

**GRADO EN INGENIERÍA INFORMÁTICA**

Módulo de Formación Básica

---

# FUNDAMENTOS DE PROGRAMACIÓN

---

**Dr. D. Pedro Gomis Román**





Este material es de uso exclusivo para los alumnos de la VIU. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la VIU, sin autorización expresa de la misma.

**Edita**

Universidad Internacional de Valencia

**Depósito Legal**

En proceso

Grado en

# Ingeniería Informática

---

**Fundamentos de Programación**  
Módulo de Formación Básica  
6ECTS

---

**Dr. D. Pedro Gomis Román**



## Índice

PRESENTACIÓN .....	7
TEMA 1. INTRODUCCIÓN. VISIÓN GENERAL DEL MUNDO DE LA INFORMÁTICA.....	11
1.1. Computadores. Aspectos históricos .....	12
1.2. Computadores. Estructura funcional .....	14
1.3. Codificación de información en computadores .....	16
1.3.1. Sistema de numeración decimal .....	16
1.3.2. Sistema de numeración binario .....	17
1.3.3. Sistemas de numeración potencias de 2: octal y hexadecimal .....	18
1.3.4. Representación de texto .....	20
1.4. Algoritmos, programas y lenguajes .....	21
1.4.1. Lenguajes compilados e interpretados .....	23
1.4.2. Lenguaje Python .....	23
1.4.3. Ejemplos clásicos de algoritmos .....	26
TEMA 2. TIPOS DE DATOS SIMPLES, EXPRESIONES Y OPERACIONES ELEMENTALES.....	29
2.1. Tipos de datos .....	29
2.1.1. Datos simples .....	30
2.1.2. Datos compuestos o estructurados .....	34
2.2. Variables y acción de asignación .....	35
2.3. Expresiones y sentencias .....	39
2.4. Operadores .....	40
2.4.1. Operadores aritméticos .....	40
2.4.2. Operadores lógicos .....	41
2.4.3. Operadores relacionales .....	43
2.4.4. Orden de las operaciones .....	45
2.5. Acciones elementales .....	46
2.5.1. Lectura de datos .....	47
2.5.2. Conversión entre tipos de datos .....	47
2.5.3. Escritura de datos .....	48
2.5.4. Comentarios .....	50
TEMA 3. ESTRUCTURAS ALGORÍTMICAS .....	51
3.1. Estructura secuencial .....	52
3.2.1. Estructura alternativa simple o condicional .....	54
3.2.2. Estructura alternativa doble (if-else) .....	55
3.2.3. Estructura alternativa múltiple o anidada .....	56
3.3. Estructuras iterativas .....	58
3.3.1. Secuencia de datos .....	58
3.3.2. Esquemas iterativos .....	59
3.3.3. Estructura iterativa while (mientras) .....	59
3.3.4. Estructura iterativa for (desde - hasta, para - en) .....	64
3.3.5. Bucles para efectuar sumatorias .....	68
3.3.6. Sentencias break y continue .....	69

TEMA 4. PROGRAMACIÓN MODULAR. FUNCIONES, PROCEDIMIENTOS Y PARÁMETROS.....	71
4.1. Uso de funciones. Funciones internas y de módulos.....	72
4.2. Funciones y procedimientos.....	76
4.3. Diseño de funciones.....	77
4.3.1. Paso de parámetros entre el programa y las funciones.....	79
4.3.2. Funciones productivas y funciones nulas (procedimientos).....	80
4.3.3. Valores de argumentos por omisión (default).....	81
4.3.4. Argumentos de palabra clave (keyword arguments).....	81
4.4. Recursividad.....	82
4.5. Módulos: integrar funciones en una biblioteca.....	84
TEMA 5. TIPOS DE DATOS ESTRUCTURADOS: HOMOGÉNEOS Y HETEROGÉNEOS, DINÁMICOS Y ESTÁTICOS.....	87
5.1. Datos estructurados inmutables (estáticos).....	88
5.1.1. Cadena de caracteres (string).....	88
5.1.2. Tuplas.....	92
5.1.3. Conjuntos congelados (FrozenSet).....	97
5.2. Datos estructurados mutables (dinámicos).....	98
5.2.1. Listas.....	98
5.2.2. Vectores y matrices con NumPy.....	106
5.2.3. Conjuntos (Set).....	110
5.2.4. Diccionarios.....	110
5.3. Funciones nulas (procedimientos) y paso de parámetros por referencia.....	112
5.4. Ficheros ( <b>files</b> ).....	114
GLOSARIO.....	121
ENLACES DE INTERÉS.....	127
BIBLIOGRAFÍA.....	129
Referencias bibliográficas.....	129
Bibliografía recomendada.....	130
ANEXO A.....	131

## Leyenda



### Glosario

Términos cuya definición correspondiente está en el apartado "Glosario".

## Presentación

Como asignatura inicial para aprender herramientas de programación de computadores, **Fundamentos de Programación** contribuye a que los alumnos obtengan competencias en el uso de ordenadores y su programación para resolver problemas propios de la ingeniería. El ingeniero en informática o de otras ramas de la ingeniería seguramente, durante su vida profesional o académica, debe desarrollar programas o supervisar equipos de programadores. Muy probablemente la mayoría de los lectores han usado programas complejos como sistemas operativos (tipo Windows, Linux, Android) y programas de aplicaciones, como procesadores de texto, hojas electrónicas de cálculo y videojuegos. A modo de ejemplo, la tecnología actual de equipos médicos como los desfibriladores implantables o marcapasos incluyen programas que toman decisiones para que el dispositivo actúe automáticamente en caso de una anomalía cardíaca. Pero problemas aparentemente más simples, como hallar las raíces de un polinomio de orden 2 o superior o calcular la media y desviación estándar de miles de datos experimentales, son ideales para ser resueltos con técnicas de programación.

Este curso se enfoca en aprender a **programar** más que en especializarse en el lenguaje de programación vehicular. El arte de la buena programación se basa tanto en aprender las expresiones para hacer cálculos, estructuras algorítmicas, manejo de estructuras de datos, etc. (la sintaxis y semántica del lenguaje de programación), como en el trabajo práctico de resolver ejercicios y

problemas. El objetivo es adquirir habilidades para hacer que el computador, a través de programas, resuelva los problemas que el usuario se enfrenta.

**¿Qué lenguaje de programación se usará** como vehículo para aprender programación en este curso? Nuestra primera experiencia en el uso de programas informáticos en Ingeniería a comienzos de la década de 1970 fue con FORTRAN (FORMula TRANslation), lenguaje desarrollado por IBM y de amplio uso en los comienzos de la era de los computadores digitales en la solución de problemas de cálculo en ingeniería. Luego, muchas Escuelas de Ingeniería utilizaron el lenguaje C (o C++) por su buena compatibilidad con las instrucciones próximas al manejo del microprocesador. Otras Escuelas han usado como lenguajes de introducción a la programación el Java, que es un lenguaje apreciado por estar orientado a objetos y de amplio uso en programar *applets* (pequeñas aplicaciones que se ejecutan en navegadores web), o el Pascal por su facilidad en el aprendizaje a programar. Siguiendo la idea del Pascal y que la prioridad cuando se inician estudios de Ingeniería es el aprendizaje de los procesos de la programación más que especializarse en el lenguaje, muchos cursos de informática o introducción a la programación han usado un lenguaje algorítmico (genérico) o pseudocódigo que permita luego su *traducción* al lenguaje en que se implementará en el computador.

Desde hace varios años, sin embargo, muchas Escuelas o Facultades de ingeniería alrededor del mundo han introducido el lenguaje Python para enseñar a programar. **Python será el lenguaje utilizado en esta asignatura** por ser un lenguaje de alto nivel que simplifica la sintaxis para escribir programas a partir de unas instrucciones en lenguaje natural o en pseudocódigo. Python es un lenguaje muy eficiente y ayuda el proceso de aprendizaje de programación por su claridad en escribir las estructuras algorítmicas, por disponer de una gran cantidad de módulos o librerías de funciones y porque se programa en un entorno amigable. Así que podemos enfocarnos en aprender a programar sin perdernos tanto en la sintaxis del lenguaje. Además, Python soporta estructuras complejas de datos y programación orientada a objetos, lo cual es fácilmente aplicable al aprendizaje posterior de Java o C++.

El temario de este manual estará organizado de la siguiente forma:

- Tema 1: Introducción. Visión general del mundo de la informática.
- Tema 2: Tipos de datos simples, expresiones y operaciones elementales.
- Tema 3: Estructuras Algorítmicas.
- Tema 4: Programación modular. Funciones, procedimientos y parámetros.
- Tema 5: Tipos de dato estructurados: homogéneos y heterogéneos, dinámicos y estáticos.

Las actividades prácticas se realizarán en el entorno de programación Python siguiendo el guion de prácticas o instrucciones proporcionadas por el profesor.



Este material didáctico y el resto de recursos de aprendizaje disponibles servirán para que el alumno logre los **resultados de aprendizaje** previstos:

- Explicar el funcionamiento de un ordenador enfatizando la necesidad de desarrollo de programas (*software*) por parte del programador.
- Usar las estructuras algorítmicas (de control) básicas: secuencial, condicional e iterativa.
- Resolver problemas aplicando una metodología de diseño modular (utilización de subprogramas o funciones). Analizar el concepto de recursividad en programación.
- Manejar correctamente los mecanismos de comunicación entre programas y funciones, así como las distintas formas de paso de parámetros y devolución de resultados.



## Tema 1.

### Introducción. Visión general del mundo de la informática

La **informática** es el área del conocimiento que estudia todo lo que hace referencia a la obtención y procesamiento de información por medios automatizados, a partir de unos datos determinados. El origen del término proviene del francés *informatique* o *información automática*, es decir, procesamiento automático de la información por medio de computadoras. En España, principalmente, se ha utilizado el nombre de informática aunque en América se utiliza el término computación. En lengua inglesa prevalecen los términos *computing*, *computation* y *computer science*. Según la Real Academia Española (2014), informática es el “Conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de computadoras”. De forma equivalente, en inglés, *computer science* se encarga del estudio de los procesos de información.

Aunque las computadoras analógicas fueron utilizadas el siglo pasado para realizar cálculos matemáticos con bastante éxito, desde mediados de siglo se impuso el uso de tecnología digital en el diseño de computadores. Los términos **computador** o **computadora** (del latín: *computare*, calcular) popularmente usados son válidos ambos, según la Real Academia de la Lengua (2014). En España desde los años 1960 se difundió más el nombre de **ordenador**, probablemente influenciado del francés *ordinateur* (del latín: *ordinator*), aunque actualmente se usan de manera equivalente ‘ordenador’ o ‘computador’.

## 1.1. Computadores. Aspectos históricos

Desde sus orígenes, el ser humano desarrolló herramientas para aumentar su fuerza física (palancas, mecanismos, ruedas, etc.) y también aumentar sus habilidades intelectuales. El lenguaje como forma de comunicación y la escritura para transmitir en el tiempo y distancia la información fueron las primeras formas de mejorar sus habilidades intelectuales. Luego, el cálculo para construir edificaciones, evaluar los movimientos de los cuerpos celestes, etc., fue un nuevo hito en el desarrollo del ser humano. Así, el hombre buscó herramientas que puedan incrementar su poder de cálculo.

- El ábaco, que se estima apareció sobre el 2400 A.C. en Babilonia, fue usado por muchas civilizaciones (India, China, Egipto, Grecia) para realizar cálculos aritméticos. El nombre actual proviene del griego abax: tabla o rectángulo.
- Wilhelm Schickard en el siglo XVII construye la primera máquina de calcular (con imperfecciones) basada en engranajes de relojería. Blaise Pascal, matemático francés, construye a los pocos años una segunda máquina mecánica capaz de sumar, restar y hacer cálculos aritméticos. Se le llamó "máquina aritmética" o "pascalina" y es considerada un antepasado de los computadores analógicos.
- En 1671 Gottfried Wilhelm Leibniz (von) Leibniz, matemático alemán, extendió las ideas de Pascal y desarrolló una máquina (*Stepped Reckoner* o máquina de Leibniz) que incorpora la posibilidad de multiplicar, dividir y sacar raíz cuadrada. Leibniz también describió el sistema numérico binario para realizar cálculos, antecesor de la lógica binaria y álgebra de Boole, en que se basa la computación actual.
- En 1820 Charles X. Thomas patenta y produce una máquina de calcular mecánica con éxito comercial usada hasta principios del siglo XX en agencias gubernamentales, bancos y empresas aseguradoras, que precede las cajas registradoras de las tiendas que se usaron hasta comienzos de 1970.
- Durante el siglo XIX se desarrollaron cintas de papel y tarjetas perforadas portadoras de información que se usaron en pianos automáticos y, en el siglo XX, en control numérico de máquinas. Luego, en las primeras épocas de la programación de computadores digitales, se usaron como medio de introducir el código y datos de programa al equipo.
- Las reglas de cálculo (*side rule*, *slipstick*) fueron otro de los hitos de herramientas de cálculo, consideradas como computadores analógicos. Las reglas de cálculo (hoy en día piezas de museo) pueden realizar operaciones aritméticas, logaritmos y funciones trigonométricas.
- Computadores analógicos electrónicos. A diferencia de los cálculos con tecnología digital, donde la información está discretizada, las técnicas de cálculo analógicas utilizan información o variables continuas en el tiempo y amplitud. Durante el siglo XX, hasta la década de 1970, tuvo amplio uso las computadoras analógicas electrónicas. Éstas pueden realizar cálculo

aritmético o resolver ecuaciones diferenciales, a través de circuitos electrónicos que realizan operaciones aritméticas, derivadas e integrales sobre señales de voltaje continuas en tiempo y amplitud (analógicas).

Estos computadores y máquinas de cálculo descritos fueron, de hecho, **computadores de programas fijos**, resuelven un número limitado de problemas para lo que fueron diseñados. La idea de funcionamiento de los computadores modernos es atribuida a Charles Babbage (entre 1822 y 1837), ingeniero británico, quien desarrolló un computador mecánico que incluía los conceptos de unidad aritmética, control de flujo, ramificaciones condicionales, bucles y una memoria integrada. A Babbage se le considera uno de los padres de la computación.

Alan Turing (1912 - 1954), matemático inglés (figura 1), fue quizás el primer teórico de la informática y propuso formalismos científicos a los procesos algorítmicos. En 1936 Turing desarrolló un hipotético computador (un modelo matemático) que contenía una memoria ilimitada sobre una cinta y un conjunto de pocas instrucciones para grabar y leer ceros y unos sobre la cinta y moverse sobre ella. La tesis de Turing afirma que si una función es calculable, la máquina de Turing puede programarse para hacerlo. Alan Turing sentó las bases de los computadores de propósitos generales, con **programas almacenados**, en una publicación en 1936. En 1945 Turing trabajó en el desarrollo de un **computador digital con programas almacenados** usando tecnología electrónica, publicando ese año sus resultados.

Simultáneamente, John von Neumann (físico húngaro-americano), que conocía el trabajo teórico de Turing, propuso en 1945 un diseño de arquitectura de computador que, aunque similar en ciertas ideas al diseño de Turing, se dio a conocer como la "arquitectura de von Neumann" (Coperland, 2008). El primer computador moderno se considera el Manchester Ferranti Mark 1, usando la arquitectura actual atribuida a von Newman y en cuyo desarrollo de programas participó Turing. Este **computador con programas almacenados** ejecutó su primer código programado en 1949. Así, en la arquitectura de von Newman y los ordenadores actuales, se almacenan las instrucciones y datos de la misma manera, utilizando la unidad de memoria principal (figura 2). La unidad de procesamiento central (CPU) contiene una unidad aritmética y lógica (ALU), registros del procesador, una unidad de control que contiene un registro de instrucciones, un contador de programa y mecanismos de entrada y salida.



Figura 1. Memorial a Alan Turing Memorial, Sackville Park, Manchester, Inglaterra. Fuente: elaboración propia.

## 1.2. Computadores. Estructura funcional

La informática se encarga de los **procesos** de información y las máquinas electrónicas que hoy en día se encargan de estos procesos son los computadores digitales (ordenadores)<sup>1</sup>. Cuando un computador realiza los diferentes pasos de un trabajo dado, se dice que está realizando un **proceso**. Los **procesos informáticos** comprenden básicamente de tres fases: (i) la entrada de datos, (ii) el procesado donde se tratan los datos mediante una secuencia de acciones preestablecidas por determinado programa, y (iii) la salida de datos.

Así, podemos definir un computador como una máquina que:

1. Acepta entradas. Las entradas pueden ser introducidas por un humano a través de un teclado, pueden ser recibidas de una red, o ser proporcionadas de forma automática por medio de sensores conectados a la computadora.

---

1 Usaremos indistintamente los nombres "computador" u "ordenador".

2. Ejecuta un procedimiento automático, es decir, un procedimiento en el que cada paso puede ser ejecutado sin ninguna acción manual de un humano.
3. Produce salidas. Las salidas pueden ser datos que se muestran a una persona, pero también podría ser cualquier cosa que afecta al mundo fuera del ordenador, tales como señales eléctricas que controlan el funcionamiento de un dispositivo.

Funcionalmente, la estructura de los ordenadores actuales, basada en la de von Newman, permite procesar la información, realizar operaciones matemáticas y lógicas utilizando su gran capacidad de memoria gobernadas por un programa informático. En el modelo de arquitectura básica de una computadora descrito en la figura 2 se observa la memoria como el lugar en que se almacena la información. Luego de ser ésta codificada en forma binaria, la **memoria** almacena instrucciones de programas y datos de cualquier tipo. La memoria suele incluir una parte de solo lectura de las instrucciones básicas (**ROM**, de *read-only-memory*), que permanecen grabadas y otra memoria de acceso aleatorio para escribir y leer datos e instrucciones de programas (**RAM**, de *random access memory*), que se borra al cerrar el programa o el ordenador. Adicionalmente se dispone una memoria externa al procesador central en medios magnéticos como discos duros o electrónicos de estado sólido como las memorias flash de los teléfonos móviles modernos. La unidad aritmética y lógica (ALU) realiza operaciones matemáticas elementales y lógicas y produce nuevos datos. La unidad de control se encarga de trasladar la información a la ALU y devolver los resultados a la memoria. Estas unidades de control y ALU en conjunto forman la unidad de procesamiento central (conocida por sus siglas en inglés CPU, *Central Processing Unit*).

