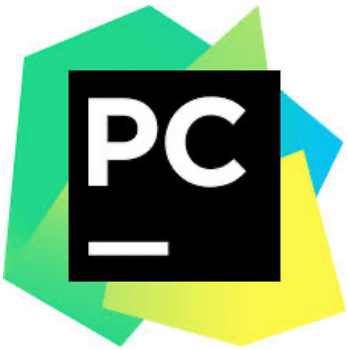
Importante:

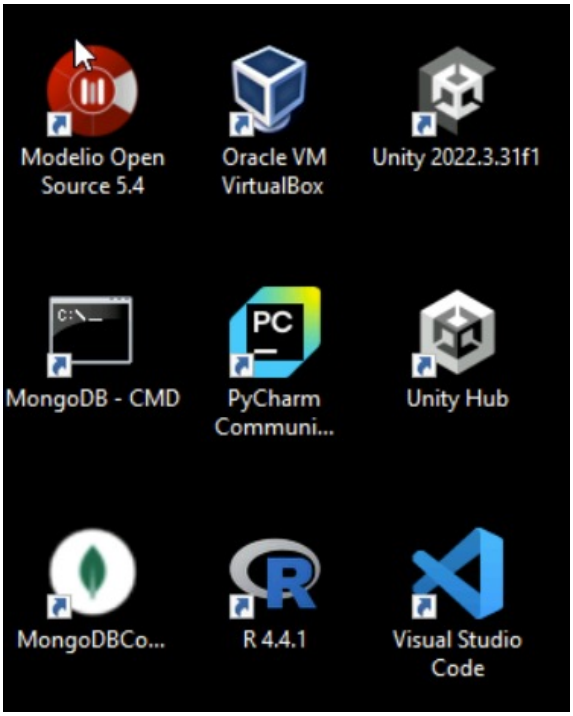
- Guardad en la **unidad R**, en otro caso se puede perder la información (proyectos, códigos).
- Las sesiones son en **máquinas virtuales**, si entramos todos a la vez **puede tardar**.
- Al ser sesiones virtuales, **se cierran cuando pasa un determinado tiempo**, pero si entras rápido mantienes la misma sesión (en exámenes se os cerrará y tendréis que iniciar de nuevo).
- Entrando **desde cualquier ordenador** de la universidad o fuera de la universidad, **dispondrás de tu perfil**.

Python desde la CMD

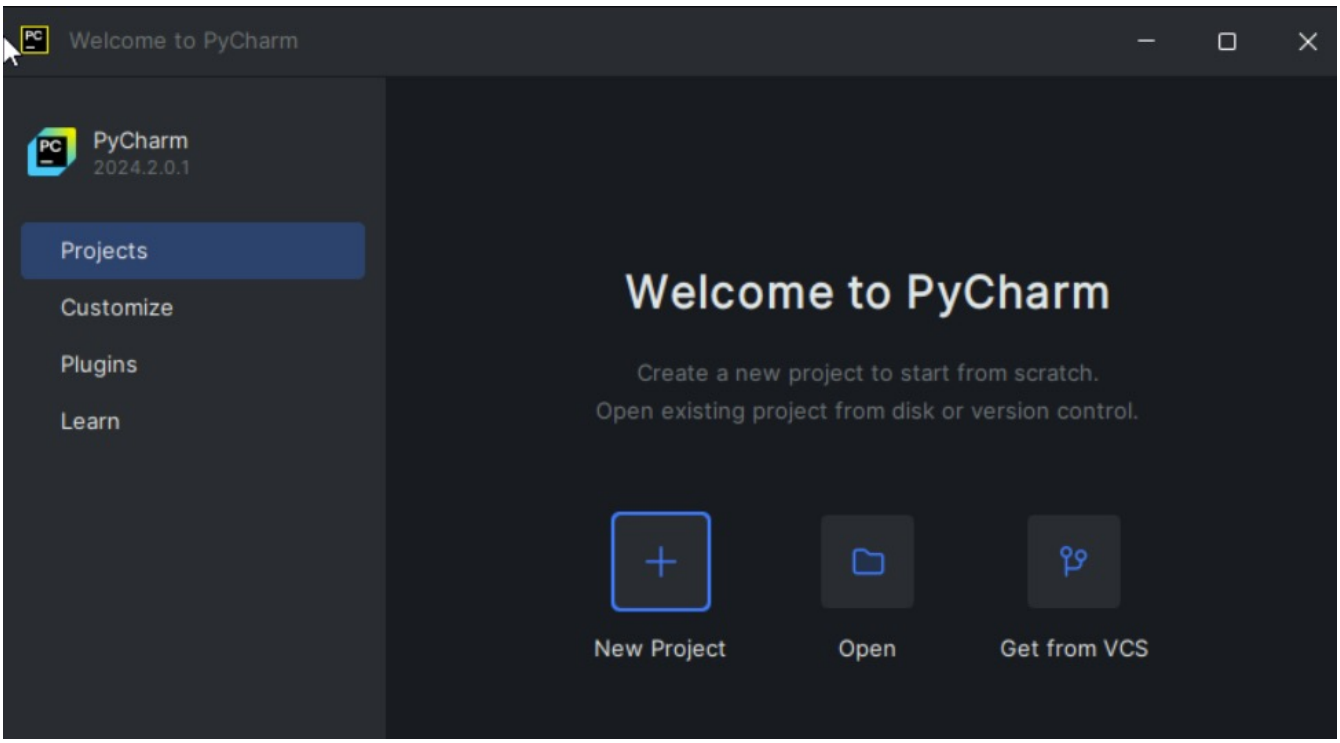


Creación de un proyecto con PyCharm

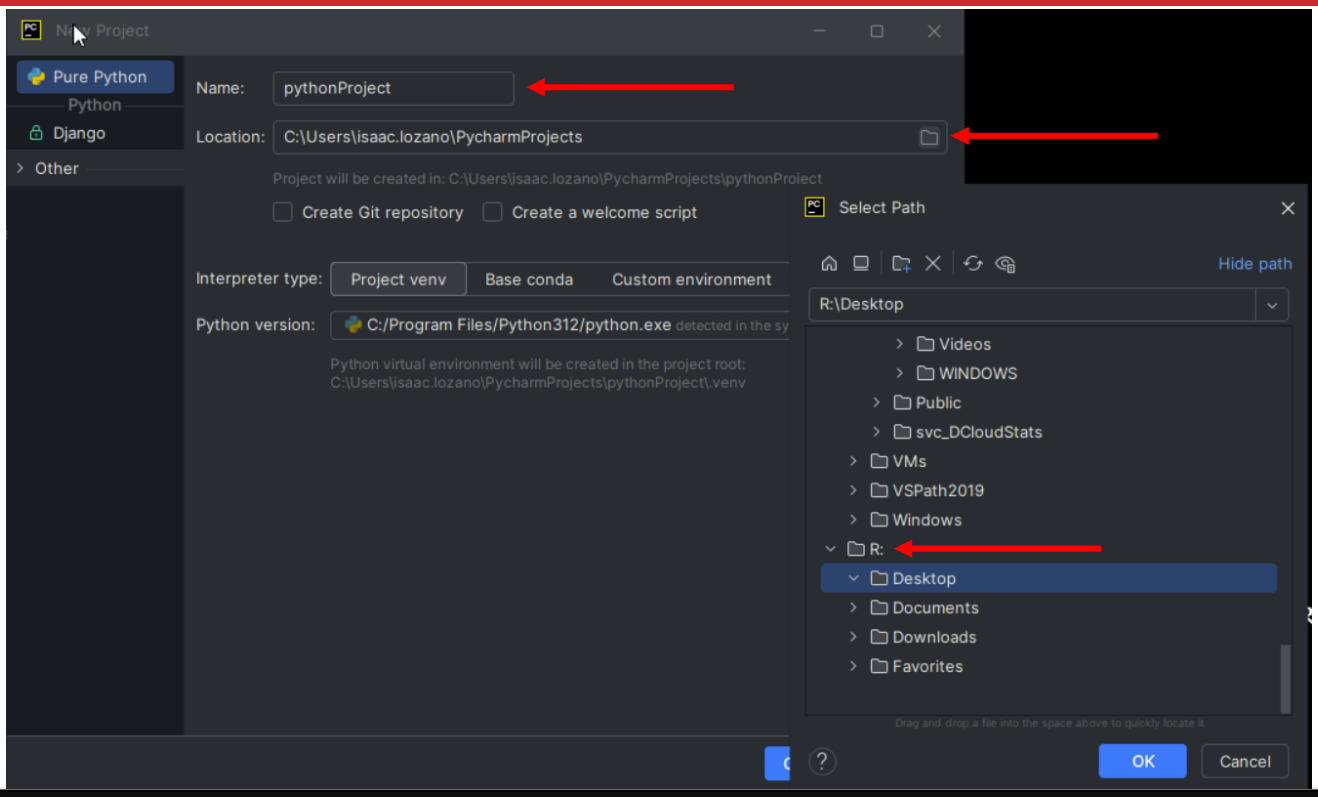




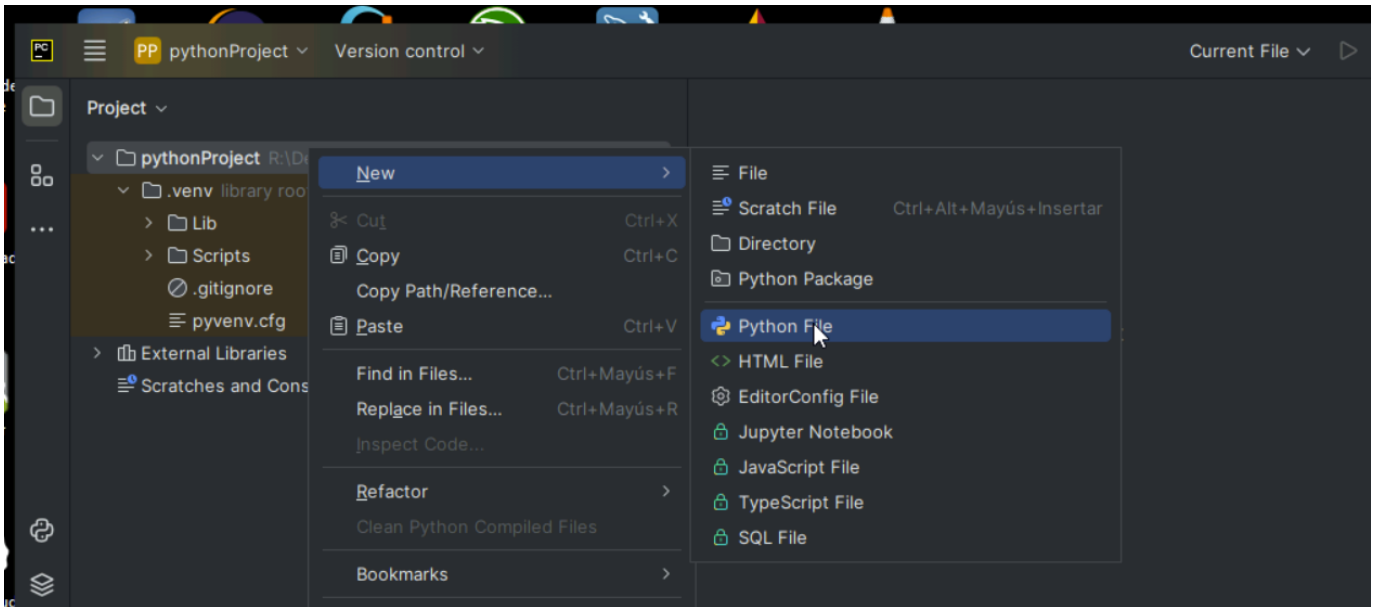
Creación de un proyecto con PyCharm



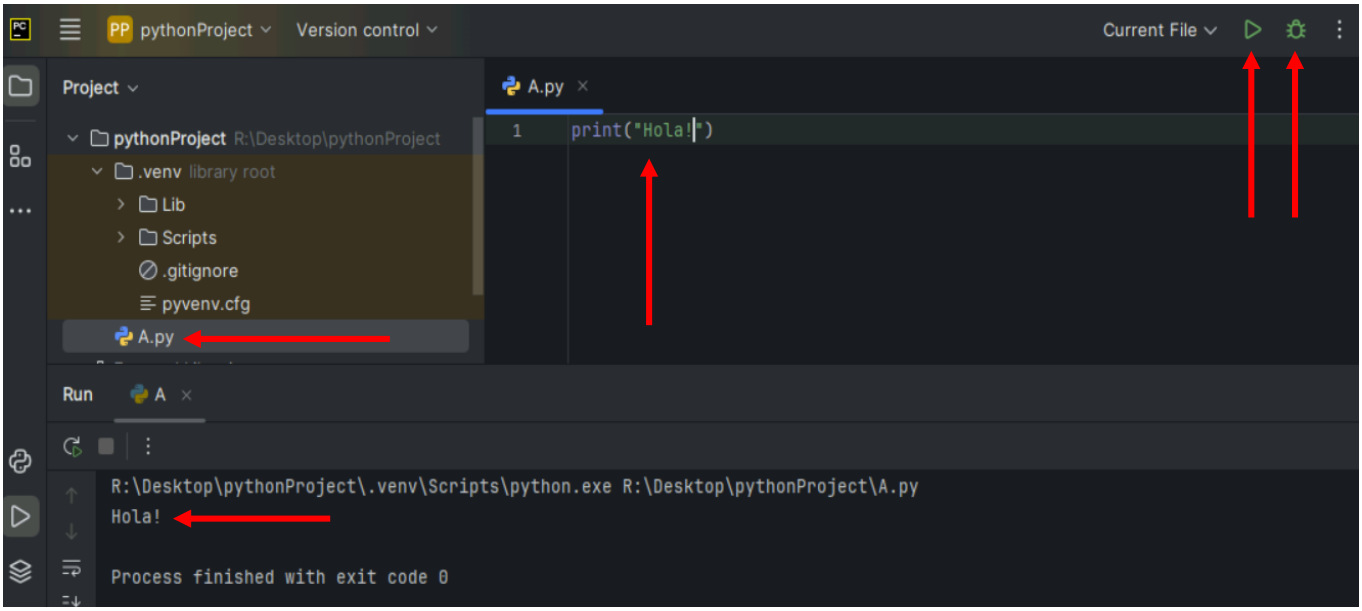
Creación de un proyecto con PyCharm



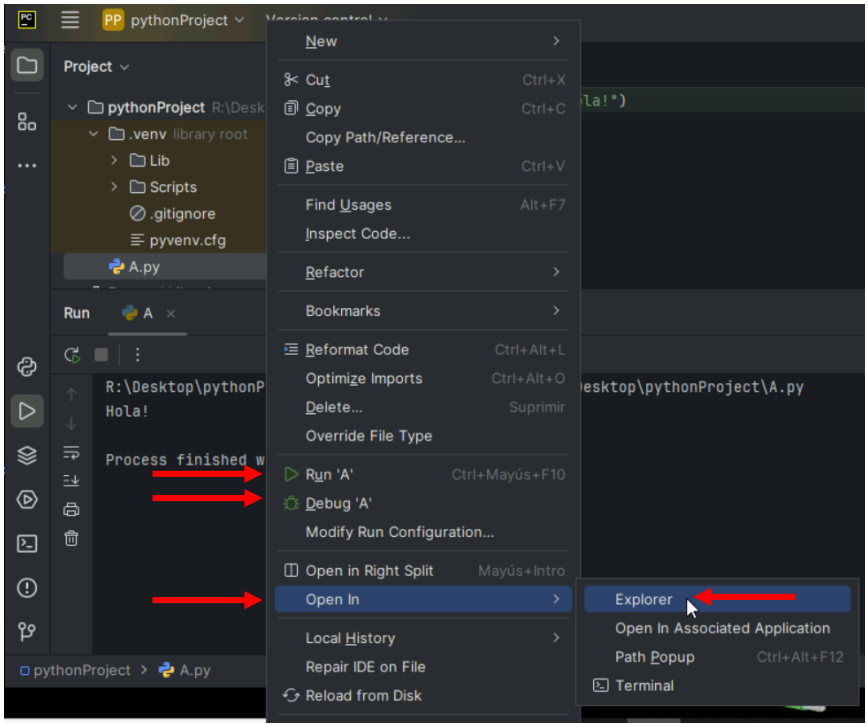
Creación de un script de Python



Ejecución


pythonProject
Version control
Current File
Project
A.py
pythonProject R:\Desktop\pythonProject
.venv library root
Lib
Scripts
.gitignore
pyvenv.cfg
A.py
1
print("Hola!")
Run
A
R:\Desktop\pythonProject\.venv\Scripts\python.exe R:\Desktop\pythonProject\A.py
Hola!
Process finished with exit code 0

Ejecución


pythonProject
Project
pythonProject R:\Desk
.venv library root
Lib
Scripts
.gitignore
pyvenv.cfg
A.py
Run
A
R:\Desktop\pythonP
Hola!
Process finished w
pythonProject > A.py
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Inspect Code...
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Suprimir
Override File Type
Run 'A' Ctrl+Mayús+F10
Debug 'A'
Modify Run Configuration...
Open in Right Split Mayús+Intro
Open In
Local History
Repair IDE on File
Reload from Disk
Explorer
Open In Associated Application
Path Popup Ctrl+Alt+F12
Terminal

¿Cómo instalar en vuestro ordenador?