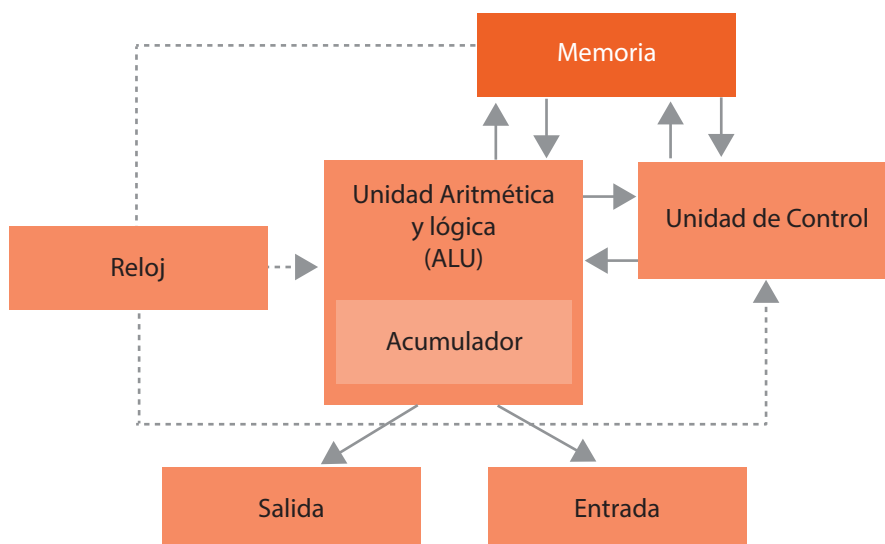


Figura 2. Diagrama de la arquitectura de von Newman. Fuente: elaboración propia.

La información que se almacena en la memoria y se procesa en la computadora está codificada en forma binaria (1's y 0's; cierto o falso). La **unidad primaria** de la información en computación es el **bit** (combinación de **binary digit**). Una pregunta binaria tendrá solo dos respuestas: cierto o falso. En

informática se representan usualmente estos dos estados como 1 y 0. Electrónicamente, los ceros y unos se realizan mediante cambios de estado de corte a saturación de los componentes electrónicos básicos o cambios de un voltaje a otro (por ejemplo, 0V a 5V). Este tema será tratado con más detalles en asignaturas de Fundamentos o Tecnologías de Computación. En la siguiente sección se presentan diferentes formas de codificación numérica y el estándar en informática.

## 1.3. Codificación de información en computadores

Todo tipo de información que se quiera procesar en una computadora al final es representada de forma numérica para luego ser codificada en forma binaria usando bits como unidad básica. La forma numérica conocida de números naturales, enteros, reales y complejos, así como el texto de cualquier lenguaje (castellano, inglés, etc.) se representa finalmente en formato binario para ser procesada por el CPU del computador. Resumimos a continuación los tipos más usados de representación numérica y caracteres alfabéticos y otros signos de puntuación.

### 1.3.1. Sistema de numeración decimal

El sistema universal de representación numérica es el decimal, de base 10, que utiliza los números indio-arábigos introducidos a Europa de la India a través de la región persa y del actual mundo árabe, y probablemente influenciados por un sistema en base 10 de China. Se considera al matemático persa Al-Khwarizmi o Al-Juarismi el primero en utilizar la numeración india decimal para los métodos de cálculo, en sus publicaciones del siglo IX, como el famoso tratado de álgebra.

Base	10
Conjunto de símbolos	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

*Ejemplos.*

El número entero 2435 (2 es el dígito más significativo y 5 el menos significativo), su valor es:

$$2 \cdot 10^3 + 4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0 = 2435$$

El número real 1428,35, su valor es:

$$1 \cdot 10^3 + 4 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2} = 1428,35$$

En general, cualquier número N puede representarse mediante un polinomio de potencias de una base cualquiera b (decimal:  $b = 10$ ; binario:  $b = 2$ , etc.); es decir



$$N_b = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \dots + a_1 \cdot b^1 + a_0 \cdot b^0 + a_{-1} \cdot b^{-1} + \dots + a_{-m} \cdot b^{-m} \quad (1)$$

donde,  $b$  = base del sistema de numeración,  $a_k$  es un número perteneciente a ese sistema de numeración y  $a_k$  pertenece al conjunto  $[0, b-1]$

Los números enteros, es decir, los números naturales más sus negativos y el cero son valores discretos y se pueden ordenar. Sin embargo los valores continuos, como los números reales, en general no se pueden representar con exactitud y solo se pueden aproximar en las computadoras digitales. Aunque si se incrementa suficientemente el número de bits para representarlos, la aproximación es tan buena como se quiera. La codificación de los reales sigue unos estándares en los diversos lenguajes que serán comentados en otra sección.

### 1.3.2 Sistema de numeración binario

El sistema de numeración en base 2 (binario) es el empleado en la tecnología actual de las computadoras digitales.

Base	2
Conjunto de símbolos	{0, 1}

*Ejemplo.*

El número 11011 en base 2 (binaria) tiene el valor 27 en base decimal, donde el primer "1" es el bit más significativo, y el último el menos significativo:

Número:  $11011_{10}$

Valor en base 10:  $1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 2 + 1 = 27_{10}$

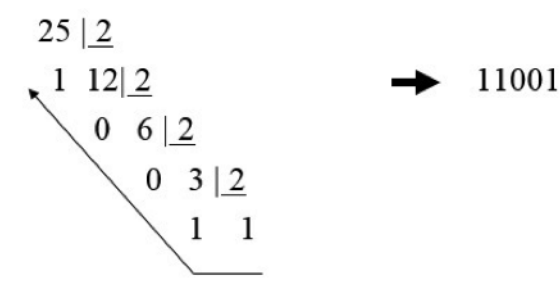
La conversión de binario a decimal o, simplemente, el valor equivalente del número binario en decimal se calcula siguiendo la ecuación (1) donde el número a convertir es de base 2. A continuación un ejemplo con parte fraccionaria en base dos.

Número:  $110,01_{10}$

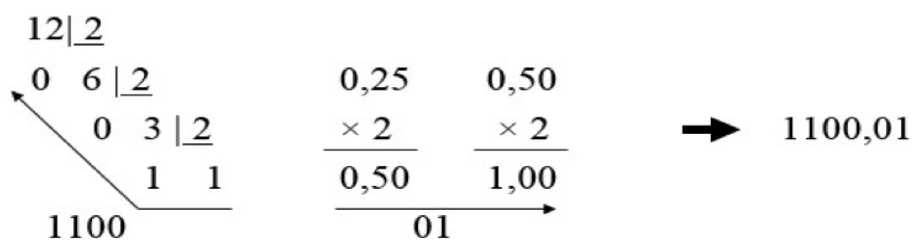
Valor en base 10:  $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 4 + 2 + 0,25 = 6,25_{10}$

La conversión de base decimal a binaria se realiza del siguiente modo. Se divide el número  $N$  entre 2 y el cociente se vuelve dividir entre 2 y así sucesivamente. El último cociente y los restos forman el número binario. Ejemplo:

Si queremos representar el número decimal 25 en binario:



El número decimal 12,25 se representa en binario:



La información que se almacena y se procesa en las computadoras es agrupada en palabras de 8 bits que forman las unidades de información usadas hoy en día. Estas unidades de 8 bits se denominan **byte**<sup>2</sup>. Con un byte podemos representar 256 números diferentes ( $2^8$ ), usualmente los números naturales en el rango [0, 255]. Por ejemplo, el byte 10010111 representa el número decimal 151:

Binario	1	0	0	1	0	1	1	1
Peso	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
Decimal	128			16		4	2	1
								<b>Suma = 151</b>

### 1.3.3. Sistemas de numeración potencias de 2: octal y hexadecimal

La representación numérica en base 8 (octal) y 16 (hexadecimal) tiene mucha utilidad en informática, sobre todo en el uso de instrucciones cuando se programa directamente sobre la máquina, por la facilidad de agrupar números binarios. Al ser potencias de 2, el sistema octal agrupa 3 bits de información ( $2^3$ ) y el hexadecimal 4 bits ( $2^4$ ).

<sup>2</sup> El vocablo inglés *byte* es aceptado por la Real Academia de la Lengua (RAE). El término *octeto* para palabras de 8 bits también fue usado en España en el comienzo de la era de la informática.

### Sistema de numeración octal

Base	8
Conjunto de símbolos	{0, 1, 2, 3, 4, 5, 6, 7}

*Ejemplo.*

El número 325 en base 8 ( $325_8$ ) equivale a 213 en base 10. En base 2 se puede expresar cada dígito en formato binario. El 3 equivale a 011, 2 equivale a 010 y 5 a 101, así queda 011010101 en base 2. Su valor en base 10 corresponde al 213:

Valor en base 10:  $3 \cdot 8^2 + 2 \cdot 8^1 + 5 \cdot 8^0 = 192 + 16 + 5 = 213^{10}$

Valor en base 2: 011 010 101

### Sistema de numeración hexadecimal

Base	16
Conjunto de símbolos	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

donde A tiene un peso de 10, B uno de 11, hasta F que tiene un peso de 15.

*Ejemplo.*

El número  $35A_{16}$  equivale en decimal al 858. Fácilmente se puede convertir a formato binario reemplazando cada dígito hexadecimal por los 4 bits equivalentes del conjunto de símbolos (ver tabla 1).

Valor en base 10:  $3 \cdot 16^2 + 5 \cdot 16^1 + A \cdot 16^0 = 768 + 80 + 10 = 858^{10}$

Valor en base 2: 0011 0101 1010

La tabla 1 muestra las equivalencias entre los sistemas de numeración descritos.

Decimal	Binario	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Tabla 1. Equivalencia entre sistemas de numeración.

### 1.3.4. Representación de texto

La representación de texto en informática se realiza con una codificación numérica de palabras de 7 u 8 bits, siguiendo comúnmente el código ASCII (*American Standard Code for Information Interchange*). El código ASCII original utiliza 7 bits para representar 128 ( $2^7$ ) caracteres alfanuméricos y de puntuación. En este estándar, por ejemplo, la letra 'e' es el binario 1100101 (número decimal 101, o el hexadecimal 65). La 'E' (mayúscula) se representa por el binario 0101100, equivalente decimal 69 (hexadecimal 45).

En el anexo A se muestra la tabla de caracteres ASCII y una extensión ASCII que utiliza un bit adicional para definir 128 caracteres adicionales y así poder representar otros signos como, por ejemplo, vocales acentuadas, la ñ, ç, etc. La 'é' se representa por el binario 10000010, equivalente al decimal 130 (hexadecimal 82). IBM desarrolló a principio de los años 1980 este código extendido a 8 bits (se llamó código de página 347) y lo incorporó en los primeros sistemas operativos de los computadores personales (PC, del inglés *personal computer*) de la época; el PC-DOS y MS-DOS. Actualmente en los

sistemas operativos Windows aún se puede generar estos caracteres extendidos. Sobre cualquier editor de texto mantenga pulsada la tecla ALT y simultáneamente tecleé un número del teclado numérico<sup>3</sup> (*keypad*). Por ejemplo ALT + 164 escribe la ñ o ALT+241 escribe el símbolo matemático  $\pm$ . Pero hay que resaltar también que existen muchas otras variantes de código de página ASCII extendida como el ISO-8859-1 (Latin1), ISO-8859-15 (Latin9, que incorpora el símbolo del euro), etc.

Aun así, con 8 bits, la codificación de las tablas ASCII y ASCII extendida no es suficiente para representar la gran cantidad de caracteres y símbolos de todos los alfabetos del mundo. Con el uso extendido de internet y el diseño de páginas webs se han desarrollado formatos de codificación de caracteres más universales como el Unicode<sup>4</sup>, estándar de codificación de caracteres para múltiples lenguas, así como el UTF-8 (8-bit *Unicode Transformation Format*) o el UTF-16.

Adicionalmente, información más compleja que se procesa en una computadora, como por ejemplo imágenes, grabaciones de audio y video, se codifica también con secuencias de bits, en distintos formatos. Por ejemplo, imágenes en formatos BMP, JPEG, GIF, TIFF, PNG o DICOM en imágenes médicas; audio en WAV, MP3, OGG; video en AVI, MOV, MP4, WMV, FLV.

## 1.4. Algoritmos, programas y lenguajes

Se entiende por **algoritmo** la sucesión ordenada de acciones elementales que se han de realizar para conseguir la solución correcta de un problema en un tiempo finito, a partir de unos datos dados. Codificar un algoritmo es escribirlo en un lenguaje que el computador pueda entender y ejecutar, es decir realizar un **programa**. Los algoritmos son los métodos generales para la resolución de problemas o realización de tareas. Se puede relacionar con un guion o receta a seguir para esta realización. El nombre algoritmo proviene de las primeras traducciones al latín de las obras del matemático Al-Juarismi, cuando usó el sistema decimal proveniente de la India en sus métodos de cálculo: "*dixit Algorismi...*" ("dijo Algoritmo..."). El uso del nombre actual de algoritmo proviene de este vocablo, aunque el concepto de algoritmo es mucho más antiguo. Hay evidencias que lo antiguos Babilonios utilizaban una serie de operaciones iterativas para aproximar la raíz cuadrada de un número. También Euclides en el siglo III a.C describió en su obra *Elementos* una metodología para hallar el mayor divisor común a dos números enteros mayores que cero.

Un **programa** informático es una secuencia de instrucciones que detalla cómo realizar un cálculo. El cálculo puede ser matemático, como hallar las raíces de un polinomio o el valor medio de una secuencia de datos, pero también pueden ser cálculos para tareas de todo tipo, como buscar y reemplazar texto en un documento, procesar una señal física o extraer contornos de una imagen debidamente digitalizada.

---

3 Nuestra experiencia con los primeros computadores personales que llegaban a Latinoamérica a finales de los años 1980 o escribiendo en español en un país de habla inglesa, teníamos los teclados con la configuración en inglés. Las vocales acentuadas y la ñ era normal crearlas de esta manera. Por ejemplo, á con ALT+160.

4 <http://unicode-table.com/es/> [https://en.wikipedia.org/wiki/List\\_of\\_Unicode\\_characters](https://en.wikipedia.org/wiki/List_of_Unicode_characters)

Para escribir la secuencia de instrucciones de un programa se requiere un **lenguaje de programación**. El lenguaje básico que entiende el CPU de un ordenador es el conjunto de instrucciones que se usa para programar el microprocesador ( $\mu$ P), llamado normalmente **lenguaje de máquina**. Como a nivel del  $\mu$ P se programa en formato binario (0's y 1's) cada  $\mu$ P comercial tiene asociado un conjunto de instrucciones mnemotécnicas que traducen la instrucción a formato hexadecimal y binario, llamado lenguaje ensamblador o *assembler*. Actualmente los  $\mu$ P de las computadoras manejan las instrucciones y los datos en formato de 32 bits (4 bytes) o 64 bits (8 bytes) y cada fabricante (Intel, AMD, IBM Power PC, etc.) tiene su juego de instrucciones para su  $\mu$ P. Por ejemplo, los  $\mu$ P de la familia Intel han usado el lenguaje ensamblador x86, aunque recientemente con el uso de  $\mu$ P de 64 bits de Intel y AMD se utiliza el lenguaje x86-64 o x64 (Lomont, 2012) para desarrollar sistemas operativos o lenguajes de programación como el Python, C++ o Java. Los lenguajes ensamblador o de máquina se conocen también como **lenguajes de bajo nivel** pues están al nivel del  $\mu$ P o la máquina, versus los de **alto nivel** que están cercanos al usuario.

Los **sistemas operativos** (OS, del inglés *operating systems*) de las computadoras, *tablets*, o teléfonos móviles inteligentes (*smartphones*) son también programas más complejos que sirven para relacionar la CPU (el  $\mu$ P) con otros programas y el usuario. Están diseñados sobre los  $\mu$ P de los dispositivos, utilizando sus respectivos lenguajes de máquina o ensambladores. De esta forma, el usuario puede desconocer qué tipo de  $\mu$ P tiene su PC o teléfono móvil, sin embargo interacciona en un ambiente de trabajo igual al trabajar sobre el sistema operativo Windows, OS X de Mac o Linux en su PC o el sistema operativo Android o iOS en su teléfono móvil o celular inteligente.

Los **lenguajes de programación de alto nivel** son, entonces, los que están más cercanos al lenguaje natural de un humano y para éste no depende en la práctica sobre qué máquina u OS está trabajando. Por ejemplo, el Python, MATLAB, C++, Java son lenguajes de alto nivel.

Así como los **lenguajes naturales** (español, inglés, etc.), los lenguajes de alto nivel tienen su vocabulario, sintaxis y semántica. El **vocabulario** será todos los elementos propios del lenguaje como los signos de operaciones aritméticas, lógicas y de relaciones, las palabras propias del lenguaje para definir estructuras algorítmicas como el if, for, while, etc.

La **sintaxis** define la forma como se pueden combinar los elementos del lenguaje. Por ejemplo,  $x = 3 + 2$  es una sintaxis correcta, pero  $x = 3 \ 2 +$  no lo es. La **semántica** se refiere a que expresiones sintácticamente correctas tengan un resultado correcto. Por ejemplo,  $x = 7/5$  es correcto sintácticamente y semánticamente pero en  $x = 7/'hola'$  aunque la sintaxis permita asignar a x el resultado de una división, semánticamente sería incorrecto dividir números entre texto. También una semántica correcta incluye que el programa no se cuelgue (quede en un bucle infinito) o no dé errores lógicos o de ejecución. Este último sería, por ejemplo, dividir entre una variable que el programa le asigne el valor 0.

### 1.4.1. Lenguajes compilados e interpretados

Los programas que escribimos en lenguaje de alto nivel se hacen en un editor de texto y se llaman **código fuente**. Al ejecutar nuestro código (corremos el programa, *run the program*), dependiendo del tipo de lenguaje, se traducen a lenguaje de máquina mediante dos formas diferentes: a través de un compilador o de un intérprete.

Un **compilador** lee el código fuente y lo traduce completamente al lenguaje de máquina creando un programa ejecutable. Tiene la ventaja que el programa ejecutable se puede usar las veces que se quiera sin tener que traducir el programa original. Además se ejecuta más rápido. Los lenguajes de programación clásicos como el Fortran, Pascal o C++ son compilados.

Un **intérprete**, sin embargo, es un programa que lee una a una las instrucciones (sentencias) del código fuente y las traducen a las apropiadas del lenguaje de máquina. Va traduciendo y ejecutando las sentencias del programa a medida que las encuentra. Esto los hace más flexibles y fáciles de depurar pero se pierde velocidad respecto a los programas compilados. Entre los lenguajes de programación interpretados destacan el Python, Matlab, y Ruby.

### 1.4.2. Lenguaje Python

El **Python** es un lenguaje de programación de alto nivel, considerado de muy alto nivel y de propósitos generales, que es ejecutado por un intérprete. Es muy compacto y ayuda a los que se inician a la programación a escribir estructuras algorítmicas con un código claro. El lenguaje fue creado por Guido van Rossum a finales de la década de 1980 en Holanda y se publicó la primera versión a comienzos de la década de 1990. Actualmente es el lenguaje de programación más usado en los cursos de introducción a la programación o ciencias de la computación en las principales Universidades de Estados Unidos (Guo, 2014). Python es también uno de los lenguajes más utilizados por los desarrolladores de programas a nivel mundial, de acuerdo al ranking IEEE Spectrum 2016 (Diakopoulos & Cass, 2016).

*Python Software Foundation* (<https://www.python.org/psf/>) administra el lenguaje Python y su licencia de código abierto. Los detalles de instalación y uso del ambiente de desarrollo de programas de Python escapan al objetivo de este manual y serán presentados en los guiones de apoyo de las prácticas de laboratorio. Sin embargo, describimos un resumen de sus características para empezar a trabajar en este lenguaje. La página web para descargar el programa es <https://www.python.org/> y en la opción de descargas (Downloads) se puede encontrar el sistema operativo (Microsoft Windows, Mac OS X, Linux, etc.) y tipo de máquina disponible para ser instalado. Es de acceso libre. Usaremos la versión 3.x en lugar de la 2.x. El ambiente de desarrollo integrado de programas (IDLE<sup>5</sup>, de *integrated DeveLopment Enviroment*) de Python permite escribir códigos fuentes y luego ser ejecutados por el intérprete. Las instrucciones se ejecutan sobre una ventana o consola llamada **Shell**. Sobre el Shell se puede también ejecutar en modo línea de comando o modo inmediato una expresión

---

5 El nombre de Python está influenciado por la comedia británica de la pandilla de amigos de Monty Python, y parece que el uso de IDLE en lugar de las siglas IDE se debe al personaje Eric Idle.

como una calculadora. Al abrir el IDLE (Python), aparecerá una ventana similar a la que se muestra a continuación y, siguiendo la tradición en programación, escribiremos nuestro primer código de introducción al mundo de la programación con un saludo:

```
Python 3.6.0a3 (v3.6.0a3:f3edf13dc339, Jul 11 2016, 21:40:24)
Type "copyright", "credits" or "license()" for more information.
>>> print('Hola mundo!')
Hola mundo!
```

En el ejemplo de la figura 3 se observa el Shell de la versión 3.6.0 y el cálculo de 2 expresiones aritméticas.

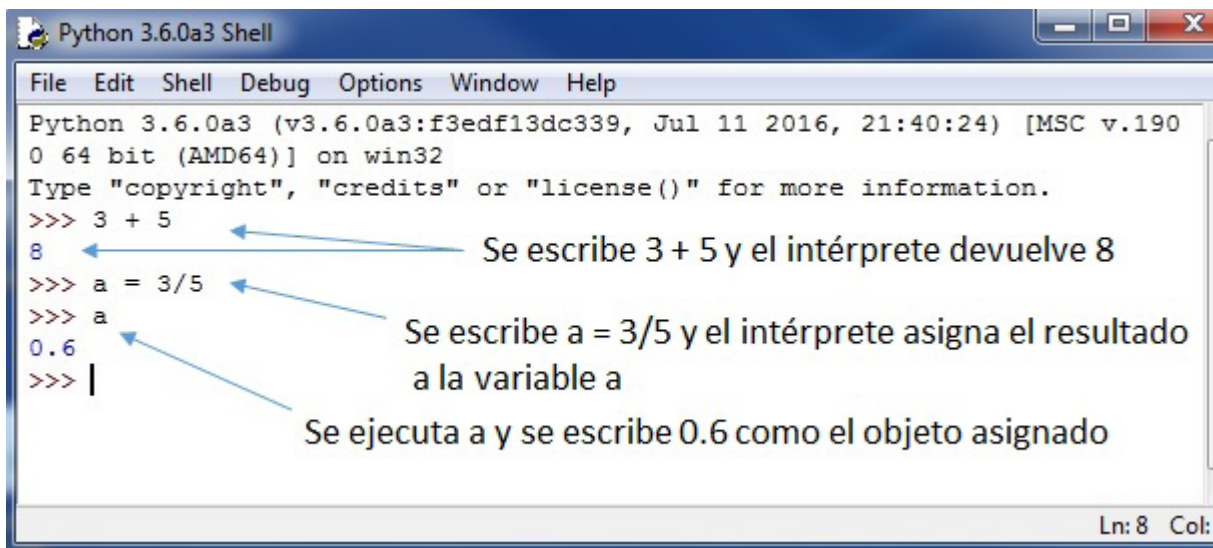


Figura 3. Imagen del Shell de Python y cálculo de expresiones aritméticas con el intérprete.

El símbolo `>>>` es el llamado **prompt** (en español entrada o apunte, aunque en informática se suele usar el nombre inglés) de Python.

En la figura 4 se muestra el código fuente de un programa simple que calcula el área de un triángulo rectángulo de catetos  $c_1$  y  $c_2$ , por ejemplo,  $c_1 = 3$  y otro  $c_2 = 4$ . Se escribe el código en un editor, creado de la pestaña (tab) *File* → *New File* y se salva desde esa ventana con *Save as* (por ejemplo: *areaTriang.py*). Luego se ejecuta desde la pestaña *Run* → *Run module F5* (o simplemente teclear *F5*). El resultado aparecerá en el Shell de Python.



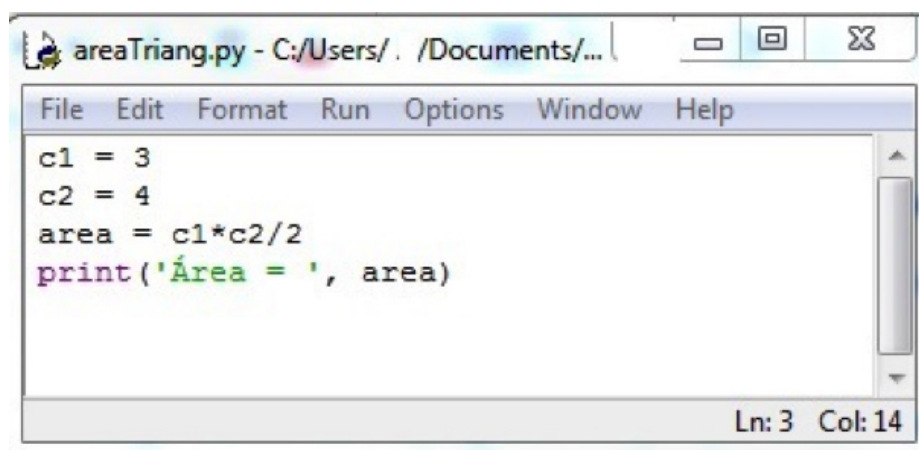


Figura 4. Imagen del editor de Python con ejemplo de código fuente del programa *areaTriang.py*

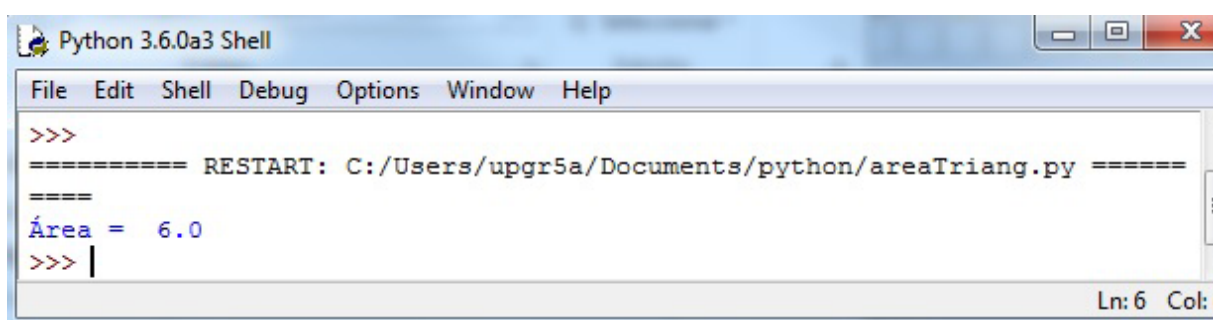


Figura 5. Imagen del Shell de Python mostrando resultado del programa *areaTriang.py*

La función `print` de Python escribe el resultado en el Shell del lenguaje de programación (ver figura 5), no se debe confundir con la acción *Print Windows* de la pestaña *File* que envía todo el contenido a la impresora.

Como se muestra en la figura 6, el programa *áreaTriang.py* sería más eficiente si puede generalizarse para calcular el área de cualquier triángulo rectángulo. La función *input* permite leer del Shell un dato del teclado, que será en formato texto, luego la función *int* convierte este dato de texto a número entero.

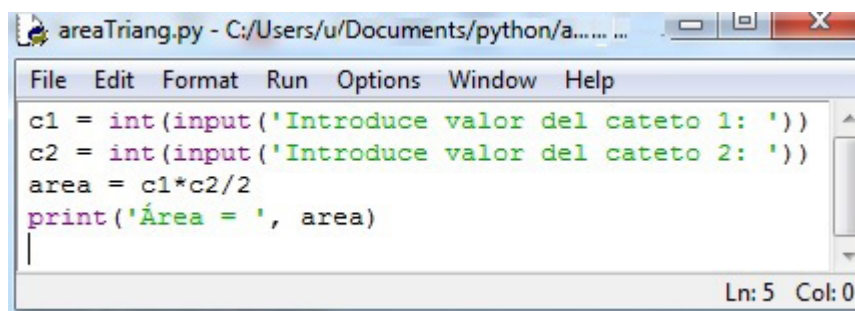
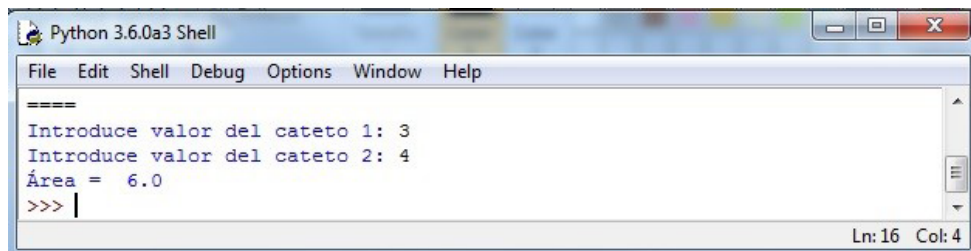


Figura 6. Imagen del editor de Python del programa *areaTriang.py* modificado pidiendo datos de entrada

El Python Shell muestra el mensaje de texto de la función *input* y espera que el usuario introduzca el número, en este caso 3, que al ser introducido del teclado se interpreta como un carácter ASCII. La función *int* lo convierte a un número tipo entero (ver figura 7).



```
====
Introduce valor del cateto 1: 3
Introduce valor del cateto 2: 4
Área = 6.0
>>> |
```

Figura 7. Imagen del Shell de Python mostrando resultado del programa *areaTriang.py*

### 1.4.3. Ejemplos clásicos de algoritmos

Entre los primeros métodos para resolver problemas con una secuencia de instrucciones y toma de decisiones lógicas para repetir los cálculos de forma iterativa o de bucles se encuentra el algoritmo de Euclides para hallar el máximo común divisor de 2 números naturales y la aproximación a la raíz cuadrada de cualquier número real.

El método propuesto por Euclides (siglo III a.C.) para hallar el máximo común divisor (MCD) de dos números naturales, conocido como algoritmo de Euclides, propone que

“dados 2 números naturales, *a* y *b*, comprobar primero si son iguales. Si lo son entonces *a* es el MCD. Si no son iguales, entonces probar si *a* es mayor que *b* y si lo es restar a *a* el valor de *b*, pero si *a* es menor que *b*, entonces restar a *b* el valor de *a*. Repetir el procedimiento con los nuevos valores de *a* y *b*”.

En los próximos temas realizaremos un programa en Python para este cálculo y una versión recursiva más eficiente.

Herón de Alejandría (destacado matemático e inventor griego, siglo I d.C.) describió un método para aproximar la raíz cuadrada de un número (se cree que el método ya era conocido en la antigua Babilonia) por aproximaciones sucesivas. Es decir, la raíz cuadrada de *x* es un número *y* que satisface  $y^2 = x$ . Isaac Newton (siglo XVII) propuso posteriormente un método general para hallar las raíces de una función no lineal,  $f(x) = 0$ .

El método de Herón para hallar la raíz cuadrada de  $x$  consiste en:

1. Se propone un número candidato cualquiera  $g$  (podemos, por ejemplo, comenzar con  $x/2$ ).
2. Si  $g * g = x$ , o lo suficientemente aproximado de acuerdo a la precisión que queramos (usaremos a partir de ahora el operador  $*$  para la multiplicación en programas informáticos, en lugar de  $\times$  o  $\cdot$ , usados en matemáticas) paramos y  $g$  es la solución.
3. Si no, se realiza una nueva búsqueda de la raíz de  $x$  promediando el valor de  $g$  y  $x/g$ . Es decir,  $(g + x/g)/2$ . Este valor lo llamamos también  $g$ .
4. Volvemos al paso 2 hasta conseguir que  $g * g$  sea igual o lo suficiente aproximado a  $x$ .

Si el valor inicial propuesto de  $g$  lo denominamos  $g_0$  y los siguientes valores  $g_n$ , donde  $n = 1, 2, \dots$ , entonces las propuestas de raíz de  $x$ ,  $g_n$ , en la iteración  $n$  es función de las propuestas previas  $g_{n-1}$  y el número  $x$ :

$$g_n = \frac{g_{n-1} + \frac{x}{g_{n-1}}}{2}$$

Estos algoritmos clásicos los realizaremos en lenguaje de programación Python al introducir las estructuras algorítmicas condicionales y alternativas.



## Tema 2.

# Tipos de datos simples, expresiones y operaciones elementales

En este tema presentaremos algunos conceptos y elementos básicos usados en los lenguajes de programación en general, aunque se especificarán las características particulares del lenguaje Python. Trataremos los siguientes conceptos: tipos de datos simples y compuestos, variables, identificadores, acciones elementales como asignaciones (o referencias a objetos en Python) y lectura y escritura a través de funciones internas básicas de los lenguajes.

### 2.1. Tipos de datos

La información que se procesa en los programas informáticos se representa de diversas formas. Si tratamos información numérica usaremos **valores** o **datos simples** de tipo enteros o reales. Si trabajamos expresiones lógicas con resultado cierto o falso utilizaremos datos lógicos o booleanos. Si, en cambio, manipulamos texto haremos uso de datos de tipo carácter o cadena de caracteres (*string*). Para representar información numérica (o incluso lógica o de texto) donde se agrupen los datos en forma de tablas, como los vectores y matrices, o estructuras más complejas se emplearán tipos de **datos compuestos**. Éstos se presentarán en detalle en el tema 5.

Los tipos de datos usados en los principales lenguajes de programación se muestran en la figura 8.

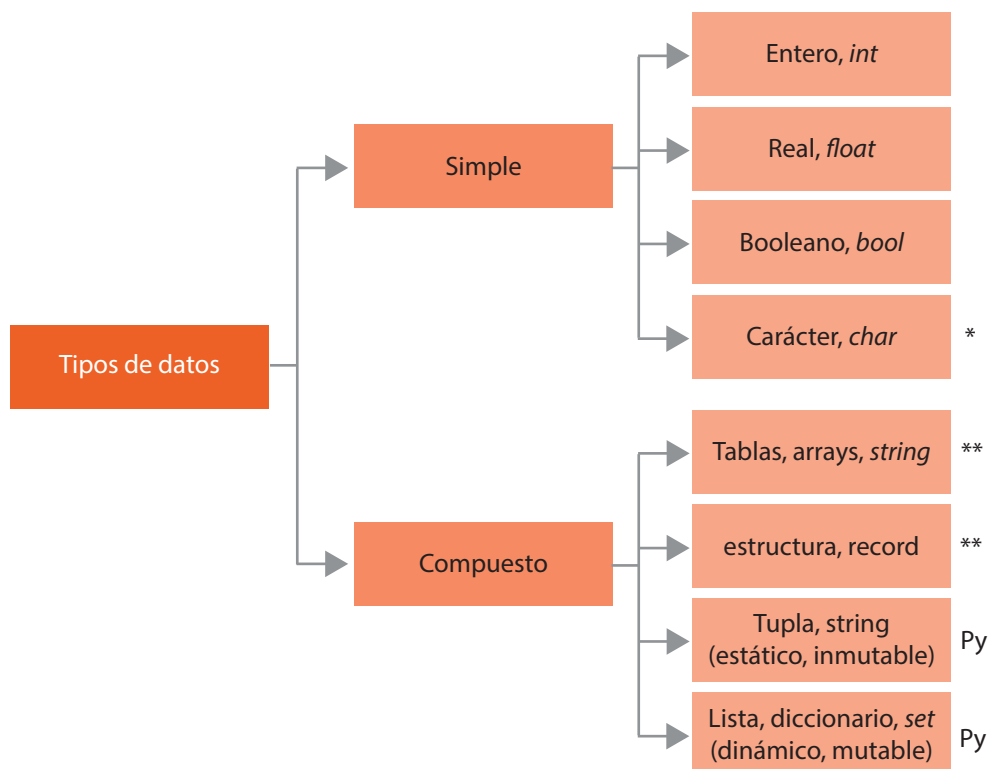


Figura 8. Tipos de datos. \* El tipo de dato carácter no existe en Python, un carácter simple se representa como cadena de caracteres (***string***). \*\* Estructuras compuestas de lenguajes como C, FORTRAN, Pascal, Matlab, etc. Py: Estructuras compuestas en Python.