JetBrains
Toolbox App

Manage your IDEs the easy way

https://www.jetbrains.com/toolbox-app/



https://www.python.org/downloads/

¿Cómo instalar en vuestro ordenador?



https://chocolatey.org/install

Install | Upgrade | Uninstall

To install Python, run the following command from the command line or from PowerShell:

```
> choco install python --pre
```

Downloads:
26,037,354

Downloads of v 3.13.0-rc2:
304

Last Update:
07 Sep 2024

Package Maintainer(s):
chocolatey-community

Software Author(s):
Python Software Foundation

Programación I

Prácticas:

Escritorio de desarrollo

GRADO EN INTELIGENCIA ARTIFICIAL

Isaac Lozano Osorio



Universidad
Rey Juan Carlos

Programación I

Ejercicios, problemas y pruebas de evaluación

GRADO EN INTELIGENCIA ARTIFICIAL

Salvador Sánchez Alonso, Isaac Lozano Osorio

Septiembre 2025

Material docente en abierto de la Universidad Rey Juan Carlos



Universidad
Rey Juan Carlos



Licencia

©2025

Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso
(salvador.sanchez@urjc.es)

Este documento se distribuye bajo la licencia Compartir Igual 4.0 (CC BY-SA 4.0)



Contenidos

- Cuadernos de trabajo:
  - Cuaderno 1: Introducción
  - Cuaderno 2: Modularidad
  - Cuaderno 3a: Estructuras selectivas
  - Cuaderno 3b: Estructuras iterativas
  - Cuaderno 4: Estructuras de datos
  - Cuaderno 5: Estructuras de datos avanzadas
  - Cuaderno 6: Recursividad
  - Cuaderno 7: Algoritmos
- Pruebas de evaluación
  - PE1 – curso 2024/2025 (soluciones)
  - PE2 – curso 2024/2025 (soluciones)
- Examen Junio 2025

CC BY SA

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 1

1. Escribe en el intérprete las constantes `20`, `34.5` y `True` y comprueba sus tipos.

   SOLUCION:

```
>>> type(20)
<class 'int'>
>>> type(34.5)
<class 'float'>
>>> type(True)
<class 'bool'>
```

2. Crea una variable de texto de una línea y muéstrala en pantalla. Haz lo mismo con una variable en dos líneas.

   SOLUCION:

```
>>> nombre = 'Antonio'
>>> print (nombre)
Antonio
>>> nombre_completo = 'Antonio Sin-Hernandez \
Rodriguez de Lara'
>>> print(nombre_completo)
Antonio Sin-Hernandez Rodriguez de Lara
```

3. Escribe un código que, mediante el uso de una variable de tipo lógico (bool), determine si una empresa es micro, según los estándares de la UE. Dichos estándares establecen que toda empresa que cuenta con menos de 10 empleados y una facturación por debajo de los 2 millones de euros o bien con un balance inferior a los 2 millones de euros, es una microempresa. Haz los cálculos que permiten saber si una empresa de 20 empleados, 18 millones de euros de facturación y 5 millones de euros de balance es una micro empresa y almacena el valor en una variable lógica.

   SOLUCION PROPUESTA:

```
MIN_EMPLEADOS = 10
MIN_FACTURACION = 2 # MEuros
MIN_BALANCE = 2 # MEuros

empleados = 20
facturacion = 18
balance = 5

es_microempresa =  empleados < MIN_EMPLEADOS and     \
                   facturacion < MIN_FACTURACION or \
                   balance < MIN_BALANCE
print(es_microempresa)
          #por comprobar que es una bool como pide el enunciado
print(type(es_microempresa))
```

4. Modifica el código anterior para que calcule también si se trata de una pequeña empresa, siendo la definición de pequeña empresa toda aquella que no es microempresa y que cumple bien que su balance es menor a 10MEUR o bien que con menos de 50 empleados factura un máximo de 10MEUR.

   SOLUCION PROPUESTA: Hay que añadir la siguiente línea al programa:

```
es_pequenia = not es_microempresa and
              (balance <= 10 or empleados <50 and facturacion <=10)

print(es_pequenia)  # solo necesario para probar lo anterior
print(type(es_pequenia)) # solo necesario para probar lo anterior
```

5. Comprueba el tipo y el valor de las siguientes expresiones: 10>2, 3+2, 3+2.0, "25", 'Hola'.

   SOLUCION:

```
>>>type(10>2)
<class 'bool'>
>>> type(3+2)
<class 'int'>
>>> type(3+2.0)
<class 'float'>
>>> type("25")
<class 'str'>
>>> type('Hola')
<class 'str'>



>>> 10>2
True
>>> 3+2
5
>>> 3+2.0
5.0

>>> "25"
'25'
>>> 'Hola'
'Hola'
>>>
```

Ejercicios resueltos

1. Encuentra tres valores que tengan los tipos `int`, `str` y `float`. Comprueba con Python que sus valores son de los tipos pedidos.
2. Escribe un código que imprima tu nombre y edad, así: ¨Nombre: Pedro Martín, edad: 25.

Para los siguientes ejercicios, elige el nombre adecuado para las variables y constantes, así como el tipo de dato a emplear para gestionar la magnitud que dicho dato representa en el mundo real. Evita hacer cambios de tipo implícito y presta atención a la prelación de operadores de cada tipo.

3. Calcula y muestra por pantalla el área de un cuadrado:
    a) Utilizando dos variables, una para almacenar la longitud del lado y otra para almacenar el área.
    b) Utilizando una única variable para almacenar la longitud del lado.
4. Escribe un programa en Python que calcule el impuesto que debe pagar un contribuyente a partir de sus ingresos anuales y el número de hijos. El impuesto a pagar es un tercio del ingreso imponible, siendo este último igual a los ingresos totales menos una deducción personal de 600€ y otra deducción de 60€ por hijo.
5. Escribe un programa que dada una hora (expresada en hora, minutos y segundos) muestre por pantalla el total de segundos transcurridos desde la última medianoche y los que quedan para la siguiente medianoche.
6. Una industria mantiene una flota de camiones para repartir productos. En cada viaje, el conductor anota la distancia recorrida en kilómetros, los litros de gasoil utilizados, el coste del litro de gasoil y los demás costes de mantenimiento del camión (agrupados). Como parte del proceso de contabilidad, el controlador necesita calcular, para cada camión y para cada viaje, los kilómetros recorridos por litro, el coste total del viaje y el coste total por kilómetro (incluidos los gastos de mantenimiento). Diseña un programa sencillo que lleve a cabo estos cálculos para un camión en un viaje.
7. Python permite convertir elementos de un tipo en otro. Lleva a cabo las siguientes conversiones y comenta los resultados:

```
texto_numerico = "45"
int (texto_numerico)
int ("Hola")
int (3.99999)
int(-3.99999)
float (34)
int("diez")
```

8. Escribe un programa que, a partir de los lados de un rectángulo, calcule su área y perímetro y los muestre por pantalla.

9. Por consideraciones históricas, un programador suele interpretar que los identificadores i, j, k, l, m, n corresponden a datos enteros, mientras que a, b, c, x, y, z son identificadores que suelen asociarse con valores reales. Escribe un programa que, a partir de 3 números reales que inicializarás, calcule su media, suma total y producto total y muestre todos estos datos por pantalla.

10. La temperatura expresada en grados centígrados TC, se puede convertir a grados Fahrenheit (TF) mediante la siguiente fórmula:

    TF = 9.0*TC/5.0 + 32.0

    Escribe un programa que utilizando un cierto valor de temperatura en grados Fahrenheit, muestre en pantalla la temperatura equivalente en grados Centígrados.

11. Transcribe el siguiente programa, ejecútalo y comenta los resultados:

```
a = 3/2
b = 3.0 / 2
c = 3 // 2
print ('a= ', a, 'b= ', b, 'c= ', c)
```

12. El salario base de un vendedor es de 2.000 euros mensuales. A este salario se le suma un 3% de comisión sobre el total de las ventas que ha realizado, pero al total obtenido hay que descontarle un 32% del IRPF. Escribe un programa que, a partir del importe de las ventas que ha realizado el vendedor durante el último mes y escriba el salario neto que cobrará ese mes.

13. Se desea conocer el importe en Libras Esterlinas (GBP) al cambio de una cantidad en Euros (EUR). Escribe un programa que, a partir de una cierta cantidad en euros y del tipo de cambio del día, muestre el equivalente en libras teniendo en cuenta que la casa de cambio retiene una comisión del 2% sobre el total de la operación.

14. En Python es posible hacer operaciones con variables de tipo str. Vamos a probarlo tecleando:

```
fruta = " ciruela "
tipo = " claudia "
print (fruta + tipo)
```

    a) ¿Qué se obtiene? ¿Qué es lo que hace la operación + con las cadenas de texto?

    Teclea ahora:

```
print fruta * 3
```

    b) ¿Qué se obtiene? ¿Qué hace la operación * con los valores tipo texto?

15. Haz un programa que pida al usuario el nombre de las 2 frutas que tengo en mi frutero y cree una variable para el frutero y que imprima esta última.

16. Haz un programa que pida al usuario dos textos (cadenas de caracteres) y los muestre en pantalla en orden alfabético. Pruébalo también introduciendo cadenas de caracteres que sean numéricas, por ejemplo 99 y 102 y razona sobre la salida producida.

17. Explica lo más brevemente posible qué hace la expresión: **`a=a==False`**, sabiendo que la variable **`a`** está inicializada a un valor bool (sea cual sea). Explica también qué pasaría si **`a`** no hubiera sido previamente inicializado.
18. Haz la tabla de verdad de **`p and not q or r`** teniendo en cuenta el orden de prelación de los operadores en Python**.**

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 2

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

En todos los ejercicios:

- *Elige nombres significativos para variables y subprogramas.*
- *Escribe la documentación –`docstring`– de todos los subprogramas utilizando el formato propuesto en la asignatura: parámetros, objetivo y precondiciones (si las hay).*
- *No utilices estructuras selectivas (`if`) ni iterativas (`for`/`while`)*

Ejercicios resueltos

1. Programa una función que, recibiendo los 3 coeficientes a, b y c de una ecuación de segundo grado, calcule el valor del discriminante (valor del radicando) de la misma según la fórmula:

$$raiz = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Prueba la función con un programa que la invoque con 3 valores para sus coeficientes.

SOLUCION PROPUESTA:

```
def discriminante(a, b, c):
""" float,float,float --> float
    OBJ: calcula el discriminante de la ecuación de segundo grado """
    return b**2-4*a*c

#Probador 4,5,1
print(discriminante(4,5,1))
```

2. Programa una función que determine si un número entero es impar y pruébala.

SOLUCION PROPUESTA:

```
def es_impar(n):
   """ int --> bool
       OBJ: Determina si un entero es impar o no
   """
   return (n % 2 != 0)      # alternativa n % 2 == 1

#Probador
print(es_impar(25), 'si ')    #Aprecia que este subp requiere 2 probadores
print(es_impar(24), 'no ')
```

3. Escribe y prueba una función que calcule el impuesto que debe pagar un contribuyente a partir de sus ingresos anuales y el número de hijos. El impuesto a pagar es un tercio del

ingreso imponible, siendo este último igual a los ingresos totales menos una deducción personal de 600€, a lo que se ha de añadir otra deducción de 60€ por hijo.

SOLUCION PROPUESTA:

```
def impuesto (ingresos, num_hijos):
    """ float, int --> float
       OBJ: impuesto que debe pagar un contribuyente a
       partir de sus ingresos anuales y el número de hijos. Existe una
       deducción personal de 600 euros y una por hijo de 60eur/hijo
       PRE: num_hijos >= 0                              """
    DEDUCCION_PERSONAL = 600
    DEDUCCION_POR_HIJO = 60
    ingreso_imponible = ingresos - DEDUCCION_PERSONAL
    return (ingreso_imponible - DEDUCCION_POR_HIJO * num_hijos)/3

#Probador
print(impuesto(25000,2))
```

4. **Escribe un programa modularizado que, utilizando funciones, calcule el perímetro y el área de un círculo. Construye los respectivos probadores.**

SOLUCION PROPUESTA:

```
def area_circulo(radio):
    """ float --> float
        OBJ: Calcula el area de un circulo
        PRE: radio >= 0 """
    import math
    return math.pi * radio**2

# Programa probador
radio=3.0
print ('área = ',area_circulo(radio), 'debe dar 28.274.. ')


def perimetro_circunferencia(radio):
    """ float --> float
        OBJ: Calcula el perímetro de una circunferencia
        PRE: radio >= 0 """
    import math
    return math.pi * radio * 2.0

#Programa probador
radio=3.0
print ('perímetro = ', perimetro_circunferencia(radio), 'dará 18.84... ')
```

Ejercicios propuestos (con * los que están en DOMJudge)

1. (*) Implementa una función "fuerza" que retorne el valor de la magnitud física fuerza a partir de los valores de masa y aceleración recibidos como parámetros. Construye después un programa que muestre por pantalla el valor de la fuerza a partir de una masa y aceleración dadas.
2. (*) El Barn es una unidad de superficie muy utilizada en la física de partículas, que equivale a 10^{-28} m². Para ilustrar su tamaño, ten en cuenta que un Barn es, aproximadamente, el área de la sección transversal del núcleo de un átomo de uranio. Programa dos funciones, una que permita convertir unidades en m² a Barns, y su inversa.

3. (*) La *Tasa de Interés Efectivo Anual* (TIEA) es función de la tasa nominal anual (TNA) y del número (entero) de períodos de capitalización (del número de veces en el año que los intereses pasan a formar parte del capital) según la siguiente expresión: $TIEA=(1 + TNA/m)^{m-1}$, siendo m el número de periodos total de un año (es decir, 12 si hablamos de períodos mensuales, 4 si es trimestral, 2 si es semestral, etc.). Escribe una función que calcule el TIEA a partir del TNA y el número de períodos.

4. Escribe un programa modularizado que, a partir de una hora en formato [hora, minutos y segundos] y utilizando una función que calcule el número total de segundos transcurridos desde la última medianoche, muestre dicho total posteriormente por pantalla.

5. (*) Escribe una función que determine si un punto de coordenadas en 2D está o no sobre la circunferencia $x^2+y^2=1000$. *Error frecuente*: considerar que necesitas una instrucción `if`. No es necesario utilizarlo y te pedimos de hecho que no lo utilices.

6. (*) El antiguo sistema anglosajón de unidades sigue en vigor en muchos lugares y su uso es frecuente en algunos contextos. Programa una función que determine el número de pintas que contiene una cierta cantidad de líquido expresada en mililitros, sabiendo que 1 pinta (pt) = 568,261ml.

7. (*) La temperatura expresada en grados centígrados TC, se puede convertir a grados Fahrenheit (TF) mediante la siguiente fórmula: TF = 9*TC/5 + 32. Igualmente, es sabido que −273,15 °C corresponden con el 0 Kelvin. Escribe una función devuelva la temperatura en grados Farenheit y otra en Kelvin a partir de la temperatura en grados centígrados. Escribe un programa para probarlas que a partir de una temperatura en grados centígrados obtenga e imprima por pantalla la temperatura en Kelvin y Farenheit.