### 2.1.1. Datos simples

Los datos elementales son los **datos simples**, llamados también escalares por ser objetos indivisibles. Los datos simples se caracterizan por tener asociado un solo valor y son de tipo entero, real o de coma/punto flotante (*float*), booleano y carácter. En Python no existe el tipo de dato simple carácter. Aunque el dato conste de solo una letra o carácter ASCII se representa como un dato compuesto de cadena de caracteres (*string*).

#### a. Enteros

En matemáticas los números enteros (*integer*) son los números naturales, sus negativos y el cero. Ejemplos de enteros: 5, -20, 0, -104. Los números con parte decimal no se incluyen entre los enteros, como el 3.4. Los enteros, en la mayoría de lenguajes de programación incluyendo Python, se definen con la palabra *int*. Python dispone de una función interna **type** que devuelve el tipo de dato dado:

```
>>> type(7)
<class 'int'>
>>> a = 45
>>> type(a)
<class 'int'>
```

En C++ o Pascal se usan 4 bytes (32 bits y se reserva un bit para el signo, *signed 32-bit*) para enteros estándar (*int, integer*), representándose en el **rango** de números:

$$-2147483648 \dots 2147483647$$

Para mayor rango en estos lenguajes se utilizan 8 bytes (*signed 64-bit*), declarándolos como `long` o `int64`:

$$-2^{63} \dots 2^{63-1}$$

A diferencia del C++ o Pascal, en Python los datos de tipo entero se almacenan con “precisión arbitraria”, es decir se utiliza el número de bytes necesarios para representar el número entero. Por ejemplo, los números 5 (binario: 101) y 200 ( $2^7+2^6+2^3$ , binario: 11001000) se representan:

```
>>> bin(5)
'0b101'
>>> bin(200)
'0b11001000'
```

La función interna de Python `bin(N)` convierte un número entero a string con el binario equivalente (0b+binario). Para comprobar el amplio rango de valores en Python, probemos un valor mayor que  $2^{63}$ , como  $2^{220}$ :

```
>>> 2**220
1684996666696914987166688442938726917102321526408785780068975640576
```

## b. Reales

A diferencia de los enteros que son valores discretos de un número natural a otro, los números de valores continuos del conjunto de los números reales en Matemáticas son los llamados reales o de coma/punto flotante<sup>6</sup>, o simplemente *float*. No todos los números reales se

pueden representar de forma exacta en informática, debido a que muchos tienen infinitas cifras decimales. Sin embargo, de acuerdo al nivel de precisión que deseamos se pueden aproximar lo suficientemente bien estos números. Desde hace varias décadas se ha convenido el uso de la norma IEEE 754 para representar los números reales o de punto flotante, usando la notación científica.

Esta notación permite representar los números con una mantisa (dígitos significativos) y un exponente separados por la letra 'e' o 'E'. Por ejemplo el número 4000 se representa por la mantisa 4 y el exponente 3, 4e3. Se lee 4 veces 10 elevado a la 3. El número 0.25 se representa también como 25e-2. También se permite omitir el cero inicial, .25 y el número real 4.0 se puede introducir como 4. (sin el cero después del punto).

La representación en computadores de los números reales o de punto flotante, siguiendo la norma IEEE 754, usa la notación científica con **base binaria**. Python usa esta norma en doble precisión (binary64), con 8 bytes (64 bits):

Signo	Exponente	Mantisa
1 bit	11 bits	52 bits

$$\text{Valor} = (-1)^{\text{Signo}} * 1.\text{Mantisa} * 2^{(\text{Exponente} - 1023)}$$

Tabla 2. Representación de números reales (float) con 64 bits, según norma IEEE 754.

Así, con 64 bits se pueden representar los números (decimales) del  $\pm 5.0 * 10^{-324}$  (precisión) hasta  $\pm 1.7 * 10^{308}$  (rango). La norma IEEE 754 actualizada en 2008 incorpora el formato decimal64 que utiliza la base decimal para mejorar los errores de representación binaria (IEEE Standards Committee, 2008). Python incorpora la función Decimal del módulo decimal con este fin.

A continuación se muestran varios ejemplos de números reales, el tipo real (float) y un error típico con representación de punto flotante con base binaria, en el caso del valor 0.1.

---

decimal para seguir el formato de Python y la mayoría de lenguajes de programación, que usan la forma de la lengua anglosajona.



```
>>> 25e-2
0.25
>>> 4e3
4000.0
>>> type(4.)
<class 'float'>
>>> .2e2
20.0
>>> 1.1 + 2.2 # se representa con error en punto flotante binario
3.3000000000000003
```

### c. Booleanos

El tipo de dato para representar valores lógicos o booleanos en Python es *bool*, en Pascal y C++ se definen como *boolean* y *bool*, respectivamente. Los datos booleanos toman el valor True (1 lógico) o False (0 lógico). El nombre booleano se usa luego que George Boole, matemático inglés, propusiera en el siglo XIX un sistema algebraico basado en estos dos valores lógicos y tres operaciones lógicas: “y lógico”, “o lógico” y la negación. Ejemplos:

```
>>> a = 3 > 2
>>> a
True
>>> type(a)
<class 'bool'>
>>> 4 > 5
False
```

### d. Carácter

El tipo de dato carácter usado en varios lenguajes de programación es el elemento escalar o indivisible de los textos usados en informática. Los textos se llaman cadena de caracteres (en inglés *string*). Los caracteres están ordenados de acuerdo a la tabla ASCII presentada en el anexo A. Por ejemplo, los caracteres ASCII ordenados del valor decimal 20 al 127 son:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN O PQRSTU V
W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

El orden en que se representan sirve para evaluar cuál es mayor que otro, de acuerdo al valor numérico en que figuran en el código ASCII. Por ejemplo ‘b’ es mayor que ‘a’.

El tipo carácter no está definido en Python. Los caracteres simples se definen igual que un texto con una sola letra, es decir como una cadena de caracteres (*string*).

```
>>> type('a')  
<class 'str'>
```

Se puede observar que el carácter 'a' en Python es de tipo string (*str*), aunque en otros lenguajes como Pascal sería de tipo carácter (*char*).

## 2.1.2. Datos compuestos o estructurados

Los **datos compuestos** o estructurados comprenden aquellos datos con elementos de valores de un mismo tipo o de diferentes tipos, que se representan unificados para ser guardados o procesados. En el tema 5 se describirán los tipos de datos compuestos para manejar arreglos matemáticos de vectores o matrices y otros conjuntos de valores heterogéneos de números, texto o booleanos.

### Datos compuestos de cadena de caracteres: string

El tipo de dato **string** es la estructura básica para manejar texto, que incluye caracteres alfanuméricos y demás caracteres de la codificación ASCII o UTF-8. Los string en Python se definen entre comillas simples (' ') o dobles (" "). También se pueden definir entre comillas triples (""" """) cuando se tanguen múltiples líneas. Por ejemplo,

```
>>> 'Hola'  
'Hola'  
>>> b = "Casa de madera"  
>>> type(b)  
<class 'str'>  
>>> type(15)  
<class 'int'>  
>>> type('15')  
<class 'str'>
```

Los textos 'Hola' o "Casa de madera" son de tipo string en general en todos los lenguajes. El valor 15 es un número de tipo entero (*int*), sin embargo, el valor '15' es un string (*str*). Si se incluyen comillas (") o comillas simples (') dentro de un string pueden dar resultados erróneos cuando estos caracteres se usan para delimitar el string. Se puede emplear dentro del texto aquel carácter que no se ha usado de delimitador:

```
>>> 'Ella dijo "Qué lindo"'
'Ella dijo "Qué lindo"'
>>> "He doesn't know"
"He doesn't know"
```

Sin embargo, en estos casos se puede también usar el carácter barra invertida (\) que sirve de escape para agregar comillas u otras acciones dentro del string:

```
>>> print('He doesn\'t know I \'will come\'')
He doesn't know I "will come"
```

El carácter barra invertida (llamado carácter de escape) seguido de n (\n) indica salto a nueva línea. Se pueden incluir múltiples líneas en un string usando triples comillas `""" ... """`. En este caso los fines de línea están incluidos. Los salto a nueva línea se aprecian cuando se presentan en pantalla con la función interna `print()`:

```
>>> print('Cambiamos de línea \nNueva línea')
Cambiamos de línea
Nueva línea
>>> """
Programa:
Autor:
Fecha:
"""
'\n Programa:\n Autor:\n Fecha:\n '
```

## 2.2. Variables y acción de asignación

En matemáticas las **variables** se usan para representar valores numéricos. Se utiliza un carácter o texto para representarlas. En cálculo matemático una función del tipo  $y = f(x)$  involucra dos variables,  $x$  e  $y$ .

En los lenguajes de programación se requiere normalmente recordar o guardar los valores numéricos, booleanos o de texto para ser usados una o múltiples veces en el programa. Las variables tienen este cometido. En los lenguajes como el FORTRAN, C/C++ o Pascal, una **variable** se considera un contenedor o lugar dentro de la memoria RAM del computador, con un nombre asociado (identificador), donde se guarda un valor de un tipo determinado. Al usar programas informáticos podemos quedarnos con este concepto. Sin embargo, en Python el concepto es algo diferente, pues las variables no son un lugar de memoria que contienen un valor sino que se asocian, o refieren, a

un lugar de memoria que contiene ese valor. Los valores pueden ser entero, real, booleano, o datos compuestos.

La acción de **asignación** se usa para darle a una variable un valor determinado. En Python la acción de asignación de valores a una variable quiere decir que la variable con su nombre determinado se va a asociar al valor de la derecha de la asignación:

```
>>> a = 7
```

El símbolo = indica que el valor 7 se asigna a la variable a. El mismo símbolo se usa en C/C++ o FORTRAN, pero en Pascal se usa := como asignación. Sin embargo, lenguajes como el R utilizan el signo <- que refleja mejor la asimetría de esta acción, donde el valor de la derecha se asigna a la variable de la izquierda.

Lenguaje de Programación	Símbolo de asignación
C/C++, FORTRAN, Python, Java	=
Pascal	:=
R	<-

Tabla 3. Símbolos de asignación a variables en diferentes lenguajes.

En Python lo que ocurre al ejecutar la sentencia o instrucción  $a = 7$  es que se crea primero un **objeto** con el valor 7 de tipo *int* y se ubica en un lugar de memoria. Este lugar de memoria se denomina en Python identidad del objeto. Luego, la acción de asignar 7 a la variable a hará que este nombre o identificador se asocie o refiera a la dirección de memoria del objeto 7, o sea, a tendrá la misma identidad que 7. Usaremos la función interna *id()* de Python para entender mejor esta referencia de la variable al lugar de memoria donde está 7. La función *id()* devuelve la identidad o lugar de memoria donde se ubica el objeto.

```
>>> id(7)
1449917120
>>> a = 7
>>> id(a)
1449917120
>>> b = 7
>>> id(b)
1449917120
>>> c = a
>>> id(c)
1449917120
```

Tanto 7, *a*, *b* o *c* son el mismo objeto y ocupan una posición única de memoria. En este caso la posición<sup>7</sup> 1449917120. La variable *b* al asignarle el mismo valor (y objeto) 7, se referenciará a la misma posición. Lo mismo si le asignamos a la variable *c* la variable *a*; *c* se referenciará a la misma posición de 7. Esto hace que Python maneje más eficientemente la memoria pues en otros lenguajes se triplicaría el uso de ésta al tener tres variables con el mismo valor. Pero ¿qué pasa si usamos el valor 7.0 en lugar de 7?:

```
>>> x = 7.0
>>> id(x)
1722264
>>> type(x)
<class 'float'>
```

La variable *x* estará asociada al objeto 7.0, con identidad 1722264, de tipo float y valor 7.0. Así, los objetos de Python tienen tres características: valor, tipo e identidad. Como se ha notado, las variables no necesitan que sea declarado su tipo antes de ser usadas, como en Pascal o C. Pueden incluso cambiar de tipo a lo largo del programa.

Para apreciar mejor el concepto de asignación en programación y diferenciarlo de la simetría del símbolo = en matemáticas, probemos:

```
>>> 7.0 = x
SyntaxError
>>> x = x + 3
>>> x
10.0
```

La sentencia *7.0 = x* es un error de sintaxis en programación, aunque es válido en matemáticas. Pero *x = x + 3* sería absurdo en matemáticas, pero en lenguajes de programación significa sumarle a la variable *x* el valor 3 y el resultado de esa expresión asignarlo luego a la misma variable *x*.

Python permite asignaciones múltiples del tipo:

```
>>> x, y, z = 7, 8.2, 9
```

La variable *y* es real de valor 8.2, *x* es entera de valor 7 y *z* entera de valor 9. A la derecha de la asignación se ha escrito un dato compuesto, una tupla en Python, que se asigna a la tupla formada por las tres variables. Las tuplas y este tipo de asignación se verán con más detalle en el tema 5.

---

<sup>7</sup> Este valor es arbitrario. Se puede obtener cualquier otra posición de memoria.

## Identificadores y palabras reservadas.

Al nombre de una variable lo llamaremos **identificador**. Pero los nombres de otros elementos de los programas, como funciones, clases, librerías también tendrán identificadores. Los identificadores en programación pueden contener letras y números pero deben empezar siempre con una letra o el carácter guion bajo o subrayado “\_”. Aunque los matemáticos suelen usar nombres con una sola letra para las variables, en programación muchas veces es preferible utilizar identificadores con nombres que se asocien con su significado. Como, *area*, *volumen*, *lead\_III*, *lado2*. Hay que hacer notar también que en la mayoría de los lenguajes, incluyendo Python, los identificadores son sensibles al tipo de letra minúscula-mayúscula<sup>8</sup>. Es decir *n* y *N* son variables diferentes.

En los lenguajes de programación hay un grupo de palabras reservadas de operaciones, instrucciones y funciones internas que no pueden usarse como identificadores. Se puede imprimir en el Shell de Python la lista de palabras reservadas en este lenguaje, llamadas “*keywords*”:

```
>>> help("keywords")

False      def        if         raise
None       del        import    return
True       elif       in        try
and        else       is        while
as         except    lambda    with
assert     finally   nonlocal  yield
break      for       not
class      from      or
continue   global    pass
```

Hay que tener cuidado también con las funciones internas o predefinidas del lenguaje (*built-in functions*)<sup>9</sup>. Si identificamos una variable con el nombre de una función predefinida, ésta luego no podrá ser llamada pues su identificador se ha sobrescrito con la variable. Por ejemplo, hemos usado las funciones `type` para saber el tipo de dato usado. Si a una variable la identificamos con el nombre `type`, perdemos esta función en el programa:

---

8 Python permite identificadores con vocales acentuadas, como *área*. Aunque no es recomendable esta práctica por si se cambia de lenguaje de programación.

9 Se pueden consultar la lista actual de funciones internas en: <https://docs.python.org/3.3/library/functions.html>

```
>>> type(15)
<class 'int'>
>>> type = "Hola"
>>> type(15)
Traceback (most recent call last):
  File "<pysHELL#30>", line 1, in <module>
    type(15)
TypeError: 'str' object is not callable
```

## 2.3 Expresiones y sentencias

Las **expresiones** son el mecanismo para hacer cálculos y se componen de combinaciones de valores e identificadores con operadores. Pueden incluir variables, datos, operadores, paréntesis y funciones que devuelven resultados.

Toda expresión tiene un valor que es el resultado de evaluarla de izquierda a derecha, tomando en cuenta las precedencias. Ejemplos de expresiones:

```
>>> 1.5*3/2
2.25
>>> 1.2*x + 3 # El valor de x del ejemplo anterior es 10.0
15.0
>>> 3 > (3.1 + 2)/3
True
```

Las **sentencias** o **instrucciones** son las unidades básicas de los programas (llamados también en el argot de los programadores, códigos) que produce una acción, como asignar un valor a una variable, mostrar un resultado, etc. El intérprete de Python ejecuta cada sentencia produciendo la acción dada.

```
>>> y = x/2 + 3
>>> print(y)
8.0
```

```
>>> 1.5*3/2
2.25
>>> print(_)
2.25
```

La primera sentencia del cuadro de la izquierda calcula la expresión  $x/2 + 3$  y el resultado lo asigna a la variable `y`. La segunda sentencia muestra el valor de `y`. Pero, la expresión del cuadro de la derecha  $1.5*3/2$ , cuyo cálculo no es asignado a ninguna variable, su resultado se guarda asociada a la variable llamada `_`.

## 2.4. Operadores

Los **operadores** son los símbolos que representan las acciones de cálculo. Adicional a las operaciones clásicas matemáticas, en programación se utilizan operadores lógicos y de relaciones o comparación. Los operadores los podemos clasificar de 3 tipos: operadores aritméticos, operadores lógicos o booleanos y operadores relacionales.

### 2.4.1. Operadores aritméticos

En Python estos son los operadores de cálculo matemático:

Operación	Operador	Expresión	Resultado tipo
Suma	+	$a + b$	Entero si a y b enteros; real si alguno es real
Resta	-	$a - b$	Entero si a y b enteros; real si alguno es real
Multiplicación	*	$a * b$	Entero si a y b enteros; real si alguno es real
División, $a \div b$ (real)	/	$a / b$	Siempre es real
División (entera)	//	$a // b$	Devuelve la parte entera del cociente $a \div b$
Módulo (resto)	%	$a \% b$	Devuelve el resto de la división $a \div b$
Exponenciación, $a^b$	**	$a ** b$	Entero si a y b enteros; real si alguno es real

Tabla 4. Operadores aritméticos en Python.

*Ejemplos.*

```
>>> 14/4      # División real
3.5
>>> 14//4     # División, devuelve parte entera de dividir 14 entre 4
3
>>> 14%4      # Módulo, devuelve el resto de dividir 14 entre 4
2
```



Se pueden incluir entre los operadores aritméticos los que operan sobre un solo operando, llamados unarios: operador cambio de signo `-` y operador identidad `+`. Por ejemplo, `-4`, `+4`, `--4` equivale a `4`.