8. (*) Escribe una función que a partir de las coordenadas 3D de dos puntos en el espacio en formato (x, y, z) calcule la distancia que hay entre dichos puntos. Prueba tu función.

9. Un número complejo es un número de la forma a+bi, donde a y b son números reales y el valor de i es $\sqrt{-1}$. Las cuatro operaciones aritméticas básicas sobre números complejos se definen como:
   - Suma: (a+bi)+(c+di)=(a+c)+(b+d)i
   - Resta: (a+bi)-(c+di)=(a-c)+(b-d)i
   - Producto: (a+bi)*(c+di)=(ac-bd)+(ad+bc)i
   - División: $(a+bi)/(c+di) = ((ac+bd)/(c^2+d^2)) + ((bc-ad)/(c^2+d^2))i$, suponiendo $c^2+d^2<>0$

   Programa funciones para cada una de las operaciones descritas con sus respectivos. Construye los probadores que muestren por pantalla el resultado de las operaciones reseñadas. No utilices datos de tipo `complex` de Python: deberás simularlos programando los complejos utilizando como coeficientes (a,b,c y d) cuyos valores son de tipo real.

10. Una aplicación leerá de teclado los valores necesarios para mostrar en pantalla el área de un círculo, un cuadrado y un triángulo. No queremos implementar la aplicación completa aún, sólo preparar los subprogramas de que necesitaremos disponer para en un futuro construir la aplicación (funciones que leen de teclado números y los validan, por una parte, y funciones que hacen los cálculos de las distintas áreas, por otra). Nota: si lo necesitas, utiliza como aproximación de Π (pi) el valor de que proporciona la biblioteca "math" de Python.

11. (*) El calendario gregoriano estableció (1582) que un año es bisiesto si es divisible entre 4, a menos que sea divisible entre 100. Sin embargo, si un año es divisible entre 100 y además es divisible entre 400, también resulta bisiesto. Implementa una función que determine si un año es bisiesto o no y escribe un probador para todos los posibles casos. Este probador debe mostrar en pantalla, por ejemplo, si has puesto una variable anno= 2015, una salida como la siguiente: "el año 2015 ¿es bisiesto?: False".

    Resuelve los problemas poco a poco, prueba primero si es divisible por 4. Cuando lo tengas ve añadiendo el resto de las condiciones.

    Error frecuente: considerar que necesitas una instrucción `if`. Los booleanos se pueden operar como el resto de los tipos.

12. Elabora un subprograma que resuelva una ecuación de primer grado ($a \cdot x + b = 0$). Lo más importante es determinar claramente qué tipo de subprograma es (función/procedimiento), cuáles son sus entradas, cuáles sus salidas y cómo no, ponerle un nombre significativo. Para todo lo anterior, lo más sencillo es comenzar por escribir su documentación (*docstring*). Una vez lo tengas terminado, para probarlo, crea un programa que solicite los valores de los parámetros a y b el cual debe mostrar el valor resultante de la variable x.

13. Haz un subprograma que determine los días, horas, minutos y segundos que hay en una cantidad de segundos introducida por el usuario (desprecia las fracciones de segundo). Escribe un probador para el mismo.

14. La ley de los gases ideales indica que el volumen V (en litros) ocupado por n moles de un gas a presión P (en atmósferas) y temperatura T (en grados Kelvin), responde a la siguiente fórmula:

    $$V = \mathrm{nRT/P}$$

    siendo R la constante universal de los gases cuyo valor es 0,082 atmósferas litro/Kmol. Hagamos un subprograma que calcule el volumen de un gas a partir de su presión y temperatura y probémoslo.

15. Se dice que un gas está en condiciones ideales cuando está a 1 atmosfera de presión y 0ºC, es decir 273, 15ºK. Escribe una función que determine si un cierto gas se encuentra en condiciones ideales.

16. Siguiendo con lo anterior, implementa ahora un subprograma que calcule el volumen de un gas en condiciones ideales. ¡No olvides reutilizar tu esfuerzo no reescribiendo código que ya tienes de ejercicios anteriores, sino invocándolo! Por cierto, que tu nuevo subprograma debería tener tan solo una línea de código, tenlo en cuenta. Y no olvides probarlo.

17. Cuando construyamos programas para su comercialización mimaremos el aspecto de la entrada y la salida (lo que en conjunto se llama interfaz de usuario). Posiblemente pondremos al principio un rótulo, indicando al usuario qué aplicación está usando, rótulo que irá centrado en la pantalla y subrayado, dejando una línea en blanco por encima y otra por debajo. Así:

    ```
    Aplicación de combinatoria
    ==========================
    ```

    Escribe el siguiente subprograma:

```
def centrarRotulo (rotulo):
    """string--> None
       OBJ: centra rótulo, subrayado con signos =, +línea encima y debajo
    """
    tam=len(rotulo)
    lado=((72-tam)//2)-1  # 72 columnas suele ser ancho de la pantalla
    print()
    print(' '*lado,rotulo)
    print(' '*lado , '='*tam , end='\n')

#Probador
frase = 'Don Quijote de la Mancha'
centrarRotulo(frase)                          # distinto nombre
centrarRotulo('Cervantes')                    # constante
rotulo = 'El famoso hidalgo don Quijote de la Mancha' # mismo nombre
centrarRotulo(rotulo)
```

Ejecútalo en Pythontutor (http://www.pythontutor.com/visualize.html)

Nota: Este ejercicio pretende que visualices:

- Que la declaración de un subprograma no es ejecutable
- Que los argumentos reales no tienen por qué llamarse igual, ni diferente que los argumentos formales.
- De hecho, los argumentos reales de entrada pueden ser una constante
- Aprecia que cada pieza de código tiene su propia tabla de símbolos (*frames* en Python tutor). En un momento de la ejecución hay dos variables con el mismo nombre y distinto contenido.

18. Flexibiliza el subprograma **centrarRotulo** de modo que el signo con el que se subraye y el tamaño de la ventana puedan ser distintos en diferentes ejecuciones. *Nota: Este ejercicio pretende enseñarte que la forma de flexibilizar un subprograma es pasarle parámetros.*

19. Python permite dar valores por defecto en la definición de un subprograma. Documéntate y estudia cómo se hace esto. Para ello, ten en cuenta que la presión es mucho más estable que la temperatura. *Nota: Este ejercicio pretende mostrarte que con la implementación de valores por defecto ya no te harían falta los dos ejercicios anteriores.*

20. Una biblioteca es un archivo que agrupa funciones y otros elementos de código con una lógica o temática común. Así, existen bibliotecas gráficas, matemáticas, etc. Son importantes pues, al agrupar los subprogramas en bibliotecas, se facilita y fomenta su reutilización. Algunas vienen con una distribución del lenguaje puesto que son de uso común para muchas aplicaciones (piensa en la biblioteca **math** que ya habrás seguramente utilizado) pero también es posible construirnos nuestras propias bibliotecas. En este ejercicio queremos mostrar lo sencillo que resulta reutilizar código, creando las siguientes bibliotecas para tu uso personal:

    a. Una biblioteca **bib_interfaz** de funciones de interfaz con la función **centrarRotulo** del ejercicio 17, así como los subprogramas que leen datos numéricos de teclado y que permiten construir aplicaciones como

la del ejercicio 10. Organiza todo de modo que resulte fácil reutilizarlo en el futuro, pues esta biblioteca la iremos ampliando en adelante.

b. Otra biblioteca **`bib_conversiones`** con funciones de conversión de unidades que contenga todas las funciones implementadas en los ejercicios 2, 6 y 7.
c. Una tercera **`bib_geometria`** que agrupe funciones relacionadas con la geometría, por ejemplo, las de los ejercicios 5, 8 y 10 (al menos).
d. Finalmente, una **`bib_complejos`** con las operaciones con números complejos del ejercicio 9.

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 3a

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Ejercicios resueltos

1. **Cuando se incorporan las calificaciones de los alumnos a las actas universitarias es necesario expresar la calificación numérica y también la textual. Teniendo en cuenta que las calificaciones se relacionan según el siguiente criterio:**

Rango de valores	Calificación textual
0<=nota<5	Suspenso
5<=nota<=6.5	Aprobado
6.5<nota<=8.5	Notable
8.5<nota<=10	Sobresaliente

haz un subprograma que reciba una calificación numérica comprendida entre 0 y 10 y devuelva la calificación textual.

SOLUCION PROPUESTA:

```
def nota_textual(nota):
    """
       float -> str
       OBJ: retorna el equivalente textual a una calificación numérica
       PRE: 0 <= nota <=10
    """
    if 0 <= nota < 5:
        nota_texto = "Suspenso"
    elif 5 <= nota <= 6.5:
        nota_texto = "Aprobado"
    elif 6.5 < nota <= 8.5:
        nota_texto = "Notable"
    else:
        nota_texto = "Sobresaliente"
    return nota_texto
```

Nota: *Observa que no hemos comprobado si la nota se encuentra en los límites correctos, ya que tal circunstancia se controla con la precondición.*

2. **Escribe un programa capaz de identificar si un número leído por teclado es múltiplo de 3. Si la entrada del usuario es inadecuada, el programa explicará la situación y acabará la ejecución.**

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



SOLUCION PROPUESTA:

```
try:
    n = int(input('Introduce un entero: '))
except: print('Error: Se requiere un entero')
else:
     if (n % 3 == 0):
          print("El numero introducido es múltiplo de 3")
     else:
          print("El numero introducido NO es múltiplo de 3")
```

Nota: *Observa que la solución utiliza el tratamiento de excepciones: es el único modo de evitar que el programa termine abruptamente si la entrada no es numérica. Observa también que no hemos modularizado el código en esta propuesta. El motivo es que se trata de un código tan simple que no se justifica una modularización; además, se trata de un problema muy específico por lo que la solución aportada no tiene visos de ser reutilizada en el futuro.*

Ejercicios propuestos

1. Escribe un código que pide al vendedor el importe de una compra: Si la compra es superior a 100EUR se aplica un descuento del 5% si se paga al contado, pero si el pago es con tarjeta sólo se aplica el 2%. Asegúrate de que el importe de la compra es un número válido antes de proceder a los cálculos (pista: usa `try` para comprobar que es posible convertir la entrada a un `float`).
2. Escribe un subprograma que recibe una hora en formato de 24 horas y la muestre por pantalla en notación de 12 horas (ejemplo: si recibe las 18:30 mostrará por pantalla 6:30 PM). Pruébalo utilizando diferentes valores. No es necesario validar que la entrada es correcta, pero esto debe indicarse en el docstring.
3. Como habrás observado, muy a menudo necesitamos validar la información introducida por el usuario. Programa una función "validar_entero" que se asegure de que una entrada del usuario es un entero. Ten en cuenta que dicha entrada puede ser cualquier cosa, por ejemplo, un valor real, un booleano o incluso un texto como "Hola". Nuestra función recibirá un texto y retornará verdadero si es un entero (dando por válida la entrada) o falso (rechazando la entrada). Consejo: puedes tomar ideas del ejercicio resuelto 2.
4. Escribe un algoritmo que tras leer tres enteros los escriba en pantalla en orden creciente. Como siempre, valida las entradas.
5. Programa una función que permita determinar si un determinado carácter es una vocal o no, utilizando la construcción `if-elif`. Repite el ejercicio utilizando esta vez el operador booleano **or** y observa las diferencias.

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

6. Codifica un subprograma que reciba un número entero, y si está entre 1 y 12 escriba un mensaje por pantalla indicando el mes a que dicho número corresponde. En caso contrario deberá mostrar un mensaje de error. Valida las entradas utilizando la función del ejercicio 3.

7. Una universidad acaba de modificar su sistema de representación de la calificación de los alumnos. A partir de ahora, se calificarán como "A" las notas mayores o iguales a 8,5, "B" las mayores o iguales a 6,5, "C" las calificaciones mayores o iguales a 5, "D" las calificaciones mayores o iguales a 3,5, y "F" todas las demás. Programa una función que reciba una calificación numérica y retorne la letra con la nueva calificación, luego pruébala con valores introducidos por el usuario. Tal vez sea buena idea basarte en el ejercicio resuelto número 1 pero, a diferencia de lo que se hace allí (se controla la validez con una precondición), asegúrate de que la calificación introducida es válida.

8. Obtén las raíces de una ecuación de segundo grado $ax^2+bx+c=0$ a partir de sus coeficientes a, b y c. Recuerda que si el discriminante es cero la raíz es única (-b/2xa) y que si el discriminante es negativo las raíces son imaginarias, en cuyo caso deberá indicarse con un mensaje "raíces imaginarias". Modulariza tu solución.

9. Haz un programa que pida al usuario un año, si lo introducido por el usuario es posterior a 1.582 imprimirá si es bisiesto, y si no, explicará el motivo. Recuerda reutilizar tu función **`es_bisiesto`** de cuadernos anteriores ¿Cuántas veces debes probar este ejercicio?

10. Escribe un programa que lea dos caracteres y averigüe si se introdujeron en el orden de su código ASCII o no. Se deberá además comprobar si el primero de ellos es una cifra, y en caso afirmativo indicar si es impar y si es o no primo. Modulariza la solución. En tu código no deben aparecer los valores de la tabla ASCII, porque produciría un código muy difícil de mantener, puedes usar **`ord()`** y **`chr()`** o simplemente saber que en la tabla ASCII todos los dígitos son consecutivos).

11. Escribe un programa que muestre un menú en pantalla que permita calcular el seno, el coseno y la tangente de un ángulo. El menú se mostrará hasta que el usuario decida salir. El menú quedará:

```
1. Seno
2. Coseno
...
0. Salir
Elija una opción:
```

12. Una compañía de alquiler de automóviles desea adquirir un programa para emitir sus facturas, con las siguientes consideraciones:

    a. Se factura una cantidad fija de 100EUR si no se rebasan los 300 Km.
    b. Si la distancia recorrida es mayor que 300 Km, entonces:

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

i. Si la distancia es menor o igual que 1.000 Km, se cobrarán 100EUR más el kilometraje que exceda de 300 Km, a razón de 30 céntimos/Km.
ii. Si la distancia es mayor que 1.000 Km, se cobrarán los 100EUR más el kilometraje, a razón de 30 céntimos/Km para los kilómetros entre el 300 y el 1.000, y 20 céntimos/Km para el resto.

13. En el baremo para la adjudicación de plazas en las Escuelas Infantiles Públicas se otorgan puntos en función de los ingresos de la unidad familiar, según se recoge en la siguiente tabla:

Puntos	Ingresos/mes € desde (incluido)	Hasta € (excluido)
1	5000	---
2	3500	5000
3	1800	3500
4	0	1800

Construye un subprograma que devuelva los puntos de un niño según los ingresos de su unidad familiar.

14. Un monomio está caracterizado por un coeficiente (real) y una x elevada a un exponente (entero y positivo), si bien puede no llevar x. Como es sabido, un polinomio es una suma de monomios (Ejemplo: **`+2.0x**3 + x + 4.0`**)

Para escribir en pantalla un monomio, hay que tener en cuenta que:

a. Si el coeficiente es 0 no deberá imprimirse el monomio.
b. Si el coeficiente es 1 no imprimiremos el coeficiente.
c. Si el exponente es 1 no escribiremos el exponente
d. Si el exponente es 0 no escribiremos ni el polinomio ni la x.

Haz un subprograma que imprima un monomio. Advierte que Python no escribe el "+" en los números positivos y ahora hará falta hacerlo expresamente.

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 4

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Ejercicios resueltos

1. Escribe un programa en Python que compruebe si una lista de enteros está ordenada ascendentemente.

SOLUCION PROPUESTA:

```
def en_orden_ascendente (lista):
  """lista-->boolean
  OBJ: Devuelve True si la lista está ordenada ascendentemente"""
  ordenado=True
  i = 1
  ant = lista [0]
  while i < len(lista) and ordenado:
    if lista [i] < ant:
      ordenado = False
    else:
      ant = lista [i]
      i += 1
  return ordenado

mi_lista = [3,5,1,10]
print('Ordenada: ', en_orden_ascendente(mi_lista))
```

2. Dada una lista de valores enteros, indicar cuántos están por debajo de un valor dado por el usuario.

SOLUCION PROPUESTA:

```
def cuantos_por_debajo(lista, umbral):
  """ lista, int --> int
  OBJ: Devuelve cuántos valores en la lista están por debajo del umbral"""
  i=0
  cont=0
  for valor in lista:
    if valor < umbral: cont+=1
  return cont

mi_lista = [3, 5, 1, 10, 41, 23, 19, 33]
umbral = int(input('Introduzca un valor umbral: '))
print('Hay ', cuantos_por_debajo(mi_lista, umbral), ' valores por debajo del
umbral')
```

3. Suponer que se dispone de una serie de *n* candidatos (*n* es introducido por el usuario) para una votación. Escribir un algoritmo que realice:

   a) Lectura de votos para los candidatos (finalizar introduciendo -)

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

b) Buscar el candidato ganador.

SOLUCION PROPUESTA:

```
def inicializar (lista, n):
    """lista, int-->lista
    OBJ: La lista se inicializa con todos sus valores a 0"""
    for i in range(n):
        lista.append(0)
    return

def votacion(n):
  """int-->lista
  OBJ: Devuelve los resultados de la votación para cada candidato """
  lista_resultados = []
  inicializar(lista_resultados, n)
  voto=int(input('N° del candidato al que vota (-1 para terminar): '))
  while voto!=-1:
    lista_resultados[voto]+=1
    voto=int(input('N° del candidato al que vota (-1 para terminar): '))
  return lista_resultados

def ganador(lista_votos):
  """lista-->int
  OBJ: Devuelve la posición del candidato con más votos """
  max = 0
  for i in lista_votos:
    if i > max:
      max = i
      candidato = lista_votos.index(i)
  return candidato

#Prueba
num_candidatos=int(input('Número de candidatos: '))
votos = votacion(num_candidatos)
print('El ganador es el candidato en la posicion: ', ganador(votos))
```

Ejercicios propuestos

1. Ya vimos que la validación de entradas es algo tan presente que no podemos escapar de ello, así que si te parece vamos a escribir una función que solicite al usuario introducir un entero y que no pare de pedírselo hasta que la información introducida sea válida. La idea es usar la construcción `while` combinada con la función de validación programada en cuadernos anteriores, consiguiendo una función cuya cabecera sería la siguiente:

```
def leer_entero_validado():
    """ None --> int
    OBJ: Solicita un entero al usuario, lo valida y lo retorna sólo
         cuando se ha asegurado de que es realmente un entero
    """
```

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

2. Una mejora útil al anterior ejercicio, es programar una función `entero_pedido (min, max, msj)` que pedirá un entero pero limitará su rango, es decir, el entero deberá estar comprendido entre un máximo y un mínimo. Además, la función recibirá un mensaje que será el que empleará para interactuar con el usuario. Como ayuda, te ponemos aquí cómo sería el probador:

```
#Probador
min = 1
max = 12
print(entero_pedido(min, max, f'mes entre {min} y {max}: '))
```

3. Escribe un programa que, después de preguntar ¿Cuántos números reales se van a introducir?, pida esos números y calcule su media aritmética, el mayor y el menor. El programa debe controlar que la cantidad de números es mayor de 2 y en caso contrario ha de mostrar un mensaje de error. Como siempre, valida las entradas.
4. Escribe un programa que lea una serie de números enteros hasta que se introduzca el número -9999, y cuente el total de números introducidos, el total de valores positivos y el total de valores negativos (no consideres el cero ni positivo ni negativo). Reutiliza la función diseñada en el Ejercicio 1 para validar tus entradas.
5. Implementa una variante del *Ejercicio Resuelto 4* que dibuje en pantalla un rectángulo de dimensiones dadas pero con dos símbolos, las filas pares con '@' y las impares con '#'. Esta será la versión "5a". Después de hacerlo, crea una segunda versión "5b" para que esta vez los símbolos se alternen en las columnas, es decir, las columnas pares se dibujarán con '@' y las impares con '#'.
6. Escribe un programa que pida un número límite y calcule cuántos términos de la serie armónica son necesarios para que su suma supere dicho límite. Es decir, dado un límite se trata de determinar el menor número *n* tal que:

$$1+\frac{1}{2}+\frac{1}{3}+.....+\frac{1}{n} > límite$$

   Prueba tu código, y ten en cuenta que para valores altos del límite el tiempo de cálculo se dispara. Así, para un límite = 5, n sería 83; para 10 ya asciende a 12.367 y para 15, ¡¡el número de términos es 1.835.421!! El programa ha de ser robusto y controlar que el número introducido es un entero positivo.
7. Escribe un programa que lea un entero positivo *n* y genere una tabla con las *n* primeras potencias de 2, 3 y 5. Así:

1	2	4	8	16	32	64 ...
1	3	9	27	81	243	729 ...
1	5	25	125	625	3125	15625 ...

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

8. Repasa los formatos de salida, de Python, y codifica un programa que genere la siguiente salida:

```
                Tabla de multiplicar
                ====================

    |   1   2   3   4   5   6   7   8   9  10
----+----------------------------------------
  1|   1   2   3   4   5   6   7   8   9  10
  2|   2   4   6   8  10  12  14  16  18  20
  3|   3   6   9  12  15  18  21  24  27  30
  4|   4   8  12  16  20  24  28  32  36  40
  5|   5  10  15  20  25  30  35  40  45  50
  6|   6  12  18  24  30  36  42  48  54  60
  7|   7  14  21  28  35  42  49  56  63  70
  8|   8  16  24  32  40  48  56  64  72  80
  9|   9  18  27  36  45  54  63  72  81  90
 10|  10  20  30  40  50  60  70  80  90 100
```

9. Escribe un programa que acepte un año escrito en cifras arábigas y lo visualice escrito en números romanos, dentro del rango 1 a 2000. (I=1, V=5, X=10, L=50, C=100, D=500, M=1000).

10. Cada país acuña billetes y monedas de unos determinados valores. Por ejemplo, en los países de la Unión Europea los valores son: 500.0, 200.0, 100.0, 50.0, 20.0, 10.0, 5.0, 2.0, 1.0, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01 euros. Diseña un subprograma para una caja de un supermercado, el cual reciba un importe en euros y la cantidad pagada por el cliente en una compra y muestre al cajero cuántas piezas (billetes y monedas) y de qué tipo debe dar como cambio. Dicha salida no deberá en ningún caso especificar piezas para las que haya que dar cero unidades.

11. Escribe un programa, ahora con estructuras repetitivas, que muestre un menú en pantalla que permita calcular el seno, el coseno y la tangente de un ángulo. El menú se mostrará hasta que el usuario decida salir, y mientras permitirá hacer todas las operaciones que desee el usuario. El menú quedará así:

```
1. Seno
2. Coseno
...
0. Salir
Elija una opción:
```

Sugerencia: El menú podría estar centrado, utiliza las bibliotecas ya realizadas.

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

12. La varianza poblacional de una población *p*, que dispone de *n* elementos se define como:

$$\sum_{i=0}^{n} \frac{(p_i - p_{media})^2}{n-1}$$

Construye un subprograma que devuelva la varianza de una población que recibe como parámetro en una lista.

13. Mejorar el *Ejercicio Resuelto 3*, modularizándolo y poniendo un límite al número de veces que el usuario puede intentar acertar el símbolo oculto.

14. Estudia la convergencia de la siguiente serie introduciendo un número creciente de términos y observando su tendencia. Implementa una función para la serie y un código que vaya analizando su convergencia. Comenta tus resultados.

$$\sum_{n\geq 1} \frac{(n!)^2}{2^{n^2}}$$

15. Escriba un programa modularizado en Python que genere 5 frases a partir de palabras en la siguiente lista:

```
['perro','niño','nube','padre','es','esta','come','mira','ama','el','la','al','en']
```

Las frases, que deben tener entre 3 y 10 palabras, deben generarse eligiendo un número aleatorio de palabras cada vez, de modo que la primera frase puede tener 3 palabras, mientras que la segunda podría tener 6 palabras. Una vez generada una frase, el programa la mostrará por pantalla. Y así hasta 5 frases en total.

16. Mejore el programa anterior para que genere, no 5 frases, sino tantas como sea necesario hasta que alguna de ellas tenga sentido. Para ello, después de generar una frase e imprimirla por pantalla se preguntará al usuario si considera que la frase generada tiene sentido y, si su respuesta es afirmativa, mostrará un mensaje indicando el número de frases generadas hasta que se obtuvo una con sentido.

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 4

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



Ejercicios resueltos

1. **Dada una lista de valores, indicar cuántos están por debajo de un umbral.**

SOLUCION PROPUESTA:

```
def cuantos_por_debajo(lista, umbral):
  """ lista, float -> float
  OBJ: Devuelve cuántos valores en la lista están por debajo del umbral"""
  i = cont = 0
  for valor in lista:
    if valor < umbral: cont+=1
  return cont

#probador
mi_lista = [3, 5, 1, 10, 41, 23, 19, 33]
umbral = int(input('Introduzca un valor umbral: '))
print('Hay ', cuantos_por_debajo(mi_lista, umbral), ' bajo el umbral')
```

2. **Escribe un programa en Python que compruebe si una lista de enteros está ordenada ascendentemente. NO puede utilizarse la función `sort()`**

SOLUCION PROPUESTA:

```
def en_orden_ascendente (lista):
  """ lista->boolean
      OBJ: Devuelve True si la lista está ordenada ascendentemente """
  ordenado=True
  i = 1
  ant = lista [0]
  while i < len(lista) and ordenado:
    if lista [i] < ant: ordenado = False
    else:
      ant = lista [i]
      i += 1
  return ordenado

# probador
mi_lista = [3,5,1,10]
print('Ordenada: ', en_orden_ascendente(mi_lista))
```

3. **Suponer que se dispone de una serie de *n* candidatos (*n* es introducido por el usuario) para una votación. Escribir un algoritmo que realice:**
   a) **Lectura de votos para los candidatos (finalizar introduciendo -1)**
   b) **Buscar el candidato ganador.**

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

SOLUCION PROPUESTA:

```
def inicializar (lista, tam):
    """ lista, int -> lista
        OBJ: La lista se inicializa con todos sus valores a 0
        PRE: tam > 0
    """
    for i in range(tam):
        lista.append(0)

def computar_votacion(num_candidatos):
  """ int-->lista
      OBJ: Devuelve los resultados de la votación para cada candidato
      PRE: num_candidatos > 0
  """
  lista_resultados = []
  inicializar(lista_resultados, num_candidatos)
  voto=int(input(f'Candidato (0-{num_candidatos}) al que vota, -1 terminar):'))
  while voto!=-1:
    lista_resultados[voto]+=1
    voto=int(input(f'Candidato (0-{num_candidatos}) al que vota, -1 terminar):'))
  return lista_resultados

def obtener_ganador(lista_votos):
  """ lista -> int
      OBJ: Devuelve la posición en la lista del candidato con más votos
  """
  max = 0
  for i in lista_votos:
    if i > max: max = i
  return lista_votos.index(max)

# prueba
num_candidatos = int(input('Número de candidatos: '))
votos = computar_votacion(num_candidatos)
print('El ganador es el candidato en la posicion: ', obtener_ganador(votos))
```

Ejercicios nivel básico

1. Crear una lista de enteros, inicializarlos según valores aleatorios en el rango 1..20 y computar la media de los valores, el valor más alto y el más bajo sin usar las funciones `max`, `min`, `sum` u otras funciones de biblioteca de Python.
2. Modificar una lista de números reales que representan las calificaciones de los alumnos de una clase, para sustituir en ella los valores numéricos por sus calificaciones alfanuméricas (Suspenso, Aprobado, etc.)
3. Amplía el ejercicio resuelto 3 para que se incluyan los nombres de los candidatos en una lista paralela. Es decir, tendremos de partida dos listas, una con los nombres de los candidatos y otra con los votos, de forma que el candidato de la posición i-ésima tendrá su nombre en la lista "nombres" y sus votos en la lista "votos". Cuando termine la votación debe mostrarse un histograma con asteriscos con la distribución de votos por candidato, es decir, los nombres de los candidatos y a su derecha tantos asteriscos como votos haya recibido.

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

4. Programa una función que dadas dos listas determine el valor máximo del peso de las mismas, entendiendo que el peso de una lista corresponde a la suma de los valores absolutos de todos sus elementos. Ejemplo: Peso([1, -2, 3]) = 6.
5. Escribe un programa que "codifique" una frase modificando todas las vocales según el siguiente código: a por 4, e por 3, i por 1, o por 0 y u por el símbolo #. Por ejemplo, la frase: "El perro del hortelano", será: "3l p3rr0 d3l h0rt3l4n0".
6. Escribe un programa que obtenga 10 números aleatorios y almacene los pares en una lista y los impares en otra distinta. Tras ello, deberá comprobar qué lista suma más, reutilizando el código del ejercicio anterior.
7. Escribe una función que sume dos listas de igual longitud y retorne otra lista que contenga la suma de las originales elemento a elemento.
8. Modifica el ejercicio anterior permitiendo sumar listas de distinto tamaño. Los elementos sobrantes de la lista más larga se añadirán al final de la lista resultante.
9. Crea un subprograma que, a partir de una lista que contiene ceros o elementos booleanos, produzca un dibujo en pantalla. Cada True se representará por un asterisco, cada False por un ampersand y el cero significará salto de línea. Así, la lista [True, True, False, True, 0, True, False, False, 0, False, False, True, True] se representará como:

```
**&*
*&&&
&&**
```

10. Implementa una función que proporcione la posición del primer valor impar de una lista de valores enteros. Contempla el hecho de que la lista no contenga impares.

Ejercicios nivel intermedio

11. Reprograma el ejercicio básico 6 para que los números generados y almacenados en las diferentes listas no tengan repetidos.
12. Realiza una aplicación que permita sumar dos matrices de tamaño 3x4 utilizando listas de listas. Las matrices contienen números reales. Debe incluir las siguientes tareas: generar datos aleatorios para la matriz A, generar datos para la matriz B y mostrar el resultado de matriz A + matriz B.

    Ejemplo de lista de listas: [[1,0],[4,1]] correspondería a la matriz:

    1 0
    4 1

13. Implementa una función que ponga en mayúsculas todas las primeras letras de las palabras de una frase.

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

14. Implementa una función que indique si una palabra contiene las cinco vocales: por ejemplo "murciélago". Modifica posteriormente la función para que detecte sólo aquellas palabras que contienen una única vez cada vocal.
15. El código QR (abreviatura de *Quick Response Code*) fue creado en 1994 por una filial japonesa de Toyota que fabrica componentes de automóviles. Estos códigos permiten almacenar información sobre un producto codificándola en un cuadrado de NxN con píxeles que pueden ser blancos o negros. En su formato más pequeño, los códigos tienen 21x21 píxeles (versión 1), y en la más grande 177x177 (versión 40). Programa un subprograma que genere aleatoriamente códigos QR versión 1 y muestra alguno por pantalla utilizando, por ejemplo, asteriscos.
16. Programa una función que compruebe si una lista es simétrica con respecto a su elemento central. Es decir, el elemento de la primera posición es igual al de la última, el segundo al de la penúltima, etc. Ejemplo de lista simétrica:

    [1, 3, 5, 0, 5, 3, 1]
17. Programa una función que reciba el texto de un tweet y retorne una lista con todos los hashtags (#tag) que contenga y otra con las menciones (@usuario). La función debe eliminar de las palabras los signos de puntuación tales como '.,;' para lo cual utilizará `strip()`.
18. Programa una función que reciba una lista con palabras (obligatoriamente incluye las palabras inicio y fin en algún lugar) y retorne un texto que contenga en orden los elementos entre la palabra "inicio" y la palabra "fin". Ejemplo: La lista ["Desde", "el", "inicio", "de", "nuestra", "relación", "me", "temía", "que", "el", "fin", "llegaría", "pronto"] tras pasarse a la función retornaría el texto "de nuestra relación me temía que el".
19. Un texto contiene comandos en forma de frases separadas por puntos. En cada frase, la primera palabra contiene el código de la operación y la última el resultado. Crea una lista de parejas [código-resultado] utilizando como entrada un texto con el formato indicado. Ejemplo: SUM-3-30-33. POW-4-2-16. NOT-True-False. Resultado: [(SUM,30), (POW,16), (NOT,False)]
20. Dada una lista y un conjunto de elementos, eliminar todas las ocurrencias de dichos elementos en la lista. Ejemplo: [1,2,3,4,5,6,7,8] sería la lista original, queremos eliminar [2,4], lista resultante: [1,3,5,6,7,8]

Ejercicios nivel avanzado

21. Programa una función que limpie una lista de tuplas eliminando las que no estén enlazadas con la secuencia inicial de manera similar a cómo se unen las fichas de dominó: el segundo elemento de la tupla i es el mismo que el primero de la tupla i+1. Por ejemplo lista = [(4, 5), (5, 6), (6, 5), (1, 3), (0, 0)] eliminaría las dos últimas tuplas ya que las 3 primeras sí están enlazadas entre ellas.