### Operadores aritméticos con asignaciones

La acción de incrementar el valor de una variable es muy común en programas informáticos. Por ejemplo, en un contador de eventos, el contador `c` se incrementa en 1.

```
>>> c = c + 1    # la variable c se incrementa en 1
>>> d = d - 1    # la variable d se decrementa en 1
```

Python incluye sentencias que compactan el operador de asignación con cualquier operador aritmético. En los casos de incremento o decremento de una variable tenemos,

```
>>> c += 1        # equivale a: c = c + 1
>>> x += 0.01     # equivale a: x = x + 0.01
>>> d -= 2        # equivale a: d = d - 2
```

Los demás operadores aritméticos (`*`, `/`, `//`, `%`, `**`) también pueden usarse en esta forma compactada. Pueden incluirse también expresiones como incremento u otra operación. Por ejemplo,

```
>>> y *= d+1      # equivale a: y = y * (d+1)
>>> n = 7
>>> n //= 2       # equivale a: n = n // 2
>>> n
3
>>> n **= 2       # equivale a: n = n**2
>>> n
9
```

## 2.4.2. Operadores lógicos

Los operadores lógicos o booleanos operan sobre tipo de datos booleanos. Estas operaciones son “y lógico” o conjunción, “o lógico” o disyunción sobre dos operandos y la “negación” que es un operador unario. En Python las palabras reservadas para estas operaciones son ***and***, ***or*** y ***not***, respectivamente. Recordamos que los valores lógicos en Python son `True` y `False` para los valores cierto (1 lógico) y falso (0 lógico), respectivamente.

En la siguiente tabla de verdad se muestra el resultado de las operaciones lógicas (`True` equivale a 1, `False` a 0):

A	B	A or B	A and B	not A	equivale	A	B	A or B	A and B	not A
False	False	False	False	True		0	0	0	0	1
False	True	True	False	True		0	1	1	0	1
True	False	True	False	False		1	0	1	0	0
True	True	True	True	False		1	1	1	1	0

Tabla 5. Tablas de verdad con operadores lógicos. Izquierda: con valores lógicos en Python. Derecha: con valores lógicos 0 (falso) o 1 (cierto).

Por ejemplo.

```
>>> A = True
>>> type(A)
<class 'bool'>
>>> B = False
>>> A or B
True
```

En el siguiente cuadro se resumen algunas leyes lógicas de utilidad para tratar expresiones booleanas. Se pueden demostrar usando la tabla de verdad de cada lado de la igualdad.

not not A	=	A
A and True	=	A
A and False	=	False
A or False	=	A
A or True	=	True
not (not A and not B)	=	A o B
not (not A or not b)	=	A y B

Aunque no será tratado dentro del tema de este curso, Python y otros lenguajes como el C++ incluyen operadores lógicos sobre números binarios, realizados bit-a-bit. Estos operadores de bits (*bitwise operators*) realizan las operaciones lógicas sobre los bits del número binario equivalente al número decimal introducido. Por ejemplo, el  $5_{10}$  equivale a 101 y el  $6_{10}$  es 110. Si realizamos un “y lógico” bit a bit tendremos el binario 100, que equivale a  $4_{10}$ . A nivel de lenguaje de máquina se usa también el operador or exclusivo (*xor*) que devuelve 1 cuando solo uno de los operandos es 1. En Python, como en C++, los operadores de bits son<sup>10</sup>:

10 Para más detalles ver: <https://wiki.python.org/moin/BitwiseOperators>.

Operación de bits	Operador	Expresión	Resultado
"y lógico", and	&	5 & 6	101 and 110 -> 100, 4 decimal.
"o lógico", or		5   6	101 or 110 -> 111, 7 decimal.
"o exclusivo", xor	^	5 ^ 6	101 xor 110 -> 011, 3 decimal.
Complemento, not	~	~x	Cambia 0's por 1's y 1's por 0's. Equivale a -x -1.
Desplazamiento izquierda	<<	x << n	Devuelve x con los bits desplazados n lugares a la izquierda. Equivale a $x * 2^{**n}$
Desplazamiento de- recha	>>	x >> n	Devuelve x con los bits desplazados n lugares a la derecha. Equivale a $x / 2^{**n}$

Tabla 6. Operadores lógicos bit a bit sobre números binarios.

### 2.4.3. Operadores relacionales

Son usados para comparar 2 expresiones o valores y el resultado es siempre cierto o falso, es decir, booleano.

Operadores Relacionales				
Matemáticas	En Python	Significado	Ejemplo	Resultado
=	==	Igual a	'a' == 'b'	False
≠	!=	Distinto a	'b' != 'B'	True
<	<	Menor que	7 < 3	False
>	>	Mayor que	7 > 3	True
≤	<=	Menor o igual que	7 >= 3	True
≥	>=	Mayor o igual que	7 <= 7	True

Tabla 7. Operadores relacionales (de comparación) en matemáticas y su equivalencia en Python.

En el ejemplo siguiente usaremos los operadores relacionales para chequear que la temperatura medida, que le asignaremos a la variable *temp*, está o no entre 37 y 42 °C, ambas inclusive.

```
>>> temp = 38      # temperatura medida
>>> (temp >= 37) and (temp <= 42)
True
```

La expresión  $(temp \geq 37) \text{ and } (temp \leq 42)$  puede ir sin paréntesis ya que, como veremos en la siguiente sección, los operadores relacionales tiene mayor prioridad o precedencia que los booleanos. Python tiene la característica particular que este tipo de expresión se puede escribir como en notación matemática:  $37 \leq temp \leq 42$ . Expresiones del tipo  $a < b < c$  en Python significan  $(a < b) \text{ and } (b < c)$ , lo cual mejora la legibilidad del lenguaje.

Los caracteres del alfabeto (sin la ñ) que pertenecen a la tabla ASCII están ordenados. Las minúsculas del binario equivalente decimal 97 al 122 y las mayúsculas del 65 al 90. Aunque en Python los caracteres son de tipo string, podemos comparar caracteres y su resultado *True* o *False* se dará de acuerdo a su posición ASCII. Por ejemplo, *'b' > 'a'* devolverá *True*. Igualmente, *'B' > 'a'* devolverá *False*.

En el siguiente ejercicio buscaremos una expresión que sea cierta cuando, dada una variable *car*, ésta sea un símbolo del alfabeto:

```
>>> car = 'q'
>>> (car >= 'a') and (car <= 'z') or (car >= 'A') and (car <= 'Z')
True
>>> # Equivale a
>>> 'a' <= car <= 'z' or 'A' <= car <= 'Z'
True
>>> car = '&'
>>> 'a' <= car <= 'z' or 'A' <= car <= 'Z'
False
```

El resultado se puede asignar a una variable booleana:

```
>>> es_letra = 'a' <= car <= 'z' or 'A' <= car <= 'Z'
>>> es_letra
False
```

En la mayoría de lenguajes, incluyendo Python, el valor booleanos *True* (1 lógico) es mayor que *False* (0 lógico):

```
>>> True > False  
True
```

#### 2.4.4. Orden de las operaciones

Para resolver expresiones con múltiples operadores, incluyendo diferente tipo de operador, aritmético, booleano o relacional, se debe seguir un orden de prioridad para realizar primero una operación y luego otra. En caso de dudas, o para mejorar la legibilidad del programa, es recomendable que se usen paréntesis. El orden de prioridad o precedencias es diferente de acuerdo al tipo de lenguaje de programación. En Python las precedencias son similares a C++ y Matlab.

La máxima prioridad la tiene la realización de las expresiones entre paréntesis. Luego la exponenciación y las operaciones unarias. Después siguen las operaciones aritméticas, que a su vez siguen la convención matemática donde se priorizan la multiplicación y división sobre la suma y resta. En inglés se suele usar el acrónimo PEMDAS (Paréntesis, Exponenciación, Multiplicación - División, Adición - Sustracción). Después de las aritméticas siguen las operaciones relacionales y por último las booleanas, pero con mayor prioridad de la negación, luego and y por último or. Los operadores exponenciación y unarios (identidad, cambio de signo y negación lógica) se asocian con el operando de la derecha. El resto de operadores se asocian primero con el operando de la izquierda. En las expresiones con operadores de igual precedencia se ejecutan de izquierda a derecha, exceptuando la exponenciación que va de derecha a izquierda.

Por ejemplo, la expresión  $7/2*3$  calcula primero la división 7 entre 2 y el resultado lo multiplica por 3, resultando 10.5. Si se quiere dividir entre el producto  $2*3$ , hay que encerrarlos entre paréntesis:  $7/(2*3)$ . La asociación por la derecha de la exponenciación se puede ver en la siguiente expresión  $2**4**2$  y  $(2**4)**2$ .

```
>>> 2**4**2  
65536  
>>> (2**4)**2  
256
```

La tabla siguiente muestra el orden de precedencia en Python.

Operador	Precedencia
()	Mayor
**	
+x, -x (identidad, cambio de signo)	
*, /, //, %	
+, -	
==, !=, <, >, <=, >=	
not	
and	v
or	Menor

Tabla 8. Precedencia o prioridad de las operaciones en Python.

### Operaciones con texto (strings)

Los textos o cadena de caracteres (strings) no pueden operarse matemáticamente. Sin embargo el operador + sí que realiza la interesante acción de **concatenar** dos strings. Esta acción será útil para guardar información numérica en un string, convirtiendo previamente los números enteros o reales a string. Luego, el string puede ser escrito en pantalla o simplemente guardado en una variable. También el operador \* realiza la repetición del string tantas veces como el número dado. Veamos varios ejemplos:

```
>>> '40' + '8'
'408'
>>> Name = 'Luis'
>>> Apellido = 'Garcia'
>>> Name + ' ' + Apellido
'Luis Garcia'
>>> 'Ven'*4          # equivale a 'Ven'+ 'Ven'+ 'Ven'+ 'Ven'
'VenVenVenVen'
>>> 10* '-'
'-----'
```

## 2.5. Acciones elementales

Los procesos en los programas informáticos incluyen acciones de entrada de información, de cálculo o procesamiento de información y, finalmente, salida de datos.

Hemos introducido ya una de las acciones elementales de los procesos informáticos como es la **asignación** de valores a las variables. Las otras acciones elementales incluyen la **lectura** de información para ser procesada a través de variables por el programa y la **escritura** de datos en pantalla para mostrar los resultados de estos procesos. Otras formas de entrada y salida de datos se hacen a través de lectura y escritura de ficheros que se guardan en memoria externa al procesador central ya sea en medios magnéticos como discos duros o electrónicos de estado sólido como las memorias flash. En el tema 5 se hace un breve análisis del uso de ficheros.

### 2.5.1. Lectura de datos

La lectura de información numérica booleana o de texto se realiza desde el teclado del ordenador a través de funciones internas de los lenguajes de programación. En Pascal y C se usan las funciones *read* y *scanf*, respectivamente. En Python se utiliza la función interna *input()*. Esta función utiliza un mensaje de texto para indicar al usuario del programa qué tiene que introducir. Lo que se teclea y se introduce al programa, que se suele asignar a una variable, es un valor de tipo string. Por lo que hay que tener en cuenta que si los datos son numéricos habrá que convertirlos de string a enteros (*int*) o a reales (*float*). Así mismo, si el valor es booleano habrá que convertirlo a tipo de dato booleano (*bool*).

Veamos varios ejemplos de distintos tipos de información, leídos del teclado a través del Shell de Python:

```
>>> Nombre = input('Cómo te llamas? ')
Cómo te llamas? José
>>> type(Nombre)
<class 'str'>
>>> Edad = int(input('Introduce tu edad: '))
Introduce tu edad: 21
>>> type(Edad)
<class 'int'>
>>> Altura = float(input('Cuánto mides? '))
Cuánto mides? 1.78
>>> type(Altura)
<class 'float'>
```

Como se puede ver, las entradas de datos con la función *input()* hay que convertirlas a los tipos de datos numéricos adecuados, en caso que se requiera.

### 2.5.2. Conversión entre tipos de datos

Así como la función interna *type()* devuelve el tipo de dato del valor o variable introducida, se pueden convertir datos de un tipo a otro, con ciertas restricciones.

La función interna *float()* devuelve el número real del dato en formato string o entero que entre a la función. Ejemplos:

```
>>> float('123')
123.0
>>> float(Edad) # Variable Edad de tipo int del ejemplo anterior
21.0
>>> float('abc')
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    float('abc')
ValueError: could not convert string to float: 'abc'
```

La función *int()* convierte un string (que sea un número en formato texto) o un real a entero. En caso de un real, la función lo redondea hacia cero. Ejemplos:

```
>>> int('123')
123
>>> int(27.8)
27
>>> int(-24.9)
-24
```

La función *bin()* representa (en un string) un entero a su equivalente en binario. La función *str()* convierte un número a formato string. Ejemplos:

```
>>> bin(255)
'0b11111111'
>>> bin(256)
'0b100000000'
>>> str(254)
'254'
>>> str(1/3)
'0.3333333333333333'
```

### 2.5.3. Escritura de datos

La escritura consiste en mostrar valores de texto, numéricos o booleanos, y también resultados de expresiones o de variables al usuario del programa. Así como en Pascal o C se usan funciones internas llamadas *write*, *writeln* o *printf*, en Python la función interna es *print*, que ya ha sido utilizada en varios

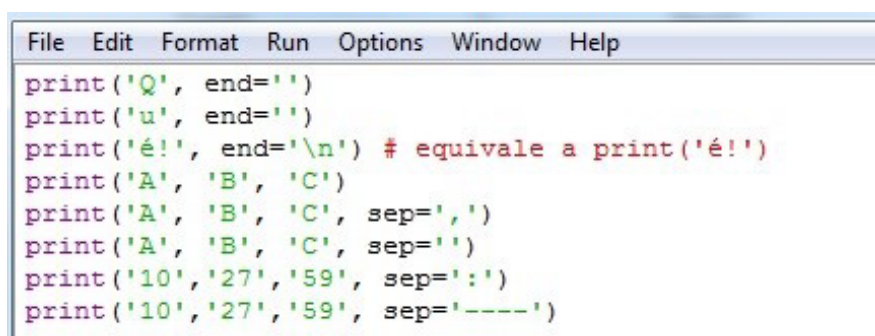


ejemplos anteriores. La función `print()` permite mostrar valores de variables (de cualquier tipo) o expresiones, texto en formato string o combinaciones de todas ellas. Alternativamente, en un string se pueden concatenar datos numéricos luego de ser convertidos a string con la función interna `str()`. Por ejemplo,

```
>>> a = 3.5
>>> b = 15
>>> print('El producto de', a , 'por', b , 'es', a*b)
El producto de 3.5 por 15 es 52.5
>>> print('El producto de '+str(a)+' por '+str(b)+' es '+str(a*b))
El producto de 3.5 por 15 es 52.5
```

La función `print` escribe como string todo su argumento (entrada) que puede ser una secuencia de expresiones separadas por coma. En el primer `print` del ejemplo anterior se separa (por omisión) cada elemento entre las comas por un espacio en blanco. Se puede notar que están combinados strings, enteros y reales, pero todo es convertido a una representación de texto. Al finalizar el `print` se avanza a una nueva línea. De hecho, se puede usar `print()` como avance a nueva línea.

Existen parámetros de la función `print()` que indican cómo terminar la escritura (`end`) y cómo separar (`sep`) los elementos del argumento de la función. Cuando se quiera terminar la función `print` sin que salte a una nueva línea, y un nuevo `print` continúe en la misma línea, se agrega al final el argumento `end=""` (comillas simples sin espacio entre ellas). Cuando no se usa el `end`, se utiliza el valor por omisión de salto a nueva línea, que sería `end='\n'`. En lugar de separar las secuencias por espacios en blanco, se puede omitir la separación (`sep=""`) o colocar el string que se quiera como símbolo separador. El ejemplo de la figura 9 muestra un programa en su código fuente con diversas opciones del uso de `print`.



```
File Edit Format Run Options Window Help
print('Q', end='')
print('u', end='')
print('é!', end='\n') # equivale a print('é!')
print('A', 'B', 'C')
print('A', 'B', 'C', sep=',')
print('A', 'B', 'C', sep='')
print('10', '27', '59', sep=':')
print('10', '27', '59', sep='----')
```

Figura 9. Imagen del editor de Python con varias opciones de la función `print()`.

A continuación se muestra un panel con el resultado de las instrucciones `print()` de la figura 9 sobre el Shell:

```
Qué!  
A B C  
A,B,C  
ABC  
10:27:59  
10----27----59  
>>>
```

El formateo clásico de strings usado en muchos lenguajes, como el *printf*, para mostrar, por ejemplo, datos numéricos con un número de decimales especificados o un ancho determinado, se puede hacer en Python con el viejo estilo de *printf* dentro del *print()* o con el método de *str.format()*. La descripción de este formato escapa al alcance de este manual. El lector interesado puede consultar a Klein (2016).

### 2.5.4. Comentarios

El lector habrá observado el uso del símbolo # para comentar sentencias o partes del programa. Los programas largos o incluso cualquier programa realizado algún tiempo atrás pueden ser complicados de seguir. Es una buena opción comentar en lenguaje natural lo que las instrucciones del programa están haciendo.

```
"""  
Created on Mon Jan  4 23:59:52 2016  
  
@author: gomis  
"""  
  
# Cálculo de la frecuencia cardíaca  
RR = 0.875      # Intervalo R-R en segundos  
freq = 1/RR*60  # Frecuencia en pulsaciones por minuto
```

Figura 10. Imagen del editor de Python usando comentarios.

En el ejemplo de la figura 10 se ha incluido un string con comillas triples como encabezado del programa. Algunos entornos de desarrollo de programas en Python, como el Spyder de Winpython (<http://winpython.github.io/>), incluyen de manera automática este encabezado de inicio en el editor de programas.

## Tema 3.

### Estructuras Algorítmicas

Desde la introducción de los primeros computadores digitales con programas almacenados, siguiendo las ideas pioneras de Turing y la arquitectura de von Newman, se desarrollaron programas muy complejos para cálculo en ingeniería o para aplicaciones administrativas, difíciles de seguir o depurar. Muchos lenguajes de programación incluían la acción “go to”, que llevaba la secuencia de ejecución del programa de un punto a otro, adelante y atrás, que hacía difícil su seguimiento. A finales de la década de 1960 surgió el paradigma de la **programación estructurada**, basado en las ideas de Böhm y Jacopini (1966) y las notas publicadas de Dijkstra (1968, 1969).

La **programación estructurada** contempla el uso de estructuras o composiciones **secuenciales**, de selección o **alternativas** e **iterativas** y con ellas se puede resolver cualquier función computable; es decir, siguiendo la tesis de Turing, cualquier función o algoritmo que pueda ser calculado por un computador. La programación estructurada produce códigos o programas más fáciles de leer, depurar y actualizar. También propicia el uso de módulos y subprogramas que sean reutilizables y el programa sea más manejable.