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

22. Implementa una aplicación que solicite 10 valores y los almacene en una lista ordenadamente (no en orden de entrada, sino en orden creciente de valor). A continuación, recorre la lista restándole a cada valor el valor que se encuentre en la siguiente posición (ej.: [1, 3, 5], resultado [(1-3), (3-5), (5-1)] = [-2, -2, 4]. Modulariza tu solución.
23. Elimina elementos duplicados en una lista de listas, considerando duplicados aquellos elementos que aparecen más de una vez en la misma sublista, pero no en la misma lista. Ejemplo: [[1,2,2,3,2,4], [3,5,2,1,4,5,9], [1,10,2,4,5]] , en la primera sublista el 2 está duplicado pero que haya otro 2 en las otras sublistas no es problemático, por lo que la salida sería: [[1,2,3,4], [3,5,2,1,4,9], [1,10,2,4,5]]
24. Distribuir 20 datos enteros en dos listas de tal manera que se vayan generando dos secuencias ordenadas, una creciente y otra decreciente. El resultado final serán dos listas, cada una de las cuales contendrá los 20 elementos originales ordenador pero la ordenación será distinta (en una creciente y en otra decreciente). IMPORTANTE: Las listas no pueden almacenarse y luego ordenarse de un golpe utilizando la función `sort()` de Python, sino que se van ordenando a medida que se insertan los elementos que las componen.
25. Escribe un subprograma que invierta los valores de las filas de orden k en una matriz (representada mediante una lista de listas). Por ejemplo `[[6,3,2], [8,6,3], [4,15,2], [3,6], [3,7,0], [2,9]]`, `K = 2` invierte la 2ª, 4ª y 6ª filas, produciendo la matriz: `[[6,3,2], [3,6,8], [4,15,2], [6,3], [3,7,0], [9,2]]`
26. Dada una matriz, generar otra con las filas duplicadas en la primera. Ejemplo:

```
Entrada : [[1, 1, 0, 1, 0, 1],
           [0, 0, 1, 0, 0, 1],
           [1, 0, 1, 1, 0, 0],
           [1, 1, 0, 1, 0, 1],
           [0, 0, 1, 0, 0, 1],
           [0, 0, 1, 0, 0, 1]]

Salida : [1, 1, 0, 1, 0, 1]
         [0, 0, 1, 0, 0, 1]
```

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 5

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Ejercicios resueltos

1. **Utilizando diccionarios, programa una estructura de datos que permita almacenar fechas como la agrupación de tres valores enteros positivos: dia, mes y año. Utilizando lo anterior, crea una función que calcule si una fecha es anterior a otra.**

SOLUCION PROPUESTA:

```
ayer = {'dia':10, 'mes':11, 'anno': 2023}
hoy =  {'dia':11, 'mes':11, 'anno': 2023}

def es_anterior(fecha1,fecha2):
    """ Fecha, Fecha -> bool
    OBJ: Calcula si una fecha es anterior a otra
    PRE: fecha1 y fecha2 son fechas válidas """
    if (fecha2['anno'] > fecha1['anno']):     respuesta = True
    elif (fecha1['anno'] > fecha2['anno']):   respuesta = False
    else: # si los dos años son iguales
        if (fecha2['mes'] > fecha1['mes']):   respuesta = True
        elif (fecha1['mes'] > fecha2['mes']): respuesta = False
        else: # el mes y el año son iguales
            respuesta = (fecha2['dia'] > fecha1['dia'])
    return respuesta

#probador
print(es_anterior(hoy,ayer))
```

2. **Se pide ahora programar una función que evalúe si dos fechas son iguales y otra que indique si una fecha es posterior a otra. Trate de reutilizar la función del problema 1.**

SOLUCION PROPUESTA:

```
def iguales (fecha1,fecha2):
    """Fecha, Fecha -> bool
    OBJ: Calcula si una fecha es igual a otra
    PRE: fecha1 y fecha2 son fechas válidas """
    return fecha1['dia'] == fecha2['dia'] and \
        fecha1['mes'] == fecha2['mes'] and \
        fecha1['anno'] == fecha2['anno']

def es_posterior(fecha1,fecha2):
    """ Fecha, Fecha -> bool
    OBJ: Calcula si una fecha es posterior a otra
    PRE: fecha1 y fecha2 son fechas válidas """
    return not es_anterior(fecha1,fecha2) and not iguales(fecha1,fecha2)

#probador
print(es_anterior(hoy,ayer))
print(es_posterior(hoy,ayer))
print(iguales(hoy,hoy))
```

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



3. **Haz un programa que cree un diccionario con personas (nombre-edad) y permita buscar si una persona está en el mismo. Al final se mostrará un informe con el contenido del diccionario.**

SOLUCION PROPUESTA:

```
edades = {  'Susana': 23,
            'Pedro': 19,
            'Andres': 78,
            'Angela': 45 }

nombre_buscado = input('Dime un nombre para buscarlo: ')
if nombre_buscado in edades:
    print (nombre_buscado, " está en el diccionario. Su edad es: ",\
                           edades[nombre_buscado], " años")
else:
    print (nombre_buscado, " no está en el diccionario")

#probador
print ("\nLas siguientes personas están en el diccionario:")
for persona in edades: print(persona, end='\t')
      print ("\nSus edades son...")
for edad in edades.values(): print(edad, end='\t')
      print ("\nEl diccionario tiene ", len(edades), " entradas")
```

Ejercicios nivel básico

1. Implementa las estructuras de datos necesarias para almacenar información anónima sobre personas con objeto de hacer un estudio estadístico. Así, se deberá almacenar el número de secuencia, sexo y edad de cada persona. Programe además un par de funciones para (a) leer por teclado datos relativos a una persona y (b) para mostrar dichos datos por pantalla.
2. Sin utilizar listas, programa un software que utilice lo programado en el ejercicio 1 para leer la edad de 10 personas y calcular la media de edad general, la media por sexo, la cantidad de mujeres que tienen entre 13 y 16 años y el número de hombres menores de 20 años.
3. Amplía el ejercicio anterior mostrando al final del proceso los datos completos de la mujer y el hombre más jóvenes de todos los introducidos.
4. Utilizando diccionarios implementa una estructura que de soporte a los puntos en 2D, es decir, con dos coordenadas x e y. Programa funciones para su suma y resta.
5. Implementa una aplicación para ayudar en la gestión de cobros de una gasolinera. Mediante una lista de diccionarios deberás calcular el gasto de un total de 10 clientes. Los diccionarios serán utilizados para almacenar el gasto de cada cliente en gasolina, el gasto en productos de la tienda y el descuento en tanto por ciento. La aplicación:
    a. Solicitará por teclado los gastos de gasolina, de la tienda y el descuento para cada uno de los 10 clientes.
    b. Mostrará por pantalla la suma de los gastos para cada cliente y el total.

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

6. Programa una función distancia_2D que calcule la distancia entre dos puntos. La función retornará un número real según la siguiente fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

7. Realiza una aplicación que permita sumar dos matrices de tamaño 3x4. Las matrices contienen datos sobre Puntos2D (diccionarios tipo {x:20, y:-3}). Debes escribir funciones que permitan hacer las siguientes tareas:
   a. Leer datos para una matriz A.
   b. Rellenar los datos de las matrices A y B.
   c. Calcular el resultado de matriz A + matriz B
   d. Mostar por pantalla la matriz resultado

Ejercicios nivel intermedio

8. Realizar un programa que lea palabras hasta que se introduzca "*fin*", mostrando una estadística de las longitudes de las palabras, es decir, el número total de palabras de longitud 1 que se hayan introducido, el total de longitud 2, etc. No hay máxima longitud de las palabras. Una posible salida de este programa sería:

```
Palabras longitud 1: ninguna
Palabras longitud 2: 10
Palabras longitud 3: 21
Palabras longitud 4: 54
etc.
```

9. Programa una función que reciba un diccionario y una lista y que como salida genere dos listas:
   - una con todos los valores de aquellos elementos de la lista original que están en el diccionario,
   - otra lista con los valores de los elementos que NO están en el diccionario.

10. Si tenemos un diccionario que contiene como claves el nombre de una persona y como valor una lista con sus preferencias personales (*deportes*, *amigos*, *música*, etc.), es posible programar una función `agregaPreferencia(diccionario, persona, preferencia)` tal que:
    - Si la persona no existe la agrega al diccionario con una lista que contiene un solo elemento.
    - Si la persona existe y la preferencia actual existe en su lista, no tiene ningún efecto, pero si dicha preferencia no existe en su lista, la agrega a la lista.

Programación I
Cuaderno de Trabajo 5

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Ejercicios avanzados

11. Supón que tenemos 2 diccionarios, uno con la plantilla del FC Barcelona (22 jugadores, parejas "dorsal" y "nombre que aparece en la camiseta") y otro con las alineaciones de las distintas jornadas de la Liga 2023-2024 (número de jornada y lista de jugadores que jugaron esa jornada). Programa funciones para:
    - Introducir los datos de una jornada a partir de su número y una lista de jugadores y dorsales. Obviamente no pueden repetirse ningún jugador en la alineación ni participar jugadores que no pertenecen a la plantilla del Barcelona FC.
    - Determinar si un jugador participó o no en una cierta jornada a partir de su dorsal o de su nombre (como se prefiera).
    - Listar todas las jornadas en que jugó un determinado jugador cuyo dorsal se indica.
    - Mostrar el nombre del jugador que más partidos jugó en una temporada, indicando el número de partidos en que intervino.
12. Realiza una aplicación que permita comprobar si un tablero de Conecta4 (https://es.wikipedia.org/wiki/Conecta_4) ha completado las 4 en raya. La aplicación deberá:
    a. Solicitar datos para un tablero de 7x6
    b. Solicitar introducir valores para representar las fichas del tablero en un momento concreto del juego
    c. Mostrar por pantalla el tablero con los valores introducidos
    d. Indicar si alguna fila, columna o diagonal incluye 4 en raya (4 casillas consecutivas con una ficha de igual color).

    Nota: Para modelar las casillas del tablero utiliza una estructura de datos con dos campos, uno que guardará si hay ficha o no (valor booleano) y otro de qué color es la ficha (amarilla o roja). Si en una casilla no hay ficha, el valor del color no tendrá validez y por tanto se ignorará.

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 6

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



Ejercicios resueltos

1. **Realiza una función que calcule el factorial de un entero empleando recursividad.**

SOLUCION PROPUESTA:

```
def factorial(n):
    """  int -> long
         OBJ: funcion factorial n!
         PRE: n >= 0
    """
    if n in (0,1):  result = 1
    else:           result = n * factorial(n-1)
    return result

# Prueba
print(factorial(6))
```

2. **Implemente una función recursiva que dada una lista averigüe si todos los elementos de la misma son iguales a uno dado.**

SOLUCION PROPUESTA (sin slicing):

```
def todos_iguales(A, ele, actual):
    """ lista, objeto, int -> bool
        OBJ: comprueba si todos los elementos de una lista son iguales a uno dado
        PRE: La lista debe contener al menos un elemento
    """
    if (actual == len(A)-1): son_iguales = (A[actual] == ele)
    else:  son_iguales = (A[actual] == ele) and (todos_iguales(A,ele,actual+1))
    return son_iguales

# Prueba
L = [1,1,1,1,1]
print(todos_iguales(L,1,0)) # Debe mostrar True
L2 = [1,2,3,1,1]
print(todos_iguales(L2,1,0)) # Debe mostrar False
```

SOLUCION PROPUESTA (utilizando *slicing*):

```
def todos_iguales(A, ele):
    """ lista, objeto -> bool
        OBJ: comprueba si todos los elementos de una lista son iguales a uno dado
        PRE: La lista debe contener al menos un elemento
    """
    if (len(A) == 1):  result = (A[0] == ele)
    else:              result = (A[0] == ele) and (todos_iguales(A[1:len(A)],ele))
    return result

# Prueba
print(todos_iguales([1,1,1,1,1],1)) # Debe mostrar True
print(todos_iguales([1,2,3,1,1],1)) # Debe mostrar False
```

Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 6

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



Ejercicios nivel básico

1. Programar un subprograma recursivo que compruebe si un cierto número se encuentra o no en una cierta lista.

```
def en_lista(lista,num):
    """ list, int -> bool
        EJ1: Comprueba si un cierto número se encuentra o no en la lista
    """
    if lista==[]: return False
    else: return lista[0]==num or en_lista(lista[1:], num)

# Probadores
print(en_lista([1, 2, 3, 5, 4], 5))
print(en_lista([1, 2, 3, 5, 4], 10))
```

2. Implementa un subprograma recursivo que, dados dos números, calcule su división utilizando restas sucesivas (no puedes utilizar el operador de división).

```
def division_restas_sucesivas(a,b):
    """ int, int -> bool
        EJ2: Calcula a//b utilizando restas sucesivas (recursivo)
        PRE: b > 0
    """
    if a < b:  return 0
    else: return 1 + division_restas_sucesivas(a-b, b)

# Probadores
print(division_restas_sucesivas(20,5))
print(division_restas_sucesivas(20,3))
```

3. Programa una función recursiva que, dada una palabra, retorne la misma invertida.

```
def invertida(palabra):
    """ str -> str
        EJ3: Dada una palabra, retorna la misma invertida (recursivo)
    """
    if len(palabra)==0: return ''
    else: return palabra[-1] + invertida(palabra[:-1])

# Probador
print(invertida('IPCC'))
```

Programación I
Cuaderno de Trabajo 6

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

4. Realiza una función recursiva que cuente los dígitos de un número entero.

```
def digitos(n):
    """ int -> int
        EJ4: Cuenta los dígitos de n (recursivo)
        PRE: n>=0
    """
    if n<10: return 1
    else: return 1 + digitos(n//10)

# Probador
print(digitos(1235))
```

5. Realizar un programa que lea desde teclado un número entero positivo y lo convierta a binario utilizando para ello una función recursiva.

```
def binario(num):
    """
    EJ5: Convierte un entero a binario utilizando recursividad
    :param num: número entero positivo
    :return: bool
    """
    if num in (0,1): return str(num)
    else: return binario(num//2) + str(num%2)

# Probador
print(binario(int(input('Introduce un entero positivo: '))))
```

6. Calcular la suma de los números pares entre 0 y n de manera recursiva.

```
def suma_pares(n):
    """ int -> int
        EJ6: Suma de los pares entre 0 y n (recursivo)
        PRE: n>=0
    """
    if n==0: return 0
    elif n%2 == 0: return n + suma_pares(n-2)
    else: return suma_pares(n-1)

# Probadores
print(suma_pares(7))
print(suma_pares(8))
```

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

7. Programar un algoritmo recursivo que obtenga la suma de las edades de todos los elementos de una lista de alumnos. Un alumno está caracterizado por tres atributos (*nombre, edad, titulación*).

```
def suma_edades(lista_alumnos):
    """ list[tuple] -> long
        EJ7: Suma de edades de items en lista (nombre, edad, titulacion)
        PRE: edades>0
    """
    if len(lista_alumnos)==0: return 0
    else: return lista_alumnos[0][1] + suma_edades(lista_alumnos[1:])

# Probador
lista = [('Sue', 19, 'GIA'), ('Matt', 21, 'GII'), ('Eddie', 21, 'GIA')]
print(suma_edades(lista))
```

8. Programa una función que retorne la posición en una lista del primer número impar (es decir, el más cercano al elemento de la posición "0").

```
def primer_impar(lista):
    """ list -> int
        EJ8: Posición del primer impar o None si no hay impares
    """
    if lista == []: return None
    elif lista[0]%2 != 0: return 0
    else:
        posicion_impar_en_resto = primer_impar(lista[1:])
        if posicion_impar_en_resto == None: return None
        else: return 1 + posicion_impar_en_resto

# Probadores
print(primer_impar([18, 24, 3, 5, 4]))
print(primer_impar([18, 24, 34, 54, 4]))
```

9. Programa una función que cuente el número de 'aes' en un texto. Por ejemplo, la frase "dábale arroz a la zorra el abad" contiene 8 aes.

```
def cuenta_aes(frase):
    """ str -> int
        EJ9: cuenta el número de 'aes' en un texto
    """
    if frase == '': return 0
    elif frase[0]=='a': return 1 + cuenta_aes(frase[1:])
    else: return cuenta_aes(frase[1:])

# Probador
print(cuenta_aes('En un lugar de la Mancha...'))
```

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

10. Con el siguiente procedimiento ya programado y probado, esboza un pseudocódigo de un subprograma recursivo que haga sonar todas las cuerdas de una guitarra.

```
def sonar(g,m):
""" guitarra, int --> None
    OBJ: hace sonar la cuerda m de la guitarra g
"""
```

```
def tocar_guitarra(g, cuerda_actual):
    """ guitarra -> None
        EJ10: hace sonar todas las cuerdas de la guitarra g
    """
    sonar(g,cuerda_actual)
    if cuerda_actual > 1: tocar_guitarra(g, cuerda_actual-1)

#probador
tocar_guitarra(mi_guitarra, 6)
```

Ejercicios nivel avanzado

11. Escribe y prueba una función recursiva que determine si un texto contiene todas las vocales o no.
12. Programa una función recursiva que reemplace las 'O' por 'X' en una matriz NxN.
13. Programa en Python una función recursiva que reciba un entero n y retorne la fila del triángulo de Pascal correspondiente a la misma.
14. Escribe un procedimiento recursivo que calcule el prefijo común a todas las palabras de una lista. Ejemplo: el prefijo común a las palabras ['ancient', 'anne, 'ant'] es 'an'
15. Investiga en Internet qué es la técnica conocida como "*memoizing*" y aplícala en la función recursiva que calcula los términos de la serie de Fibonacci.

```
def fibonacci(n, calculados):
    """ int, dict -> long
        EJ15: término n de la serie de Fibonacci usando memoization
        pre: n>=0
    """
    if n <= 1:  return n
    else:
        if n not in calculados:
           calculados[n] = fibonacci(n-1, calculados) +
                           fibonacci(n-2, calculados)
        return calculados[n]

# Ejemplo de uso
fiboterms = [fibonacci(i, {}) for i in range(100,120)]
print(f"Términos entre 100 y 120 en la serie de Fibonacci: {fiboterms}")
```

Universidad Rey Juan Carlos

Programación I

Cuaderno de Trabajo 6

©2025 - Algunos derechos reservados.

Isaac Lozano Osorio (isaac.lozano@urjc.es)

Salvador Sánchez Alonso (salvador.sanches@urjc.es)

Material docente en abierto de la Universidad Rey Juan Carlos

Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



Universidad Rey Juan Carlos

Programación I
Cuaderno de Trabajo 7

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Ejercicios resueltos

1. **Realiza un procedimiento que reciba una lista de números reales y ordene todos sus elementos de forma ascendente empleando el método de la burbuja.**

SOLUCION PROPUESTA:

```
def burbuja(lista):
    """ lista -> None
        OBJ: Ordena la lista ascendentemente, método burbuja
    """
     for pasada in range(len(lista)-1,0,-1):
         for i in range(pasada):
             if lista[i] > lista[i+1]:
                 lista[i] lista[i+1] = lista[i+1], lista[i]
```

2. **Implementa el método de la burbuja para el mismo caso anterior (lista de reales) pero esta vez con ordenación descendente y permitiendo ordenación parcial entre dos posiciones inicial y final de la lista (esto es, no necesariamente hay que ordenar la lista completa sino solo los valores entre dos posiciones dadas).**

SOLUCION PROPUESTA:

```
def burbuja2(v,inicio,fin):
    """ lista -> None
        OBJ: Ordena la lista ascendentemente, método burbuja
    """
   for pasada in range(fin):
     for i in range (fin,pasada,-1):
       if (v[i] < v[i-1]):
         v[i], v[i-1] = v[i-1], v[i]
```

Programación I
Cuaderno de Trabajo 7

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".



Ejercicios propuestos

1. Implementa un programa que, dado un número finito de fechas almacenadas en una lista, nos permita buscar una fecha por año, mes o día, así como la ordenación de las mismas por año. Nota: emplea el método de búsqueda binaria.
2. Realiza un programa que almacene 10 frases diferentes en una lista y nos permita ordenar la misma de forma descendentemente utilizando como criterio la longitud de la frase. Utiliza el método de inserción directa.
3. Modifica el programa anterior para que ordene en función del número de palabras que hay en la frase. Usa ahora el método de selección. Pista: será útil utilizar la función `split()`
4. Analiza la siguiente variante del método de la burbuja, compárala con las implementaciones de la parte "ejercicios resueltos" y comenta tus observaciones.

```
def burbuja3(lista):
   """ lista -> None
       OBJ: Ordena ascendentemente por bubble sort
   """
   n = len(lista)
   for i in range(n):
      intercambio = False
      for j in range(0, n-i-1):
         if lista[j] > lista[j+1]:
            lista[j], lista[j+1] = lista[j+1], lista[j]
            intercambio = True
      if not intercambio: break
```

5. Realiza un programa para gestionar los datos de stock de un comercio. La información a recoger será: nombre de producto, precio y cantidad en stock. El comercio vende 180 productos distintos. El programa debe ser capaz de:
   a. Dar de alta un producto nuevo
   b. Buscar un producto por su nombre, empleando búsqueda secuencial.
   c. Ordenar los productos según su precio.
6. Haz un programa para gestionar las notas de una clase de 50 alumnos de los cuales sabemos su nombre y la nota obtenida. El programa debe permitir:
   a. Introducir los datos de los alumnos por teclado.
   b. Dado un nombre de un alumno, buscarlo y modificar su nota, mostrando el resultado por pantalla (empleando búsqueda secuencial).
   c. Dado un nombre de un alumno, buscarlo y mostrar la información por pantalla empleando búsqueda binaria.
   d. Realizar la media de las notas superiores o iguales a 5.
   e. Realizar la ordenación (método de selección) de los alumnos por notas en orden descendente y mostrar la nota del alumno con mejor nota.
   f. Utilizando el método de inserción, realizar la ordenación de los alumnos por notas en orden ascendente.

Programación I
Cuaderno de Trabajo 7

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Material docente en abierto de la Universidad Rey Juan Carlos
Esta obra está bajo una licencia Creative Commons "Atribución Compartir Igual 4.0 Internacional".

Ejercicio avanzado

El puzzle 5 del Advent of Code 2024 (https://adventofcode.com/2024/day/5) es un interesante ejemplo de cómo adaptar un método de ordenación a una lista de elementos cuando el criterio de ordenación no es tan simple como "es mayor que" o "es menor que".

El problema parte de un listado de reglas de ordenación (en formato `numero|numero`) y después un listado de actualizaciones en forma de listas de números. Los números de dichas listas corresponden a las páginas de un manual, los cuales deben actualizarse pero solo si la regla es correcta, de tal forma que, por ejemplo, puesto que hay una regla [47|53], no será válida ninguna actualización en la cual aparezca el número 47 después del 53. Así, la actualización (75,47,61,53,29) es correcta, dado que 47 aparece antes que 53, pero no lo será (75,97,47,61,53) pues viola la regla [97|75].

```
47|53
97|13
97|61
97|47
75|29
61|13
75|53
29|13
97|29
53|29
61|53
97|53
61|29
47|13
75|47
97|75
47|61
75|61
47|29
75|13
53|13

75,47,61,53,29
97,61,53,29,13
75,29,13
75,97,47,61,53
61,13,29
97,13,75,29,47
```

En la parte 2, que es la que nos interesa aquí, nos piden extraer las actualizaciones incorrectamente ordenadas y ordenarlas. Por ejemplo:

- 75,97,47,61,53 (desordenada) → 97,75,47,61,53 (OK)
- 61,13,29 (desordenada) → 61,29,13 (OK)
- 97,13,75,29,47 (desordenada) → 97,75,47,29,13 (OK)

Para ordenar una actualización debemos aplicar como criterio de ordenación las reglas que nos dan. Puesto que el método de ordenación no nos lo imponen, podemos utilizar cualquiera de los que conocemos. Aquí proponemos usar el método de la burbuja por su simplicidad.

Dado que al final lo que nos piden es la suma de los números centrales de cada actualización (dicha suma es la solución al puzzle) lo que hacemos es procesar las actualizaciones -ordenando las desordenadas- y quedarnos con el número central, sumarlo y finalmente imprimirlo. Así, el código principal que guía nuestras acciones es el siguiente:

```
corrected_updates = 0
for update in updates:
    if not is_correct(update, successors):
        bubble_sort(update, successors)
        corrected_updates += update[len(update)//2]
print("Part II: ", corrected_updates)
```

Siendo `update` la lista de números de una actualización, por ejemplo [75,97,47,61,53], y `successors` un diccionario donde hemos almacenado todos los números de página y para cada una, una lista con las páginas que deben aparecer tras ella:

```
successors = { 47: [53,13,61,29], 97: [13,61,47,29,53,75], 75: [29,53,47,61,13], etc. }
```

Hemos cargado todos los datos del input en `successors` y creado una lista de listas llamada `updates` con todas las actualizaciones. También se ha implementado una función `is_correct()` que nos dice si una actualización está ordenada o no. El bucle principal es el que mostramos más arriba… Todo esto no tienes que implementarlo. Para completar el reto solo queda implementar la función **`def bubble_sort(linea,sucesores)`** ¿Te atreves?

Universidad
Rey Juan Carlos

Prueba escrita 1 – Duración: 1h45’	Grado en Inteligencia Artificial
Asignatura: Programación I	Fecha: 30 de Octubre de 2024
SOLUCIONES	

I. CUESTIONES TEÓRICO-PRÁCTICAS

1. Dadas las siguientes líneas de código:

```
a = 100
while True:            # bucle 1 (externo)
   while a < 100:      # bucle 2 (interno)
      a -= 1
   a = 99
   if a < 0 : break
```

a) El cuerpo del bucle externo se ejecuta 101 veces y el del bucle interno 100.
b) El cuerpo del bucle externo se ejecuta infinitas veces y el del bucle interno ninguna.
c) El cuerpo del bucle externo se ejecuta 1 vez y el del bucle interno ninguna.
d) Ninguna de las anteriores es correcta.

2. Indica la salida en pantalla del siguiente fragmento de programa en Python:

```
def funcion_misteriosa(n):
   """ Se han eliminado los docstrings aquí para mayor misterio!!!
   """
   x = 5
   while (x <= n):
     for i in range (0,8):
        print("*" * (n-i))
     x = x - 1
funcion_misteriosa(8)
```

a) Muestra por pantalla infinitos triángulos invertidos cuyo lado mayor es 8
b) Muestra por pantalla un triángulo de asteriscos de lado mayor infinito
c) Muestra por pantalla un triángulo de asteriscos invertido cuyo lado mayor es 8
d) Muestra por pantalla un triángulo de asteriscos invertido cuyo lado menor es 8

3. Si *calculo* es el nombre de una función que recibe un parámetro, ¿cuál de las siguientes es la forma más correcta y habitual de invocarla?

a) `calculo(d)`
b) `x = calculo(d)`
c) `calculo = d`
d) `d = calculo`

Universidad
Rey Juan Carlos

4. Dado el siguiente programa, indica su salida por pantalla:

```
def fUno (x):
      if x<10: x+=1
      if x>=10: x-=1
      return x

def fDos (x):
      if x<10: x+=1
      else: x-=1
      return x

print (fUno(9)," y ", fDos(9))
```

a) 10 y 9
b) 10 y 10
c) 9 y 9
d) 9 y 10

5. Completa las partes punteadas del siguiente código para que el programa muestre por pantalla los números entre 1 y 10.000 que cumplan la siguiente condición: la suma de todos los enteros entre 1 y un número dado k es igual a la multiplicación de todos los enteros entre 1 y el número. Por ejemplo, para el entero 3 se cumple la condición anterior, ya que 1 + 2 + 3 = 1 x 2 x 3.

```
1. for k in range (2, 10000):
2.    ……
3.    ……
4.    for …… :
5.      sum+=j
6.      mul*=j
7.    if …… :
8.      print("El numero", k,"cumple la condición ")
```

a) Línea 2: *vacía*, Línea 3: *vacía*, Línea 4: j in range(1, k), Línea 7: sum==mul
b) Línea 2: *vacía*, Línea 3: *vacía*, Línea 4: j in range(1, k+1), Línea 7: sum==mul
c) Línea 2: sum=0, Línea 3: mul=1, Línea 4: j in range(1, k+1), Línea 7: sum==mul
d) Línea 2: sum=0, Línea 3: mul=1, Línea 4: j in range(1, k), Línea 7: sum==k

6. Indique cuáles serían los operadores necesarios en la sentencia punteada para que el resultado de la ejecución del siguiente código sea “A A”. En cada espacio … puede aparecer un operador o ninguno.

```
x = True
y = False
sigue = True
while sigue:
  print ("A ")
  x = y or not x
  if  ____________________________________ :  sigue=False
```

a) not (y or (x and not y))
b) y or (x or not y)
c) Existen varias soluciones correctas
d) y or (not x and not y)

Universidad
Rey Juan Carlos

7. ¿Para qué sirven los *docstrings* en Python?

   a) Para representar texto en nuestros programas
   b) Para crear funciones más eficientes y robustas
   c) Para documentar nuestro código, y son especialmente útiles porque la función `help` aplicada a un subprograma permite obtener dicha documentación
   d) Son útiles solo en casos muy concretos, pues hacen perder tiempo de desarrollo

8. La diferencia entre las instrucciones `break` y `continue` reside en…
   a) `break` fuerza la ejecución de la siguiente iteración de un bucle, mientras que `continue` fuerza la salida al bucle más externo
   b) `continue` fuerza la ejecución de la siguiente iteración de un bucle, mientras que `break` fuerza la terminación del bucle actual
   c) `break` fuerza la ejecución de la siguiente iteración de un bucle, mientras que `continue` fuerza la salida del bucle actual
   d) En bucles anidados el comportamiento de ambos es análogo

9. En el siguiente código…

```
def f1(a):
  return(int(a)*2)

def f2(a):
  print(int(a)*2)
```

   a) Hay una diferencia evidente entre return y print pues una invocación a `f1` puede emplearse en otra expresión numérica mayor y una invocación a `f2` no puede
   b) Es evidente que en un caso así tanto `return` como `print` son equivalentes
   c) Si escribimos una instrucción como `print(f1(4))` o `print(f2(4))` se ve claramente que los comportamientos de `return` y `print` son iguales
   d) El formato de `f2()` es incorrecto ya que no incluye cláusula `return`

10. Si se ejecuta el siguiente código:

```
n = 40
while n <= 51:
    if n%2 == 0:
        print(f'{n} ', end=' ')
    n += 1
```

    Se muestra en pantalla…

   a) Nada, porque el código es incorrecto
   b) 40 42 44
   c) 40 42 44 46 48 50
   d) 42 44 46 48 50

Universidad
Rey Juan Carlos

II. PROBLEMAS

P1 (3 puntos)

Programa en Python un subprograma que, a partir de un número entero entre 2 y 10, muestre por pantalla un patrón triangular repetido 2 veces como el siguiente (ejemplo para n=4):

```
*
* *
* * *
* * * *
* * *
* *
*
*
* *
* * *
* * * *
* * *
* *
*
```

```
def dibujar_patron(lado):
    """ int -> None
        OBJ: muestra por pantalla un patrón triangular
       PRE: 2 <= lado <= 10
    """
    for i in range(2): # las dos partes del patrón triangular
        for j in range(1, lado + 1): # parte ascendente
            for k in range(j): print('*', end=' ')
            print()
        for j in range(lado-1,0,-1): # parte descendente
            for k in range(j): print('*', end=' ')
            print()

# probador
patron(4)
```

P2 (3 puntos)

Escribe un programa modularizado que calcule el número de potencias de 2 de una secuencia de números que se encuentran en una lista. Por ejemplo, la secuencia 5,7,8,10,11,13,15,16 tiene dos potencias de 2: 8 (2^3) y 16(2^4).

```
def es_potencia2(n):
    """ int -> bool
        OBJ: determina si n es potencia de 2 o no
    """
    while n%2 == 0: n/=2
    return n==1

def num_potencias2(secuencia):
    """ list -> int
        OBJ: calcula el número de potencias de 2 de una serie
    """
    contador_potencias = 0
    for numero in secuencia:
        if es_potencia2(numero):  contador_potencias+= 1
    return contador_potencias

# probador
print(num_potencias2([5,7,8,10,11,13,15,16]))
```

Universidad
Rey Juan Carlos

P3 (3 puntos)

Observa la siguiente serie numérica:

1 + 1/2 + 2/3 + 3/5 + 5/8 + 8/13 + 13/21 + 21/34 + 34/55 + 55/89 + …

como se ve, en cada término el numerador es la suma de los 2 numeradores anteriores y lo mismo para el denominador. Esto pasa solo a partir del tercer término, dado que los dos primeros términos son fijos: 1 y 1/2.