Los subprogramas serán introducidos en el siguiente tema, pero se refieren a las funciones en los diversos lenguajes de programación, incluyendo Python. Hay funciones internas como algunas ya

usadas, o diseñadas por el programador. En otros lenguajes de programación como Pascal y ADA también incorporan los procedimientos (*procedures*) como un subprograma que no devuelve resultado sino que realiza una acción.

El conjunto de sentencias o instrucciones de las acciones elementales descritas previamente (asignación de valores o expresiones a variables, *input*, *print*, etc.) forman parte del llamado código de los programas o subprogramas. Además se añaden las estructuras o composiciones algorítmicas que comprenden secuencias, alternativas e iteraciones o bucles.

### 3.1. Estructura secuencial

La estructura secuencial, o llamada también composición secuencial, es la más simple en programación pues se trata de una serie de acciones, instrucciones o sentencias que se procesan secuencialmente en bloque una a una. Las instrucciones pueden incluir llamadas a subprogramas. En Python, cada instrucción es ejecutada por el intérprete traduciéndola al lenguaje de máquina. Los ejemplos que se han presentado previamente, tanto en el Shell de Python como los códigos en el editor de programas de Python, son estructuras secuenciales.

Estructura secuencial			
Pseudocódigo	Python	Pascal	C
Inicio	Instrucción 1	Begin	{
Instrucciones	Instrucción 2	Instrucciones	Instrucciones
...	...	...	...
Fin	Instrucción n	end	}

Tabla 9. Composición secuencial en diversos lenguajes.

En lenguajes como Pascal los bloques de secuencias del programa, incluyendo las que se ejecutan dentro de una estructura algorítmica alternativa o iterativa se colocan entre delimitadores **begin** y **end**; en C/C++ entre llaves {}. En Python no se usan delimitadores, aunque, tal como se describe en la siguiente sección, las secuencias dentro de las composiciones algorítmicas deben ir indentadas (sangradas) o tabuladas.

Hemos presentado la mayoría de ejemplos con la opción interactiva del Shell de Python. A partir de ahora los códigos de los ejemplos o ejercicios resueltos se presentarán en fondo blanco con la fuente original del editor <sup>11</sup> y los resultados sobre el Shell (cuadro fondo gris). En el siguiente ejemplo se convierte grados Celsius a Fahrenheit; al ejecutarlo (Run o F5 en el Editor) e introducir por ejemplo el valor 27, se muestra el resultado en el Shell de Python:

<sup>11</sup> El editor del desarrollador de programas Spyder de winpython (<http://winpython.github.io/>) muestra advertencias de errores de sintaxis a medida que se edita el programa.

```
# Programa que convierte °C a °F
C = float(input("Entra temperatura en °C: "))
F = 1.8*C + 32
print(C, "°C equivale a", F, "°F")
```

```
Entra temperatura en °C: 27
27.0 °C equivale a 80.6 °F
>>>
```

El programa siguiente calcula el área y el perímetro de un círculo a partir de su radio. Como se requiere el uso del valor  $\pi$  (pi) lo importaremos del módulo de funciones y constantes matemáticas de Python (*math*). Se introducirá del teclado el radio de valor 1.2.

```
# Programa que calcula perímetro y área de un círculo
from math import pi
r = float(input('Introduce el radio del círculo: '))
per = 2*pi*r
area = pi*r**2
print('Perímetro =', per)
print('Área =', área)
```

```
Introduce el radio del círculo: 1.2
Perímetro = 7.5398223686155035
Área = 4.523893421169302
```

Los decimales de los números reales (*float*) del perímetro y área se pueden reducir, por ejemplo, a dos dígitos decimales utilizando el estilo adaptado *printf* de otros lenguajes (%.2f) o usando la función interna de Python *round(x, 2)*. Puede cambiarse el 2 por el número de decimales que se requiera.

```
print('Perímetro = %.2f', % per)    # funcionalidad de printf en Python
print('Área =', round(área, 2))    # función interna round()
```

```
Introduce el radio del círculo: 1.2
Perímetro = 7.54
Área = 4.52
```

## 3.2. Estructuras alternativas

La estructura o composición que selecciona ejecutar un conjunto de instrucciones u otro es la alternativa o condicional. Si se cumple como cierta una expresión lógica o booleana entonces el programa realiza una acción o bloque de secuencia de instrucciones. Si, opcionalmente, la condición es falsa entonces el programa realiza otro bloque de instrucciones. Por ejemplo, si queremos cambiar de signo un número si es negativo (buscar su valor absoluto) podemos hacer: "si el número es negativo entonces cambiarlo de signo (y si no, no hacemos nada)". Este ejemplo se puede realizar en lenguajes de programación con la sentencia `if`.

### 3.2.1. Estructura alternativa simple o condicional

El ejemplo anterior requiere la sentencia `if` de forma simple, en Python de la forma,

*if condición:*  
*secuencia\_de\_instrucciones*

La "condición" es una expresión booleana (cierta o falsa). Si esta condición es cierta, entonces se ejecuta el bloque o secuencia de instrucciones dentro del `if`. Todos los bloques a ejecutar dentro de las estructuras algorítmicas en Python aparecen indentadas varios espacios o una tabulación a la derecha. El número de espacios debe ser siempre el mismo. Se ha convenido que se usen siempre 4 espacios en blanco. Los editores de programas de Python indentan (sangran) automáticamente 4 espacios en la línea siguiente al carácter ':' (final de la instrucción). Así, el ejemplo de hallar el valor absoluto de un número será:

```
x = float(input("Entra un número: "))
if x < 0:
    x = -x
print('El valor absoluto es:',x)
```

Si `x` es un valor negativo (condición cierta), entonces se ejecuta la instrucción dentro del `if`. Y si no es cierta, entonces no se hace nada y se pasa a la instrucción siguiente, `print()`, que está en la misma columna del editor que el `if`. Por cierto, Python incluye una función interna que calcula el valor absoluto de un número: `abs(x)`.

Otro ejemplo de estructura alternativa simple sería un programa que cubre un saldo bancario de cuenta corriente descubierto (saldo negativo) con fondos de una cuenta de ahorros:

```
#Balance financiero
CuentaAhorros = 10000
saldo = float(input("Cuál es el saldo de la cuenta corriente?: "))
if saldo < 0:
    transferir = -saldo
    CuentaAhorros = CuentaAhorros - transferir
    saldo = saldo + transferir
print('Fondos cuenta de ahorro:', CuentaAhorros)
```

### 3.2.2. Estructura alternativa doble (if-else)

En la estructura alternativa doble o **if-else** se ejecuta un bloque de instrucciones cuando la condición es cierta, pero si es falsa, se ejecutan otras sentencias. Se tiene una bifurcación o dos ramas de secuencias. La sintaxis es:

```
if condición:
    secuencia_de_instrucciones_condicion_cierta
else:
    secuencia_de_instrucciones_condicion_falsa
```

Por ejemplo, queremos evaluar si un número natural es par o impar. Podemos usar el operador módulo (%) que devuelva el resto de la división entre 2. Si el resto es 0, el número será par y si no, impar:

```
# Paridad de un número
N = int(input("Entra un número natural: "))
if N%2 == 0:
    text = 'es par'
else:
    text = 'es impar' # equivaldría a: if N%2 != 0:
print('El número', N, text)
```

Nótese que este ejemplo de la paridad de un número pudiera escribirse con dos estructuras condicionales simples, donde en lugar del *else* se utiliza otro *if* con la expresión lógica opuesta como condición. Se obtendría el mismo resultado pero, además de poco elegante, dificulta la legibilidad del programa.

En el siguiente ejemplo se resuelve una ecuación de primer grado, que debe tomar en cuenta evitar la división por 0, ya que si no el programa daría un error:

```
# Halla solución x, de ecuación ax + b = 0
print('Programa que halla el valor de x de la ecuación: ax + b= 0')
a = float(input('a= '))
b = float(input('b= '))
if a != 0:
    x = -b/a
    print('x =', x)
else:
    print('No es posible dividir por cero')
```

El programa de conversión de grados Celsius a Fahrenheit, visto previamente, se puede mejorar advirtiendo la posible inclusión de una temperatura irreal por debajo del cero absoluto:

```
# Programa que convierte °C a °F
C = float(input("Entra temperatura en °C: "))
if C < -273.15:
    print('Temperatura en °C irreal por debajo del 0 absoluto!!')
else:
    F = 1.8*C + 32
    print(C, '°C equivale a', F, '°F')
```

### 3.2.3. Estructura alternativa múltiple o anidada

Cuando alguna de las secuencias de instrucciones de la condición cierta o falsa contiene a su vez estructuras alternativas, se dice que están anidadas. En muchos problemas se tiene que escoger opciones entre múltiples casos. Por ejemplo, si queremos calcular valores de la función signo, definida por la expresión siguiente, debemos usar alternativas múltiples,

$$\text{signo}(x) = \begin{cases} +1 & \text{si } x > 0 \\ 0 & \text{si } x = 0 \\ -1 & \text{si } x < 0 \end{cases}$$



```
# Función signo(x)
x = float(input("Entra x: "))
if x < 0:
    signo = -1
else:
    if x == 0:
        signo = 0
    else:
        signo = 1
print('El signo de', x, '=', signo)
```

```
# Función signo(x), version 2
x = float(input("Entra x: "))
if x < 0:
    signo = -1
elif x == 0:
    signo = 0
else:
    signo = 1
print('El signo de', x, '=', signo)
```

Python incluye la sentencia `elif` que permite encadenar los `else` e `if` seguidos permitiendo una mayor legibilidad de estructuras anidadas. En el bloque de la derecha del ejemplo anterior, se muestra el uso de esta sentencia.

El programa siguiente convierte una calificación numérica a formato cualitativo, de texto, considerando el formato español de calificación universitaria.

```
print('Programa que convierte una nota numérica a cualitativa')
nota = float(input('Nota numérica: '))
if nota < 5:
    calif = 'Suspenso (SS)'
elif nota < 7:
    calif = 'Aprobado (AP)'
elif nota < 9:
    calif = 'Notable (NT)'
else:
    calif = 'Sobresaliente (SB)'
print('La nota', nota, 'equivale a', calif)
```

Este programa puede ser robusto a notas erróneas, como -3 o 15 introduciendo el código siguiente, en lugar de "if nota < 5:"

```
if (nota < 0) or (nota > 10):
    calif = 'Nota no válida'
elif nota < 5:
```

La estructura algorítmica alternativa múltiple o anidada será entonces de la forma

```
if condición1:
    secuencia_de_instrucciones_si_condicion1_cierta
elif condición2:
    secuencia_de_instrucciones_si_condicion2_cierta
elif condición3:
    secuencia_de_instrucciones_si_condicion3_cierta
...
else:
    secuencia_de_instrucciones_si_condiciones_anteriores_falsas
```

Esta estructura puede también prescindir del else final en caso de no ser necesario. Las estructuras alternativas múltiples tipo *switch* de C/C++ o *case* de Pascal no existen en Python. Su función se puede realizar con la estructura *if ... elif ... elif*, aunque para la realización de los típicos menús de usuario para escoger una opción, se puede utilizar datos estructurados tipo tuplas, listas o diccionarios, como los que se estudiarán en el tema 5.

### 3.3. Estructuras iterativas

Las estructuras o composiciones iterativas permiten repetir una instrucción o un bloque de instrucciones de manera automática. Las estructuras iterativas junto con las alternativas forman la base de la construcción de algoritmos y, de acuerdo al paradigma de la programación estructurada, permiten resolver cualquier problema computable. Las iteraciones para ejecutar el bloque de código varias veces se llaman también bucles (*loops*).

#### 3.3.1. Secuencia de datos

Una **secuencia de datos** se considera como una estructura de secuencia si en un algoritmo se puede: (i) acceder al primer elemento de la secuencia, (ii) reconocer el último elemento o su condición y (iii) acceder al elemento  $n+1$  a partir del elemento  $n$ .

Un ejemplo de secuencia de datos serían las notas de los alumnos de una asignatura con 50 alumnos. Las 3 acciones algorítmicas descritas previamente se pueden hacer.

Otro ejemplo de secuencia sería aquella dada por los términos de la expansión en serie de Taylor de la función del seno de un ángulo (en radianes),  $\text{seno}(x)$ :

$$\text{seno}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

Un algoritmo no podría calcular los infinitos términos, pero el último elemento de la secuencia de datos a procesar por un esquema iterativo sería: (opción 1) un término final fijado por el programador o (opción 2) un término cuyo valor absoluto cumpla la condición de ser menor que un error prefijado.

### 3.3.2. Esquemas iterativos

Los esquemas iterativos aplicados a realizar cálculos o utilizar expresiones de todo tipo (aritméticas, booleanas o relacionales) sobre una secuencia de datos suelen clasificarse como esquemas iterativos de búsqueda o de recorrido.

En un **esquema de recorrido** hay que tratar todos los elementos de la secuencia para realizar los cálculos necesarios o resolver el problema. Por ejemplo, si queremos hallar el valor medio de la nota de los 50 alumnos de la asignatura, debemos hacer un recorrido sobre toda la secuencia de notas. Las iteraciones de recorrido son llamadas también **iteraciones definidas**.

En un **esquema de búsqueda** no necesariamente hay que recorrer toda la secuencia de datos para resolver el problema. Por ejemplo, si necesitamos saber si hay alguna nota 10, o alguna nota mayor que 9, no hace falta recorrer la secuencia en su totalidad. Si se encuentra antes del final, se para. Este esquema se llama también **iteraciones indefinidas**.

### 3.3.3. Estructura iterativa while (mientras)

La sentencia *while* es la composición algorítmica iterativa por excelencia que sirve para cualquier esquema iterativo. Su implementación es similar en todos los lenguajes de programación.

Estructura iterativa while			
Pseudocódigo	Python	Pascal	C/C++
mientras condición hacer	while condición:	while condición do	while (condición)
Instrucciones	Instrucciones	begin	{
		Instrucciones;	Instrucciones
fin_mientras		end;	}

Tabla 10. Composición iterativa while en diversos lenguajes.

En la estructura while el bloque de la secuencia de instrucciones se repite siempre que la *condición* dada por el valor de una variable o expresión booleana sea cierta. Hay que recordar que en Python la secuencia de instrucciones dentro de la estructura algorítmica debe estar indentada.

Haremos un ejercicio que muestre la tabla de multiplicar del número introducido por el usuario. Esto es un esquema típico de recorrido:

```
# Tabla de multiplicar (del 1 al 10) del número introducido
n = int(input('Entra un número entero: '))
k = 1
while k <= 10:
    print(n, 'x', k, '=', n*k)
    k = k + 1
print('Hemos mostrado la tabla de multiplicar del', n)
```

El programa pide al usuario un número, inicializa la variable  $k$  en 1 y luego se ejecuta la iteración *while* 10 veces, mientras la condición  $k \leq 10$  es *True*. En la décima iteración la variable  $k$  toma el valor 11 y entonces la expresión booleana  $k \leq 10$  es *False* y sale del *while*, pasando el programa a la instrucción siguiente *print()*.

```
Entra un número entero: 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
Hemos mostrado la tabla de multiplicar del 7
```

El ejercicio siguiente muestra los primeros  $N$  números naturales en forma decreciente:

```
# Mostrar los N primeros números naturales en forma decreciente
N = int(input('Entra un número natural: '))
i = N
while i >= 1:
    print(i, end = ' ')
    i = i - 1
print('\nHemos mostrado', N, 'números naturales decreciendo')
```

```
Entra un número natural: 7
7 6 5 4 3 2 1
Hemos mostrado 7 números naturales decreciendo
```

A continuación se muestra un programa que calcula la suma de los  $n$  primeros números naturales.

```
# Calculo de la suma de los primeros n números
n = int(input('Entra un número natural: '))
i = 1
suma = 0
while i <= n:
    suma = suma + i
    i = i + 1
print('La suma de los números naturales hasta',n,'es', suma)
```

```
Entra un número natural: 100
La suma de los números naturales hasta 100 es 5050
```

El programa siguiente sigue un esquema típico de búsqueda. Queremos saber si un número natural es primo o no. Un número primo solo es divisible por 1 y por sí mismo, por lo que probamos los números entre 2 hasta el anterior a él mismo. Si es divisible por alguno entonces encontraremos un divisor y paramos la búsqueda. El número no será primo. Si llegamos al final y no hemos encontrado divisores, entonces sí es primo.

```
# Programa que dice si un número es primo o no
n = int(input('Entra un número natural: '))
d = 2
un_divisor = False          # Si hay un divisor el número NO es primo
while d < n and not un_divisor:
    if n % d == 0:           # Se busca algún divisor
        un_divisor = True
    d = d + 1
print('Es primo? ',not un_divisor)
```

```
Entra un número natural: 29
Es primo? True
```

Este programa no requiere buscar todos los divisores desde 2 hasta  $n-1$ . El lector puede averiguar hasta dónde hace falta buscar posibles divisores del número que deseamos saber si es primo o no. La condición  $d < n$  del *while* puede cambiarse para detener la búsqueda antes.

Los ejemplos de algoritmos clásicos presentados en el tema 1 son también esquemas iterativos de búsqueda. El algoritmo de Euclides para hallar el MCD de dos números naturales se puede programar en Python como

```
# Algoritmo de Euclides en Python
print('Cálculo del MCD de 2 números usando el algoritmo de Euclides')
a = int(input('Entra un número natural: '))
b = int(input('Entra otro: '))
aa, bb = a, b
while a != b:
    if a > b:
        a = a-b
    else:
        b = b-a
print('El MCD entre',aa,'y',bb, 'es', a)
```

```
Cálculo del MCD de 2 números usando el algoritmo de Euclides
Entra un número natural: 28
Entra otro: 12
El MCD entre 28 y 12 es 4
```

En el programa de Euclides hemos utilizado la asignación múltiple para guardar los valores de las variables  $a$  y  $b$  en  $aa$  y  $bb$ , de forma que no se pierdan y mostrarlos en el `print()` final.

El siguiente ejemplo muestra la estructura iterativa de cálculo aproximado de la raíz cuadrada de un número, de acuerdo al método descrito por Herón de Alejandría, presentado previamente. Es un típico esquema de búsqueda (iteración indefinida) de cuándo hay que parar. Se para cuando el valor absoluto de la diferencia entre la raíz cuadrada propuesta al cuadrado y el número buscado sea menor o igual que una tolerancia de error (*epsilon*) dado. Si no se dispusiera de la función interna `abs()`, la condición lógica del `while` se escribiría: ( $dif > epsilon$  or  $dif < -epsilon$ ).

```
# Algoritmo de Herón. Cálculo de la raíz cuadrada
print('Cálculo de la raíz cuadrada de un número (algoritmo de Herón)')
x = float(input('Entra un número para hallar su raíz cuadrada: '))
g = x/2 # El valor inicial g (guess), g0, puede ser x/2
epsilon = 1e-7 # Tolerancia de la aproximación a la raíz cuadrada
dif = g*g - x # Diferencia entra el valor de raíz g0 propuesto y el real
while abs(dif) > epsilon:
    g = (g + x/g)/2
    dif = g*g - x
print('La raíz cuadrada (aproximada) de',x,'es', g)
```

Si quisiéramos rastrear cuál es el valor aproximado de la raíz del número en cada iteración, podemos incluir una instrucción al final del bloque dentro del while de la forma

```
while abs(dif) > epsilon
    g = (g + x/g)/2
    dif = g*g - x
    print('Raíz de',x,'aprox:',g)
print('La raíz cuadrada (aproximada) de',x,'es', g)
```

Se puede probar hallar la raíz de 9 y observar el error de aproximación. Intente reducir el valor *epsilon* a  $1e-20$  ( $10^{-20}$ ) y observar el resultado. Este programa, en la práctica, no es necesario pues Python (y la mayoría de lenguajes) dispone de la función *sqrt()* en el módulo de funciones matemáticas *math*, que se invoca tal como hicimos con anterioridad con el valor pi:

```
from math import sqrt
y = sqrt(x)
```

El programa siguiente realiza un juego típico de adivinar un número oculto, que es generado aleatoriamente por el computador. La solución para el jugador es dividir la búsqueda en dos conjuntos y así ir acotando el conjunto de números hasta encontrar la solución. Se suele llamar a esta estrategia **búsqueda binaria o dicotómica**. Se ha incluido en el programa el límite de 100 para el número natural a ser adivinado. En este caso 100 es menor que  $2^7$  por lo que con 7 intentos será suficiente para adivinar el número. Se hace uso del módulo de funciones aleatorias *random*. En particular, la función *randint(a, b)* que genera un número entero aleatorio en el rango  $[a, b]$  ambos inclusive.

```
# Juego de adivinar número oculto (generado aleatoriamente)
from random import randint
oculto = randint(1,100)
print('Se ha generado un número del 1 al 100')
print('Adivínalo en máximo 7 intentos, ')
intentos = 1
x = int(input('Adivina el número: '))
while (x != oculto) and (intentos < 7):
    if x > oculto:
        x = int(input('Prueba uno menor(quedan '+str(7-intentos)+' intentos): '))
    else:
        x = int(input('Prueba uno mayor(quedan '+str(7-intentos)+' intentos): '))
    intentos = intentos + 1
if x == oculto:
    print('Muy bien! lo has conseguido en', intentos, 'intentos')
else:
    print('Lástima, el número oculto era el', oculto)
```

Se propone modificar este programa para que se adivine un número del 1 al 1000 o, en general, del 1 al N.

### Estructura *while* para asegurar condiciones de entrada de datos

Las raíces cuadradas se pueden calcular solo para números positivos. Para hallar raíces de números negativos se tiene que usar números imaginarios. El programa de Herón realizado entrará en iteraciones infinitas si probamos un número negativo (si es menor que *-epsilon*). Introducimos un esquema muy usado para asegurar requisitos de entrada en programas informáticos. En este caso se debe cerciorar que el número introducido por el usuario sea positivo. El siguiente código, que se debe incluir al comienzo del programa de Herón, hace que la estructura *while* capte un valor erróneo (negativo) y no salga del bucle hasta que se introduzca uno correcto (positivo). Este esquema puede ampliarse a múltiples casos, cambiando la condición lógica del *while*.

```
x = float(input('Entra un número para hallar su raíz cuadrada: '))
while x < 0:
    x = float(input('El número debe ser positivo!!: '))
```

### 3.3.4. Estructura iterativa *for* (desde - hasta, para - en)

Cuando en un programa informático se requiere hacer un conjunto de cálculos sobre una serie de datos, es decir, hacer un esquema de recorrido (iteración definida) o aplicar el bloque de instrucciones sobre un grupo de elementos (números, string, etc.) es preferible utilizar la sentencia **for**. En Python



el *for* es una estructura iterativa particular si la comparamos con las equivalentes en otros lenguajes de programación.

Pseudocódigo	Python	Pascal	C/C++
desde inicio a fin hacer	for item in secuencia :	for var:= ini to fin do	for (ini; cond; fin)
Instrucciones	Instrucciones	begin	{
		Instrucciones;	Instrucciones
fin_desde		end;	}

Tabla 11. Composición iterativa *for* en diversos lenguajes.

En Python el bucle *for* se puede leer como “para (*for*) cada elemento (*ítem*) de (*in*) la lista (secuencia) ejecutar las instrucciones del bloque indentado”. Mostraremos un par de ejemplos para presentar esta estructura iterativa:

```
# Saludo de cumpleaños
for nombre in ['Leo', 'Ronaldo', 'Andrés', 'Sergio']:
    print('Feliz cumpleaños', nombre)
print('Ya hemos saludado a los amigos')
```

```
# Potencias de 2
for posicion in [1, 2, 3, 4, 5, 6, 7, 8]:
    print('Peso del bit', posicion, '=', 2**(posicion-1))
```

En estos ejemplos se puede observar que la estructura *for* recorre los elementos de la secuencia de datos que está entre corchetes y este recorrido lo hace a través de la variable que toma el valor del elemento en cada iteración. Las variables que se refieren a cada elemento son nombre y posición, respectivamente. La iteración se realiza tantas veces como elementos tenga la secuencia, que en estos ejemplos son de 4 y 8 elementos, respectivamente.

La secuencia de datos entre corchetes forma una **lista** en Python. Pero la sentencia *for* puede también recorrer otra secuencia de datos en tipos de datos compuestos, como los *string* u otros que presentaremos en el tema 5 (tuplas, diccionarios, etc.). Las listas son datos estructurados que también se analizarán en más detalle en el tema 5. Sin embargo, dado que son muy usadas en la estructura iterativa *for* de Python, las introduciremos en esta sección.

Las **listas** son una secuencia ordenada de valores a los cuales se accede a través de un índice que indica en qué posición en la lista se encuentra un elemento. Se caracterizan por el uso de corchetes

para delimitar la secuencia y de las comas como separador de los elementos. Las listas pueden contener datos simples numéricos y booleanos o datos compuestos como strings. También pueden tener datos heterogéneos de diferente tipo. Se dice que las listas son dinámicas o mutables porque sus elementos pueden cambiar de valor, eliminarse o se pueden añadir nuevos valores. Así, la estructura de datos entre corchetes de los primeros ejemplos con *for* corresponde a listas.

Para acceder a las listas lo haremos a través del índice que señala la posición del elemento. Las secuencias en el mundo real suelen comenzar por el uno (enero = mes 1, lunes = día 1, escena 1, fila 1 de la matriz, etc.). Sin embargo los lenguajes informáticos suelen comenzar por el cero. En la lista con los nombres del primer ejemplo de *for* tenemos que el elemento 'Leo' se accede (se indexa) con el valor 0, aunque está en la primera posición de la lista. El elemento 'Sergio', que es el cuarto de la lista, se accede con el valor de índice 3.

```
>>> nombres = ['Leo', 'Ronaldo', 'Andrés', 'Sergio']
>>> nombres[0]
'Leo'
>>> Nom4 = nombres[3]
>>> print(Nom4)
Sergio
```

Existe en Python una acción muy útil para generar secuencias de valores enteros que sirvan para recorrer los bucles *for* de Python. Se trata del tipo de dato **range()**<sup>12</sup>. Por ejemplo

```
for elemento in range(8):
    print(elemento, end = ' ')
```

Escribe los valores

```
0 1 2 3 4 5 6 7
```

De los ocho elementos que recorre la variable *elemento*, el primero es el 0 y el último el 7. Si quisiéramos emular la lista [1, 2, 3, 4, 5, 6, 7, 8] del ejemplo Potencia de 2 debemos usar *range(1, 9)*. Comienza en 1 (en lugar de 0) y termina en 9-1:

---

12 En las versiones 2.x de Python, *range()* es una función que devuelve una lista. El concepto ha cambiado en la versión 3, donde pasa a ser un tipo de dato *range*, que viene a ser una secuencia de enteros usada típicamente para recorrer bucles *for*.

```
# Potencias de 2
for posicion in range(1, 9):
    print('Peso del bit', posicion, '=', 2**(posicion-1))
```

La sintaxis general de range() es (lo que está entre corchetes es opcional):

*range([inicio,] final [, paso])*

Por ejemplo,

```
range(5)           # → [0, 1, 2, 3, 4]   No se alcanza el 5 sino
5-1
range(2,6)         # → [2, 3, 4, 5]     Comienza en 2 hasta 6-1
range(0,10,3)      # → [0 3 6 9]        De 0 hasta 9 de en pasos
de 3
range(0,-10,-3)    # → [0 -3 -6 -9]     De 0 hasta -9 de en pasos
de -3
```

Las composiciones *for* permiten realizar de forma más compacta las estructuras iterativas de recorrido o definidas, como los tres primeros ejemplos presentados con la composición *while*:

```
# Tabla de multiplicar (del 1 al 10) del número introducido (versión for)
n = int(input('Entra un número entero: '))
for k in range(1, 11):
    print(n, 'x', k, '=', n*k)
print('Hemos mostrado la tabla de multiplicar del', n)
```

```
# Mostrar los N primeros números naturales en forma descendiente (for)
N = int(input('Entra un número natural: '))
for i in range(N, 0, -1):
    print(i, end = ' ')
print('\nHemos mostrado', N, 'números naturales decreciendo')
```

```
# Calculo de la suma de los primeros n números (versión for)
n = int(input('Entra un número natural: '))
suma = 0
for i in range(1, n+1):
    suma = suma + i
print('La suma de los números naturales hasta',n,'es', suma)
```

Hemos comentado que la sentencia *for* recorre una secuencia de datos como las listas, *range* y *strings*. Veamos un simple ejemplo con *for* recorriendo un *string*:

```
# for en secuencia tipo string
for i in 'Hola':
    print('Imprimo:',i)
```

```
Imprimo: H
Imprimo: o
Imprimo: l
Imprimo: a
```

La variable *i* toma el valor de cada elemento del string '*Hola*' durante cada iteración. Al acabar el *for*, en la cuarta iteración, la variable *i* se queda con el valor '*a*'. ¡Hay que tener en cuenta este detalle si se usa la variable dentro del bucle o después!

### 3.3.5. Bucles para efectuar sumatorias

La suma de una secuencia de números es una operación muy frecuente en procesamiento de datos matemáticos. La operación sumatoria, denotada por la mayúscula griega  $\Sigma$  puede ser realizada por una estructura iterativa de recorrido o de búsqueda, en caso de que se conozcan todos los términos a sumar o no. Es decir, la podemos realizar con la estructura *for* para iteraciones definidas o *while* para bucles indefinidos.

En forma general, la serie o sumatoria de términos,  $f_i$ , que dependan de la ubicación del término  $i$ , se representan por

$$S = \sum_{i=m}^N f_i = f_m + f_{m+1} + f_{m+2} + \cdots + f_{N-1} + f_N$$

El ejemplo visto anteriormente de sumar los primeros  $n$  números es típico de sumatorias. Por ejemplo, si queremos hallar la sumatoria de términos de la serie convergente que representa la paradoja de la dicotomía de Zenón de Elea, podemos usar el esquema clásico de sumatorias en lenguajes de programación. Se inicializa la variable suma en 0 y dentro del bucle se van añadiendo términos a esta variable por cada iteración.

$$suma = \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \dots$$

Como la sumatoria es de infinitos términos, el programa puede calcular un número de términos dados o, se le puede introducir un valor epsilon con el error aceptado, y allí parar.

```
# Paradoja de la dicotimía de Zenon (términos definidos)
n = int(input('Entra el número de términos a calcular: '))
suma = 0
for i in range(1, n+1):
    suma = suma + 1/2**i
print('El valor de la sumatoria es', suma)
```

```
# Paradoja de la dicotimía de Zenon (acaba con condición)
eps = float(input('Entra el valor del término para acabar (ej. 0.001): '))
suma = 0
term = 1
i = 1
while term >= eps:
    term = 1/2**i
    suma = suma + term
    i = i + 1
print('El valor de la sumatoria es', suma)
```

### 3.3.6. Sentencias break y continue

Existen en Python sentencias que permiten salir durante una iteración o definitivamente de una estructura iterativa.

La sentencia **break** hace que se salga inmediatamente del bucle al ser accedida, es decir, se sale del bloque de instrucciones de la iteración *for* o *while*. Esto implica que aquellas instrucciones de la secuencia que están a continuación del break ya no se ejecutan y se sale de la estructura iterativa.

En cambio la sentencia **continue** hace que se salten las instrucciones dentro del bloque de la estructura iterativa, pero solo en la iteración actual. Es decir, se continuará con la siguiente iteración dada por el *for* o el *while*.

Estas sentencias deben utilizarse con moderación, ya que introduce una excepción a la lógica de control normal del bucle.

## Tema 4.

# Programación modular. Funciones, procedimientos y parámetros

Hemos presentado ejemplos de programas sencillos que sirven para hacer un cálculo específico o resolver un problema simple. Muchos programas informáticos más complejos requieren realizar subprogramas y módulos de subprogramas que puedan ser reutilizados y mejoren la manera de diseñar un programa. Desde los años 1970 se idearon paradigmas como la programación estructurada que produce códigos o programas más fáciles de leer, depurar y actualizar, y paradigmas como el **diseño modular de programas** y los esquemas **top-down** (de arriba a abajo) y **bottom-up** (de abajo a arriba) como estrategias para desarrollar programas complejos o sistemas de software.

El paradigma de la **programación modular** surgió para hacer un programa más claro, legible, menos complejo y, principalmente, para reutilizar subprogramas. De esta forma, al programador se le facilita modificar y corregir los códigos por separado y también crear una librería de subprogramas utilizables por otros programas. La idea de agrupar un conjunto de subprogramas en módulos se implantó en los lenguajes Modula, como Modula, Modula-3, donde se accede a un subprograma de un módulo con el punto "." después del nombre del módulo. De igual forma, el uso del punto se extendió para acceder a un campo de un record (registro) o a un campo o método de un objeto. Esta idea se siguió en lenguajes C++, Java y Python.

Los esquemas *top-down* y *bottom-up* siguen la misma filosofía de la programación modular. Se parte de un programa principal que llama o utiliza otros subprogramas, que a su vez pueden utilizar otros subprogramas (*top-down*). Un problema complejo se subdivide varias veces hasta resolver estructuras algorítmicas más simples. El trabajo de programar se puede dividir también en varios programadores que aporten diferentes subprogramas u objetos. Luego, los elementos básicos del programa se desarrollan en detalle y se enlazan a los subprogramas y al programa principal. En la metodología *bottom-up*, complementaria a la *top-down*, se consulta la lista de componentes para fomentar la reutilización. Se puede incluir el diseño de objetos que, dentro del paradigma de la programación orientada a objetos (OO), pueden contener atributos o campos y códigos de programa que realizan una tarea (métodos).

Los **subprogramas** (llamados también **subrutinas**) se refieren al conjunto de instrucciones que están separadas del programa principal y realizan cálculos o tareas. Se refieren, generalmente, a las **funciones** en los diversos lenguajes de programación, incluyendo Python. Pero también hay subprogramas que realizan tareas sin devolver resultados (como sí lo hacen las típicas funciones matemáticas), que son llamados en algunos lenguajes de programación *procedures* (procedimientos). Modula-3, Pascal y ADA incorporan los *procedures* como subprogramas que realizan acciones, además de las clásicas funciones.

En Python se pueden guardar un grupo de funciones en un módulo. Los **módulos** se pueden interpretar como una biblioteca o una caja de herramientas de funciones de una especialidad dada. En otros lenguajes de programación se utilizan otros términos para este concepto de módulo. Por ejemplo en Dart (lenguaje de programación desarrollado por Google en 2011) se usa el término *library* (biblioteca) que se distribuyen como **package** (paquetes), en Go (otro lenguaje de programación de Google) y Java se usa el término **package**, en Pascal el vocablo **unit** y en Matlab el término **toolbox**.

Los módulos de Python que se guardan en una carpeta que forman un package. Es decir, los package en Python son una colección de módulos.

## 4.1. Uso de funciones. Funciones internas y de módulos

En diversos ejemplos de programas hemos utilizado funciones internas y funciones del módulo de matemáticas (*math*) y del que genera valores aleatorios (*random*). Entre las funciones ya aplicadas figuran:



Función interna	Devuelve
type	tipo de dato
id	Identidad o ubicación de memoria
bin	string del binario equivalente al entero dado
int, float, str	entero, real, string del valor dado
input	string del texto leído del teclado
print	Valores a imprimir en pantalla
abs	valor absoluto de un número
round	redondea un real a los decimales especificados

Tabla 12. Ejemplos de funciones internas en Python.

Módulo math	
pi	valor de $\pi$
sqrt	raíz cuadrada de un número

Tabla 13. Ejemplo de constante y función del módulo de matemáticas (math) de Python.

Módulo random	
randint	número entero aleatorio en el rango dado

Tabla 14. Ejemplo de función que devuelve valor aleatorio del módulo Random de Python.

En matemáticas una función devuelve un resultado dependiendo de cada valor de su(s) variable(s) independiente(s). Por ejemplo:  $f(x) = 3x^2 + 1$  calcula, para  $x = 2$ , el valor  $f(2) = 13$ ;  $f_2(x, y) = x^2 + y^2$  calcula  $f(2,2) = 8$ ;  $f(x) = \sqrt{x}$  calcula  $f(9) = 3$ . Las funciones que devuelven resultados en los lenguajes de programación se comportan de manera similar.