Escribe un programa que calcule y muestre por pantalla términos de la serie hasta que la diferencia entre el valor del término actual (T_n) y el calculado la vez anterior (T_{n-1}) sea menor que un valor epsilon predeterminado.

```
EPSILON = 1e-7

# Imprime los dos primeros términos fijos de la serie
print('1 + 1/2', end=' ')

# Prepara los valores para comenzar a iterar
penultimo_numerador = penultimo_denominador = 1
numerador_anterior = 1
denominador_anterior = 2
numerador = numerador_anterior + penultimo_numerador
denominador = denominador_anterior + penultimo_denominador

# Itera hasta que el término a añadir es suficientemente pequeño (< eps)
while abs(numerador/denominador - numerador_anterior/denominador_anterior) > EPSILON:
    print(f'+ {numerador}/{denominador}', end=' ')
    penultimo_numerador = numerador_anterior
    numerador_anterior = numerador
    penultimo_denominador = denominador_anterior
    denominador_anterior = denominador
    numerador = numerador_anterior + penultimo_numerador
    denominador = denominador_anterior + penultimo_denominador
```

Ejercicios extra

(La máxima puntuación de este apartado es de 2 puntos)

PE1 (1 punto)

Haz un subprograma que recibe una lista con 5 mediciones de temperatura y un umbral y dice cuántas hay por encima de dicho umbral (1punto)

```
def num_mediciones_superan_umbral(mediciones, umbral):
    """ list, float -> int
        OBJ: determina cuántas mediciones superan un umbral
        PRE: mediciones es una lista de float
    """
    sobre_umbral = 0
    for medicion in mediciones:
        if medicion > umbral: sobre_umbral+= 1
    return sobre_umbral

# probador
print(num_mediciones_superan_umbral([5,3,4,78,65,12],10))
```

Universidad
Rey Juan Carlos

PE2 (1 punto)

Escribe un programa que calcule el número de pares e impares de una serie de números que se introducen por teclado (1 punto)

```python
def cuenta_pares_impares(serie):
    """ list -> int, int
        OBJ: calcula el número de pares e impares de una serie
        PRE: serie es una lista de enteros
    """
    pares = impares = 0
    for numero in serie:
        if numero % 2 == 0: pares += 1
        else: impares += 1
    return pares, impares

# probador
print(cuenta_pares_impares([5,3,4,13,8,15,12]))
```

PE3 (1,5 puntos)

"Un elefante se balanceaba" es una canción tradicional con una letra muy sencilla:

1 elefante se balanceaba sobre la tela de una araña.
Como veía que no se caía fue a llamar a otro elefante.

El mismo verso se repite, cada vez con un elefante más, si bien no de la misma manera pues al principio es un elefante (singular) y después son más de uno:

30 elefantes se balanceaban sobre la tela de una araña.
Como veían que no se caían fueron a llamar a otro elefante.

La canción se completa cuando los intérpretes llegan a una cierta cota de aburrimiento. Tu tarea es escribir un programa modularizado en Python capaz de generar todos los versos de la canción y, puesto que en un programa es difícil predecir con qué número de elefantes se aburrirán y pararán los intérpretes, supondremos que dicho número se genera aleatoriamente cada vez que se ejecuta el programa.

```python
import random

def escribe_verso(num_elefantes):
    """ int-> string
        OBJ: Retorna el verso de la cancion infantile con num_elefantes
        PRE: num_elefantes > 0
    """
    if num_elefantes==1:
      return "1 elefante se balanceaba sobre la tela de una araña.
              Como veía que no se caía fue a llamar a otro elefante.\n"
    else:
      return f"{num_elefantes} elefantes se balanceaban sobre la tela
             de una araña. Como veían que no se caían fueron a llamar
             a otro elefante.\n"

# probador
versos = random.randint(2,20)
for elefantes in range(1,versos): print(escribe_verso(elefantes))
```

Universidad
Rey Juan Carlos

Programación I – Grado en Inteligencia Artificial	Prueba escrita 2 (22 / 01 / 2025)
SOLUCIONES	

PARTE TEÓRICO-PRÁCTICA (30%)

[Todas las preguntas valen lo mismo: 0,60 puntos]

1. ¿Qué valor retornaría la llamada `Fibo_Christmas(5,3)` según el siguiente código?:

```
def Fibo_Christmas(x,y):
   if (y >= x):   result = 1
   else: result = Fibo_Christmas(x-2,y-1) + 2*Fibo_Christmas(x-1,y+1)
   return result
```

SOLUCION: 5

2. Papá Noel tiene que ordenar miles de listas de direcciones para optimizar sus entregas, porque repartir cientos de miles de regalos en una noche no es cosa fácil. Rodolfo el Reno le ha recomendado utilizar alguno de los algoritmos de ordenación vistos en la asignatura, y le pone como ejemplo la secuencia numérica de una calle en la que deben entregar regalos en los números [43, 4, 8, 16, 11, 40, 9, 14, 18, 5]. Rodolfo le muestra a Santa las direcciones de esta calle tras varias pasadas del algoritmo, si bien no se ha completado la ordenación. El estado actual de la lista es:

   4, 5, 8, 9, 11, 40, 16, 14, 18, 43

   ¿Con qué algoritmo ha sido ordenada? ¿Cuántas iteraciones se han llevado a cabo?

SOLUCION: Método de selección tras 4 o 5 iteraciones

3. Indica qué mensaje saldría por pantalla tras ejecutar el siguiente código:

```
def reno1(x):
    print('Rudolph',end='-')
    return x

def reno2(x):
    if x: print('Dasher',end='-')
    return not x

if(reno1(False) or reno2(True)):  print(1,end='-')
if(reno1(True) and reno2(False)): print(2,end='-')
```

SOLUCIÓN: Rudolph-Dasher-Rudolph-2-

4. Como sabes, un conjunto permite mantener una colección de elementos no repetidos, pero es importante recordar que los conjuntos en Python no pueden contener objetos mutables. Sabiendo esto, ayuda a Papá Noel completando el código siguiente para que, haciendo uso de un conjunto, se muestren en la pantalla de su trineo de reparto todas las coordenadas de las casas en que hay que depositar un regalo, pero claro, sin repeticiones, porque algunos niños han escrito varias cartas en un intento de recibir más regalos de los que les corresponderían. En resumen, si alguna coordenada aparece más de una vez en `address_list` solo se deberá mostrar una vez.

```
address_list = [ [34.01,45.09], [11.31,5.69], [21.03,37.54],
                 [64.22,-13.04], [11.31,5.69], [-56.61,74.82]
               ]
unique_addresses = set()
for address in _______________ :
    x,y = ____________________________
    unique_addresses.add(_____________________)
for _________________ in unique_addresses: print(coord)
```

SOLUCIÓN:

```
for address in address_list :
    x,y = address
    unique_addresses.add((x,y))
for coord in unique_addresses: print(coord)
```

5. Mientras Papá Noel preparaba todo para el día 25, algunos alumnos de la URJC participaban en el Advent of Code 2024, uno de cuyos retos decía: “Los Elfos usan un protocolo de comunicación formado por paquetes de datos de 4 caracteres. El final de un mensaje lo marca un paquete en que ninguno de los caracteres se repite. ¿puedes codificar una función que detecte si un paquete de 4 caracteres indica el fin de un mensaje?” Tus intentos de codificar algo así empiezan a ser desesperados, así que recurres a una amiga, la cual ha sido tan gentil de pasarte su código como pista para que completes el tuyo con la condición de que no uses ni (a) `break` ni (b) un `return` desde dentro de un `for`. Te ha marcado lo que deberías cambiar, ¿Puedes hacerlo?:

```
def all_different(paquete):
    """ list-> bool  """
    for i in range(1,len(paquete)):
        if paquete[i] in paquete[:i] + paquete[i+1:]:
           return False  # <---- cambiar
    return True
```

SOLUCIÓN:

```
def all_different(paquete):
    """ list -> bool """
    i = 1
    are_different = True
    while are_different and i<len(paquete):
        if paquete[i] in paquete[:i] + paquete[i+1:]:
           are_different = False
        i += 1
    return are_different
```

Universidad
Rey Juan Carlos

PARTE PRÁCTICA (70%)

EJERCICIO 1 [1,5 puntos]

Escribe un subprograma recursivo que permita calcular valores de los polinomios de Hermite, $H_n(x)$, siendo x un valor numérico (real) y n el orden del polinomio. Los polinomios de Hermite cumplen las siguientes propiedades:

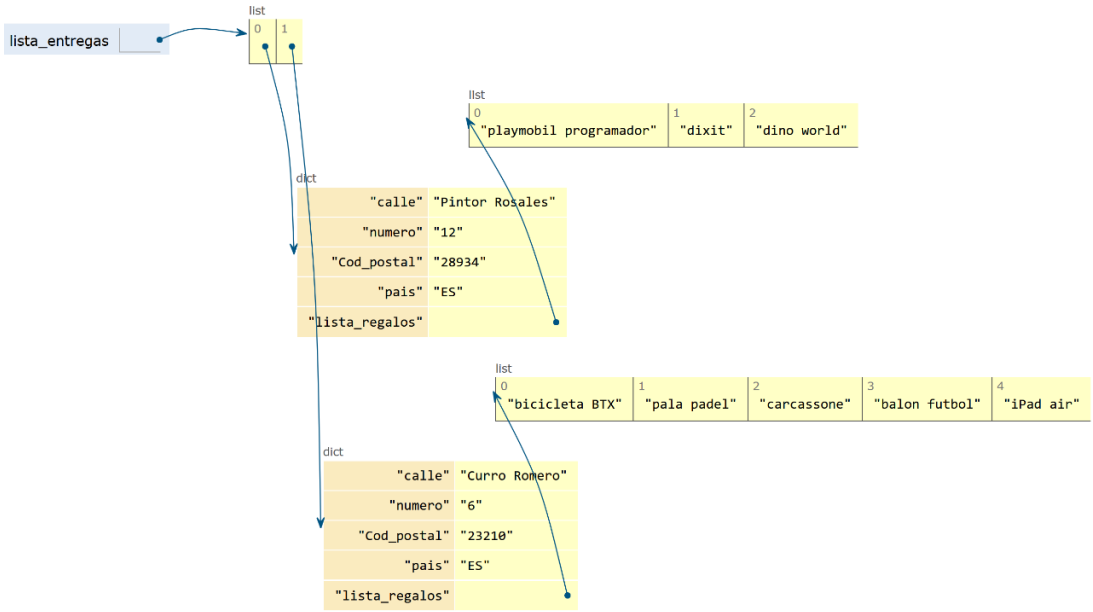
$H_0(x) = 1$, para todo x
$H_1(x) = 2 \cdot x$
$H_n(x) = 2 \cdot x \cdot H_{n-1}(x) - 2 \cdot (n-1) \cdot H_{n-2}(x)$, para todo $n > 1$

```
def H(x, n):
    """ float, int -> float
        PRE: n>=0
    """
    if n == 0: return 1.0
    elif n== 1: return 2*x
    elif n > 1: return 2*x*H(x,n-1) - 2*(n-1)*H(x,n-2)
```

EJERCICIO 2 [5,5 puntos]

Mientras tú te divertías con el Advent Of Code 2024 y comías mazapán, Papá Noel estaba estresado pensando en el mejor modo de repartir regalos a los más pequeños. Para una tarea así, se han de planificar bien las cosas pues cuando debes visitar 5.500 hogares cada segundo sin equivocarte, la planificación es lo más importante. Afortunadamente contaba con un ejército de elfos que permiten que todo, finalmente, salga bien (o al menos así ha sido hasta hoy). Este año, Papá Noel utilizó una lista donde se han ido añadiendo todos los domicilios que debía visitar –calle, número, código postal, país y la lista de regalos a entregar– a medida que han ido llegando las cartas de los niños. Un ejemplo parcial de lista sería:

Universidad
Rey Juan Carlos



Un elfo ya veterano ha sugerido a Papá Noel que ordene la lista por código de país, para así agilizar las entregas. Han pensado que este año, en Europa occidental, lo mejor sería visitar primero Portugal, luego España, después Francia, y luego Italia, Austria, Suiza, Alemania, Países Bajos y finalmente Bélgica, por este orden. El problema es que ordenar una lista tan inmensa a mano es una tarea fuera del alcance de una persona de la edad de Papá Noel (1.679 años). Pero bueno, no hay problema porque esto es algo relativamente sencillo en Python.

1) Tu primera tarea consiste pues en ayudar a ordenar la lista siguiendo la ruta recomendada. Para ello, implementa un subprograma que, a partir de la lista y de una ruta en formato texto (ejemplo: "PT-ES-FR-IT-AT-CH-DE-NL-BE") adapte el método de inserción directa para ordenar ascendentemente la lista de entregas. **[1 p.]**

```
def insercion(A):
    """ lista -> None """
    for i in range(1, len(A)):
        clave = A[i]
        j = i-1
        while j >=0 and clave < A[j]:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = clave
```

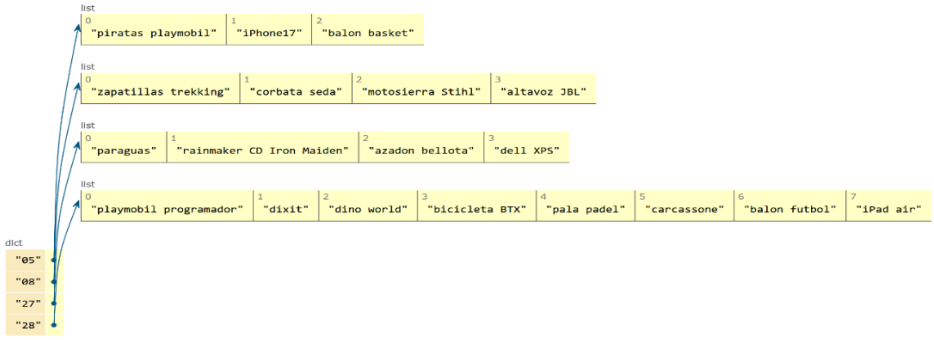
```
def ordenar(entregas,ruta):
    """ list, str -> None
        OBJ: Ordena la lista de entregas por país, utilizando la ruta proporcionada.
             Método de inserción.
    """
    for i in range (1, len(entregas)):
        clave = entregas[i]
        j = i -1
        while j >=0 and ruta.index(clave['pais']) < ruta.index(entregas[j]['pais']):
            entregas[j+1] = entregas[j]
            j -= 1
       entregas[j+1] = clave
```

2) Tu segunda tarea será facilitar el reparto de regalos en España. Para ello necesitamos, a partir de la lista ordenada resultado del apartado 1, extraer una sublista con todas las entregas para nuestro país España **[1 p., 1.5 si lo haces recursivamente]**

```python
def extraer_sublista_es(lista_entregas):
    """ list -> list
        OBJ: Extrae una sublista con todas las entregas para España
    """
    lista_entregas_es = []
    for entrega in lista_entregas:
        if entrega['pais'] == 'ES':
            lista_entregas_es.append(entrega)
    return lista_entregas_es
```

```python
def extraer_sublista_es_rec(lista_entregas):
    """ list -> list
        OBJ: Extrae recursivamente una sublista con todas las entregas para España
    """
    if lista_entregas == []: return []
    elif lista_entregas[0]['pais'] == 'ES':
          return [lista_entregas[0]] + extraer_sublista_es_rec(lista_entregas[1:])
    else: return extraer_sublista_es_rec(lista_entregas[1:])
```

3) Una vez centrados en España, y para facilitar la asignación de repartos locales a cada elfo, necesitamos un diccionario donde se listen por código de provincia (los dos primeros dígitos del código postal) los regalos a entregar en esa zona, añadiendo los regalos únicamente, sin la dirección, con el objeto de planificar el número de elfos a asignar a cada provincia. El resultado será algo como lo que se muestra a continuación (solo se ven 4 provincias, pero obviamente se clasificarían todas) **[1,5 puntos]**:




```python
def clasificar_por_provincia(lista_entregas):
    """ list -> dict
        OBJ: Genera un diccionario donde se listan por código de provincia
             los regalos a entregar en esa zona.
    """
    entregas_por_provincia = {}
    for entrega in lista_entregas:
        cod_provincia = entrega['cod_postal'][:2]
        if cod_provincia not in entregas_por_provincia:
            entregas_por_provincia[cod_provincia] = entrega['lista_regalos']
        else:
            entregas_por_provincia[cod_provincia] += entrega['lista_regalos']
    return entregas_por_provincia
```

4) A partir del diccionario anterior y de un diccionario con la información por comunidades autónomas, Papá Noel necesita saber cuáles son las comunidades con un mayor y menor número de regalos por niño. La información sobre comunidades es una lista que se descarga del Ministerio del Interior y que incluye el nombre de la comunidad, una lista con los códigos de sus provincias (2 dígitos) y el número de niños en la comunidad como la siguiente: **[2 puntos]**

```python
info_autonomias = {
    'Catalunya': { 'num_ninnos': 1001000, 'provincias': ['08','25','17','43' ]},
    'Madrid': { 'num_ninnos': 643000, 'provincias': ['28'] },
    'Extremadura': { 'num_ninnos': 260000, 'provincias': ['10','06'] },
    'Euskadi': { 'num_ninnos': 260000, 'provincias': ['48','20','01'] }
    # etc. }
```

```python
menor = [None, 0]
mayor = [None, 0]
for comunidad in info_autonomias:
    total_actual = 0
    for provincia in info_autonomias[comunidad]['provincias']:
        if provincia in entregas_por_provincia:
            total_actual += len(entregas_por_provincia[provincia])
    media_actual = total_actual / info_autonomias[comunidad]['num_ninnos']
    if mayor[0] == None or media_actual > mayor[1]:
        mayor[0] = comunidad
        mayor[1] = media_actual
    if menor[0] == None or media_actual < menor[1]:
        menor[0] = comunidad
        menor[1] = media_actual
# probador
print('Mayor: ', mayor[0], ' - ', round(mayor[1],2),' regalos por niño/a')
print('Menor: ', menor[0], ' - ', round(menor[1],2),' regalos por niño/a')
```

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

Grado en Inteligencia Artificial - PROGRAMACIÓN I

PRUEBA DE EVALUACIÓN EXTRAORDINARIA - JUNIO 2025

DURACIÓN: 90 minutos

©2025 - Algunos derechos reservados.
Isaac Lozano Osorio (isaac.lozano@urjc.es)
Salvador Sánchez Alonso (salvador.sanches@urjc.es)
Esta obra está bajo una licencia
Creative Commons "Atribución Compartir Igual 4.0 Internacional".
Material docente en abierto de la Universidad Rey Juan Carlos

PARTE 1

1A– TEORÍA

E1 - Dada la siguiente variable: `palabra ='Examen Programación Junio'`

Escribir la salida por pantalla de cada una de las siguientes líneas **[1 punto]**:

```
1. print(palabra [7:-2])
2. print(palabra [2:5], '2')
3. print(palabra [:7], '3')
4. print(palabra [9:], '4')
5. print(palabra [-5:] + '5')
6. print(palabra [:-5], 6)
```

```
1.

2.

3.

4.

5.

6.
```

E2 - Responde a las siguientes cuestiones indicando si son verdaderas o falsas y razona brevemente (1 línea por pregunta) el porqué. **[1 punto]**

a) A las variables booleanas no se les puede asignar una expresión, solo una constante.

b) Una sentencia `if` tiene siempre obligatoriamente que ir acompañada de un `elif` o de un `else` para tener sentido completo.

c) Supongamos que un subprograma `f()` existen unas variables locales llamadas "c" y "d" propias. Al existir dichas variables, las variables "c" y "d" pertenecientes al programa principal pierden visibilidad en nuestro subprograma `f()` y se convierten en inaccesibles.

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

E3 - Completa la siguiente función para que retorne una lista con números desde a hasta b (ambas incluidas, en cada posición habrá un elemento igual al anterior menos 1), teniendo en cuenta que a es mayor que b y devuelva el número de elementos cargados en el vector. **Ejemplo:** si ejecutamos `rellena_descendente(v,87,78)`, obtendremos en las primeras posiciones de v: `87 86 85 84 83 82 81 80 79 78`, y también devolverá el número `10` como número de elementos generados. **[1 punto]**

```
def rellena_descendente(v,a,b):

    num = _____

    i = _______

    while ( num > _________ ):

        v[i] = num
        i _________

        num ____________

    return ___________
rellena_descendente(v,87,78)
```

E4 - El siguiente programa debería escribir las potencias de 5, desde 5 hasta 5^n, aunque por algún motivo no cumple con lo esperado. Indica qué errores tiene, de qué tipo (ejecución / compilación) y cómo habría que modificar el código para que funcione correctamente. **[1 punto]**

```
1. def potenciasDe5 ():
2.     result = 0
3.     i = 0
4.     for i in range (n):
5.         print ("5^", i, " = ", result)
6.         result *= 5
7.         i += 1
8.     return
9. potenciasDe5(5)
```

Línea 1:

Línea 2:

Línea 3:

Línea 4:

Línea 5:

Línea 6:

Línea 7:

Línea 8:

Línea 9:

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

E5 – Indica cuál es la salida del siguiente código **[1 punto]**:

```
def piirustus (n):
    for i in range(n):
        for j in range(n):
            if i == j or i == 0 or i == n - 1: print('*', end='' )
            else: print( '.', end='')
        print()

piirustus(5)
```

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

Parte 1B – PRÁCTICA

P1. Dada la siguiente serie:

$$S = 1/0! - 2/1! + 3/2! - 4/3! + 5/4! - \ldots$$

a) Escribe un subprograma en Python que calcule el valor de la serie S para un valor determinado de n. **[1,5 puntos]**

b) Como puede comprobarse introduciendo algunos valores, la serie S converge a cero. Escribe un programa de prueba que permita calcular el primer valor de "n" para el cual el valor de S en valor absoluto es menor que 10^{-15}. **[1 punto]**

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

P2 - Un número de dos cifras se dice “trino” si existe una diferencia de 3 unidades entre las cifras contiguas que lo componen. Así, por ejemplo, 147 lo es, ya que 4-1= 3 y 7-4 = 3. También lo es 30, y por supuesto 96, 63 y 963. Sin embargo, 32, 98 o 145 no son números “trinos”.

a) Escribir una función que determine cuándo un número es “trino” **[1,5 puntos]**

b) Una vez completado lo anterior, escribe un programa que lea por teclado números de tres cifras hasta que se introduzca un número trino. En ese momento, deberá indicarlo por pantalla y comenzará a calcular la suma de los 3 números impares que siguen a dicho número, mostrando el resultado desglosado por pantalla. Por ejemplo, si se introduce el número 147 el programa produce la siguiente salida **[1 punto]**

```
Introduzca un número: 147
Acaba de introducir un número trino. Su valor es: 147
Suma de los 3 impares siguientes:  149 + 151 + 153 = 453
```

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

Grado en Inteligencia Artificial - PROGRAMACIÓN I

PRUEBA DE EVALUACIÓN EXTRAORDINARIA - JUNIO 2025

DURACIÓN: 90 minutos

PARTE 2

PARTE 2A– TEORÍA

E1 – Indica qué hace el siguiente código y cuál sería la salida por pantalla **[1,5 puntos]**:

```
def khachapuri(n):
    if n <= 9:       r = 0
    else:
        if n%10==0:  r = 1 + khachapuri(n/10)
        else:        r = 0
    return r

print (khachapuri (123400000))
```

E2 - Responde a las siguientes cuestiones indicando si son verdaderas o falsas y razona brevemente (1 línea por pregunta) el porqué. **[2 puntos]**

a) La recursividad es una técnica de programación que permite que un procedimiento se invoque a sí mismo, pero no que una función lo haga.

b) En Python, un docstring y un comentario es lo mismo.

c) En Python, las listas, los diccionarios y las tuplas son tipos de datos mutables.

d) En Python, cuando se pasa una lista como parámetro a una función, cualquier modificación realizada dentro de la función afectará a la lista original.

E3 - Corrige el siguiente algoritmo en caso de que existan errores. **[1,5 puntos]**

```
def invierte_letras(cadena,n):
    ''' string, int -> None
        OBJ: Lee N letras y las escribe en orden inverso (recursivo)
        PRE: 0 <= n < len(cadena)
    '''
    if (n = 0):
        print (cadena[1], end="")
    else:
        invierte_letras(cadena, n-1)
         print (cadena[n], end="")
```

Universidad Rey Juan Carlos

Apellidos: _____

Nombre: _____

PARTE 2B – PRÁCTICA

P1. Supón una lista de la forma [(string, float)] que representa los nombres y sueldos de trabajadores de una empresa, donde el sueldo es un valor entre 0 y 2000€. Codifica un procedimiento que tome una lista como esa e incremente a cada trabador su sueldo hasta el mínimo interprofesional si este no llega al salario mínimo (actualmente 1.184€), indicando por pantalla a qué trabajador(es) se le(s) ha sido corregido su sueldo. Posteriormente codifica y aplica una función que incremente el sueldo un 3% a todos los trabajadores que están por debajo de la media indicando a qué trabajadores se les ha incrementado el sueldo. **[1,5 puntos]**

P2. Escribe el código de la siguiente función recursiva: **[1,5 puntos]**

```
def sustituye_rec(lista_palabras, buscada, reemplazo):
   """ lista, string, string -> lista
       OBJ: Recibe una lista de palabras, busca todas las apariciones de la
            palabra a buscar y la sustituye por la palabra a reemplazar. Devuelve
            la lista de palabras con las sustituciones realizadas.
   """
```

Un ejemplo de funcionamiento podría ser:

```
Lista inicial: ['avión', 'tren', 'coche', 'cohete', 'bicicleta', 'coche']
Sustituir: coche por moto
Lista final: ['avión', 'tren', 'moto', 'cohete', 'bicicleta', 'moto']
```

NOTA IMPORTANTE: La implementación debe ser recursiva. Cualquier implementación no recursiva o que use replace no será valorada.

Apellidos: _____

Nombre: _____

P3. Dada una serie de elementos de información heterogéneos de tipo "Contacto" (campos: nombre, apellido, numero de tfno. móvil, edad y sexo) almacenados en una lista, programa una función que muestre por pantalla la información de los primeros 5 contactos que sean mujeres entre 18 y 20 años (**IMPORTANTE**: listará 5 como máximo, si no hay 5 listará únicamente los que haya). **[2 puntos]**

Software

Programación 1

Convocatoria de Asignaturas en Abierto 2025-26
Material docente en abierto de la Universidad Rey Juan Carlos
Grado en Inteligencia Artificial

Universidad Rey Juan Carlos

Profesores:
Isaac Lozano Osorio y Salvador Sánchez Alonso

Repositorio:
https://github.com/isaaclo97/docencia-p1

SWHID:
swh:1:dir:938f20641c4384205fa44ec9e934b4b6dcaa9542

Depositado en BURJC Digital:
https://hdl.handle.net/10115/111477

©2025 Autores Isaac Lozano Osorio y Salvador Sánchez Alonso. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en: https://creativecommons.org/licenses/by-sa/4.0/

Programación 1 — Ejercicios para DomJudge

> **Convocatoria de Asignaturas en Abierto 2025-26** Material docente en abierto de la Universidad Rey Juan Carlos — Grado en Inteligencia Artificial https://hdl.handle.net/10115/111477

Material docente para la asignatura **Programación 1** (P1). Contiene los ejercicios de programación utilizados en las prácticas de la asignatura, organizados en cuadernos temáticos e importables directamente en DomJudge.

Objetivo docente

Cada ejercicio incluye el enunciado en PDF, casos de prueba de ejemplo y secretos, y una solución de referencia en Python. El alumnado resuelve los problemas en DomJudge, que evalúa automáticamente la corrección de las soluciones enviadas.

Estructura del repositorio

Los ejercicios están organizados en 7 cuadernos temáticos dentro de `ejercicios/`. Cada ejercicio sigue la estructura estándar de DomJudge:

```
ejercicios/
├── Cuaderno 1/       – Introducción: entrada/salida y operaciones básicas
│   ├── 1. Area/
│   │   ├── problem.yaml             – nombre del problema
│   │   ├── problem.pdf              – enunciado
│   │   ├── data/sample/             – casos de prueba visibles
│   │   ├── data/secret/             – casos de prueba ocultos
│   │   └── submissions/accepted/    – solución de referencia en Python
│   └── ...
├── Cuaderno 2/         – Operaciones con reales y conversiones
├── Cuaderno 3/         – Condicionales y bucles
├── Cuaderno 4/         – Listas
├── Cuaderno 5/         – Cadenas
├── Cuaderno 6/         – Recursividad
└── Cuaderno 7/         – Ordenación
```

Listado de ejercicios

Cuaderno	Ejercicio	Problema
1	1	Área del cuadrado
1	2	Impuestos
1	3	Hora
1	5	Rectángulo
1	6	Temperatura

Cuaderno	Ejercicio	Problema
1	8	Tipo de empresa
1	9	Título de empresa v2
2	1	Fuerza
2	2	Barns
2	3	TIEA
2	4	Circunferencia
2	5	Pinta
2	6	Kelvin
2	7	3D
3	1	Orden
3	2	Bisiesto
3	3	Meses
3	4	Escuelas
3	5	Reales (suma, mínimo, máximo)
3	6	Romanos
3	7	Supermercado
3	8	Potencias
3	9	Ceros finales
3	10	Número cute
4	1	Codificación
4	2	Impares (lista)
4	3	Eliminar elemento
4	4	Duplicados
5	1	Contar palabras
5	2	Separar
6	5	Binario
6	7	Edades
6	13	Pascal
6	15	Fibonacci
7	2	Ordenar

Cómo importar en DomJudge

Cada directorio de ejercicio puede empaquetarse como ZIP e importarse directamente en DomJudge desde la interfaz de administración (`Problems > Import`). El ZIP debe contener en su raíz los ficheros `problem.yaml`, `problem.pdf` y los directorios `data/` y `submissions/`.

Requisitos

- Python 3 para ejecutar las soluciones de referencia.
- Una instancia de DomJudge para usar los ejercicios como juez automático.

Licencia

Copyright (C) 2025 Salvador Sánchez, Isaac Lozano Osorio. Distribuido bajo la licencia MIT.