En el contexto de lenguajes informáticos una **función** es una instrucción o un bloque de instrucciones que realizan un cálculo o una tarea, y que tiene un identificador como nombre (identificador igual que las variables). De las funciones usadas previamente sus nombres (identificadores) son type, int, abs, sqrt, round, etc. En el primer ejemplo de tipo de datos ya utilizamos el **llamado a una función** (también se dice **invocar** la función):

```
>>> type(7)
<class 'int'>
```

El nombre de la función es `type` y, en este caso, el valor 7 es su argumento. Los **argumentos** son los valores que le pasamos a las funciones. Ejemplo:

```
from math import sqrt
x = abs(-9) + 3
y = sqrt(x)
print('Raíz cuadrada de',x, '=' ,y)
```

Primero se debe importar la función `sqrt` del módulo `math` (`abs` no hace falta importarla pues es función interna). Lo que está entre paréntesis de `abs` y `sqrt` (el valor -9 y la variable `x`, respectivamente) es el argumento de la función. Estas funciones **devuelven** un valor (*return value*) luego de ser llamadas. La función `abs` es llamada dentro de una expresión aritmética, el valor que devuelve pasa a reemplazarla en la expresión y se suma a 3. Luego, este nuevo valor se asigna a la variable `x`. Hay que tener cuidado que el tipo de dato que devuelva la función sea compatible con la operación que se vaya a hacer. Los argumentos de las funciones pueden ser expresiones, incluso expresiones que incluyen llamadas a otras con funciones, como se observa a continuación,

```
y = sqrt(4 + x**2)
z = sqrt(abs(-40))
```

Las funciones `round` o `randint` requieren de dos argumentos y devuelven un valor:

```
from random import randint
n = randint(10, 100)/3.2567
print(n, round(n,2))
```

La lista de funciones internas de la versión 3 de Python se puede consultar en la web de Python Software foundation (<https://docs.python.org/3.3/library/functions.html>).

A continuación se resume una lista de algunas funciones de los módulos `math` (matemáticas), y `random` (de valores aleatorios)<sup>13</sup>.

13 La lista completa de módulos se puede consultar en <https://docs.python.org/3.3/library/index.html>

Módulo math	devuelve
pi	valor $\pi = 3.141592653589793$
e	valor $e = 2.718281828459045$
ceil(x)	entero mayor que x, hacia $\infty$
floor(x)	entero menor que x, hacia $-\infty$
trunc(x)	redondea hacia 0
factorial(x)	x!
exp(x)	$e^{**}x$
log(x)	logaritmo natural (base e), $\ln(x)$
log10(x)	logaritmo base 10
sqrt(x)	raíz cuadrada de x
sin(x), cos(x), tan(x)	seno, coseno, tangente de x
degrees(x)	ángulo x de radianes a grados
radians(x)	ángulo x de grados a radianes

Tabla 15. Constantes y funciones de uso frecuente del módulo de matemáticas (math) de Python.

Módulo random	devuelve aleatorio
randint(a,b)	entero en el rango [a, b]
randrange(a,b,paso)	de range(a, b, paso)
shuffle(s)	baraja la secuencia s
choice(s)	escogido de la secuencia s
random()	real en el rango [0.0 1.0)
seed()	inicializa generador aleatorios

Tabla 16. Funciones de uso frecuente del módulo Random (aleatorio) de Python.

El uso de las funciones de los módulos se puede hacer de 2 maneras. Hasta ahora hemos importado la función o valor que necesitamos y la aplicamos directamente en la instrucción. También podemos importar múltiples funciones del módulo:

```
from math import sqrt, log10
x = sqrt(10)
dB = log10(x/2)
```

Alternativamente, se puede importar el módulo `math` y utilizar las funciones del módulo separadas con punto:

```
import math
x = math.sqrt(10)
dB = math.log10(x/2)
```

En este caso, se puede abreviar el nombre del módulo:

```
import math as m
x = m.sqrt(10)
dB = m.log10(x/2)
```

## 4.2. Funciones y procedimientos

Hemos comentado que los subprogramas o subrutinas en la mayoría de los lenguajes (y en Python) se refieren a las funciones. En algunos lenguajes de programación, como Modula-3, Pascal y ADA, se distinguen aquellos subprogramas que realizan cálculos y devuelven resultados (como las funciones matemáticas), que se llaman **funciones**, de aquellos que realizan tareas o acciones sin devolver resultados. Éstos se llaman **procedimientos** (*procedures*). Algunos autores (Kaelbling et al., 2011) llaman *procedures* (*procedimientos*) a todo tipo de subprograma o subrutina, pero el término de función es el más utilizado.

En Python, se pueden diseñar funciones que hacen cálculos y devuelve un resultado (la típica de matemáticas), como la mayoría de funciones que hemos usado. Por ejemplo, *log10(100)* devuelve 2.0. Pero hay funciones, como *shuffle(s)* del módulo *random*, que baraja o pone en posiciones aleatorias los elementos de la secuencia *s*. No devuelve resultado alguno, sino que realiza una acción sobre un objeto y éste queda modificado. Este tipo de función equivale a un *procedure* de otros lenguajes.

Luego de la serie de libros usados como textos en el MIT<sup>14</sup> para los cursos introductorios de **Computer Science** (Introducción a la informática), como “How to think like computer scientist” y “Think Python” (Downey, E.K., A. B., Elkner, J. & Meyers, C., 2002; Wentworth, P., Elkner, J., Downey, A. B., & Meyers, C., 2012; Downey, E.K, 2015) se han extendido los términos “**funciones productivas**” (*fruitful function*) para llamar a las funciones que devuelven resultados y “**funciones nulas o estériles**” (*void function*) a las que no devuelven resultados, equivalentes a los *procedures* de Pascal. Lenguajes como C/C++ y Java también usan las *void functions* para los procedimientos que no devuelven resultados.

<sup>14</sup> El Massachusetts Institute of Technology (MIT) dispone de numerosos materiales de enseñanza de acceso libre en la web MIT OpenCourseWare (OCW): <http://ocw.mit.edu/index.htm>

Python, en realidad, siempre devuelve un resultado. En las funciones productivas devuelve un valor de algún tipo de dato (entero, real, booleano, lista, etc.). Cuando la función es nula o *void*, Python devuelve el valor *None*.

## 4.3. Diseño de funciones

Hasta ahora hemos usado funciones internas o de módulos de Python que ya están diseñadas. Las hemos llamado en una instrucción para que nos devuelva un valor o ejecute un procedimiento. Si queremos diseñar funciones debemos definirlas. En Python la **definición de una función** es de la forma:

```
def nombre_funcion(parametros):  
    cuerpo_de_la_funcion
```

Se usa la palabra reservada (keyword) de Python *def* para indicar que en esta línea comienza la definición de la función, luego ponemos el nombre de la función con un identificador válido y entre paréntesis los **parámetros** de entrada de la función. Los parámetros se relacionan con los valores que le son pasados como argumentos. El cuerpo del subprograma está indentado o tabulado (se indenta generalmente con cuatro espacios en blanco). Al volver a la columna 1 del editor se termina el bloque de código que define la función y volvemos al programa principal. Conviene definir todas las funciones que use nuestro programa al comienzo de éste. Sin embargo, Python solo requiere que la función esté definida antes de que sea usada. Veamos un ejemplo donde se diseña una función que devuelva la ganancia en decibelios (dB) de un amplificador. Sus parámetros de entrada serán los voltajes de entrada (x) y salida (y) del amplificador:

```
from math import log10  
def ganancia_dB(x, y):  
    """  
        Calcula ganancia en dB:  $20\log(y/x)$   
    """  
    gain = y/x  
    dB = 20*log10(gain)  
    return dB  
Vi = 10  
Vo = 1000  
GdB = ganancia_dB(Vi, Vo)  
print('Ganancia =', GdB, 'dB')
```

El nombre de la función es *ganancia\_dB*, sus parámetros son *x* e *y*, la palabra reservada *return* indica que la función es productiva (devuelve un resultado, en este caso el valor de dB) y el string entre comillas triples permite documentar la función (este estilo de documentar se llama *docstring* en

Python). Al devolver un resultado de tipo dato simple, como *float* en esta función, lo pudiéramos usar en una expresión aritmética. Al ejecutar el programa tenemos:

```
>>>
Ganancia = 40.0 dB
```

El programa principal envía los valores de *Vi* y *Vo* a la función. Estos valores de *Vi* y *Vo* son los **argumentos** de la función. Los **parámetros** formales *x* e *y* de la función reciben los valores de los argumentos (como si fuera una asignación) y entran a la función. Hay que notar que se debe guardar el orden de los parámetros *Vi* → *x*, *Vo* → *y*. En esta función las variables *gain* y *db* son **variables locales**, es decir, las usa solo esta función y al acabar su ejecución se borran. Los parámetros *x* e *y* también actúan como variables locales.

El estilo *docstring* que incluimos en la función sirve para documentar en el Shell de Python lo que hace la función:

```
>>> help(ganancia_dB)
Help on function ganancia_dB in module __main__:

ganancia_dB(x, y)
    Calcula ganancia en dB: 20log(y/x)
```

Las funciones pueden incluir también **variables globales**. Éstas se declaran dentro de la función precedidas por la palabra reservada *global*. Si una variable del programa principal tiene el mismo identificador que la definida como *global* dentro de una función, entonces será modificada por cualquier asignación de la variable global. El uso de las variables globales se debe hacer con moderación. Aunque intentaremos evitarlas en nuestros programas, presentamos un ejemplo del uso de estas variables:

```
def f(x):
    global a
    a = 3
    return 3*x

a = 7
print('Función:',f(4))
print('Valor de la variable global a:',a)
```

```
Función: 12  
Valor de la variable global a: 3  
>>>
```

### 4.3.1. Paso de parámetros entre el programa y las funciones

Los **parámetros** de las funciones en Python se definen, como en el caso de las variables, con un identificador válido. En otros lenguajes, en los que hay que definir el tipo de variables antes de usarlas, habría que definir también el tipo de parámetro. Este no es el caso en Python, el tipo de parámetro será el mismo que el tipo de argumento que se envía del programa. En este tema diseñamos funciones que usan solo parámetros de tipos de datos simples (*int*, *float*, *bool*) o strings. Estos tipos de datos no permiten que alguna acción modifique sus valores (es decir son inmutables), solo cambiarían sus valores si la variable es asignada nuevamente. Esto quiere decir que el paso de parámetros es por valor siempre en este tipo de datos. Un **paso de parámetros por valor** significa que el valor del argumento se copia al parámetro a la función, pero si el parámetro es modificado dentro de la función, su valor no se referencia a la variable del argumento.

En el paso de los parámetros el **orden es importante** (ver excepciones en la sección de *keyword arguments*). En el ejemplo anterior, que sintetizaremos sin variables locales,

```
from math import log10  
def ganancia_dB(x, y):  
    """  
    Calcula ganancia en dB: 20log(y/x)  
    """  
    return 20*log10(y/x)  
y = 10  
x = 1000  
print('Ganancia =', ganancia_dB(y, x), 'dB')
```

se observa que los argumentos, llamados ahora y (en lugar de Vi) y x (en lugar de Vo), ocupan la primera y segunda posición, respectivamente, del llamado a la función dentro del print(). El primer argumento (y) que vale 10 se envía al primer parámetro de la función (x) y el segundo argumento (x) que vale 100 se envía al segundo parámetro (y). Lo importante es la posición y no los nombres de las variables y parámetros. El programa devuelve el mismo valor 40.0 dB.

El paso de parámetros por referencia lo presentaremos al analizar tipos de datos mutables en el próximo tema.

### 4.3.2. Funciones productivas y funciones nulas (procedimientos)

El siguiente ejemplo muestra los dos tipos de subprogramas típicos en lenguajes de programación: funciones (productivas) y procedimientos (funciones *void* o nulas):

```
def f1(x):  
    return (x/2 + 1)  
def f2(x):  
    print(x/2 + 1)
```

La función *f1()* devuelve un resultado y se puede usar como una función matemática en una expresión aritmética, pero, cuidado, *f2()* no es una función productiva, es un procedimiento o *void function*. Miremos estos resultados al ser llamada:

```
>>> f1(5)  
3.5  
>>> f2(5)  
3.5  
>>> print(f1(10))  
6.0  
>>> print(f2(10))  
6.0  
None  
>>> y = 4 + f1(6)      # f1(6) devuelve 4, y se le asigna el valor  
8  
>>> y = 4 + f2(6)      # f2(6) muestra el valor 4; pero errónea  
expresión  
4.0  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    y = 4 + f2(6)  
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Al llamar la función *f1()* en el primer *print()*, *f1(10)* devuelve el valor 6.0 y la función lo muestra, pero al llamar *f2(10)* dentro del segundo *print()*, esta función en su cuerpo de instrucciones realiza la acción de imprimir la expresión de valor 6.0 y, al acabar sin *return*, devuelve *None*, que también es mostrado por este *print()*. Luego, la instrucción *y = 4 + f1(6)* llama la función *f1()* con argumento 6, la función devuelve 4.0 que se suma a 4 y a la variable *y* se le asigna el valor 8.0. La instrucción *y = 4 + f2(6)*, sin embargo produce un error pues *f2()* es una función void o procedimiento. Se realiza la acción de mostrar 4.0, pero al devolver *f2()* un *None* la expresión aritmética es incorrecta.



### 4.3.3. Valores de argumentos por omisión (default)

Al diseñar funciones se puede prever que un argumento tenga un valor por omisión si el usuario lo omite al llamar la función. Por ejemplo, la función *ganancia\_dB* puede considerar que el usuario introduzca o no el valor de *y*:

```
from math import log10
def ganancia_dB(x, y=100):
    """
    Calcula ganancia en dB: 20log(y/x)
    """
    return 20*log10(y/x)
Vi = 10
print('Ganancia =', ganancia_dB(Vi), 'dB')
print('Ganancia =', ganancia_dB(Vi+5, 50), 'dB')
```

```
>>>
Ganancia = 20.0 dB
Ganancia = 10.457574905606752 dB
```

En el primer *print()*, la llamada a la función solo envía un argumento (*Vi* que vale 10), el segundo argumento se toma el valor por omisión (*y=100*). En la segunda llamada se usan los dos parámetros de la función. Se puede notar que en esta llamada el primer argumento es el valor de la expresión *Vi+5* (15).

### 4.3.4. Argumentos de palabra clave (keyword arguments)

Cuando una función tenga varios parámetros con valores de argumento por omisión, éstos se pueden omitir pero también pueden ser llamados en cualquier orden, siempre que se use el nombre del parámetro y su argumento asignado. Veamos el ejemplo siguiente,

```
def fkey(x, y=1, z=1, w=0):
    return (x + y)*(z + w)

print(fkey(2))                                # x=2, y=1, z=1, w=0
print(fkey(2, w = 4))                         # x=2, y=1, z=1, w=4
print(fkey(1, z = 2, y = 3))                 # x=1, y=3, z=2, w=0
```

```
>>>  
3  
15  
8
```

El primer *print* muestra lo que retorna la función *fkey()*, que será  $(2 + 1) * (1 + 0)$ . Los otros *print* muestran llamadas a la función con argumentos *keyword*. Los argumentos con el nombre del parámetro y su valor, del grupo de parámetros con valores por omisión, se colocan sin importar el orden en que se introducen.

## 4.4. Recursividad

Una estructura recursiva puede contener otra del mismo tipo, es decir se integra por partes de sí misma. La palabra **recursivo** se origina del latín “*recurrere*”, que significa “repetirse” o “correr hacia atrás”. Por ejemplo, un objeto fractal se caracteriza por su estructura básica que se repite a diferentes escalas. Se dice que tiene una estructura auto-similar.

En informática, **recursión** es un método de programar en la cual una función se llama a sí misma una o más veces en el cuerpo de la función. La condición más importante es que función acabe y no se quede recurriendo interminablemente.

El ejemplo más claro de una función recursiva es cálculo del factorial (denotado por el símbolo!) de un número natural. El factorial se define en términos de sí mismo:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n - 1)! & \text{si } n > 0 \end{cases}$$

Por ejemplo 4! Es 4 veces 3!, que a su vez es 3 veces 2!, que a su vez es 2 veces uno, que es 1 vez 0!, que es 1. Esto resulta en: 4! Es igual a 4 por 3 por 2 por 1 y por 1, que es 24.

Si diseñamos una función que calcule el factorial usando una composición iterativa de recorrido puede ser de la forma:

```
def factorial(n):
    if n == 0 :
        result = 1
    else:
        result = 1
        for i in range(1, n+1):
            result = result*i
    return result
# programa ppal
x = int(input('Dame un entero: '))
r = factorial(x)
print('El factorial de ', x, ' es ', r)
```

```
Dame un entero: 4
El factorial de 4 es 24
```

La función factorial de forma recursiva es mucho más simple:

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

Esta función puede resultar poco elegante de acuerdo a los principios clásicos de la programación estructurada, que indica que una función debe tener un solo punto de salida. Es común encontrar funciones con más de un punto de salida. Para resolver esta controversia modificamos esta última función con una bien estructurada:

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

Otra típica secuencia que puede ser calculada de forma recursiva es la formada por los números de Fibonacci<sup>15</sup>: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... (la versión moderna suele incluir también el 0 al comienzo de la secuencia). Los números de Fibonacci se pueden definir como

$$Fibonacci_1 = 1$$

$$Fibonacci_2 = 1$$

$$Fibonacci_k = Fibonacci_{k-2} + Fibonacci_{k-1} \quad (K > 2)$$

Fibonacci	1	1	2	3	5	8	13	21	34	55	89	144	...
k	1	2	3	4	5	6	7	8	9	10	11	12	...

Tabla 17. Números de Fibonacci y su posición k en la secuencia.

Una función recursiva para calcular los números es:

```
def fibonacci(k):
    if k == 1 or k == 2:
        result = 1
    else:
        result = fibonacci(k-1) + fibonacci(k-2)
    return result
```

## 4.5. Módulos: integrar funciones en una biblioteca

Las funciones que hemos definido se pueden guardar en un módulo para ser reutilizadas cuando queramos. Si tenemos un gran número de funciones lo usual es crear varios módulos con funciones del mismo tema. Por ejemplo un módulo de nuestras funciones para el procesamiento de señales, otro para el procesamiento de imágenes, otro para procesar audio, etc. La colección de módulos se puede guardar en una carpeta formando un package en Python.

Un **módulo** será un fichero de extensión .py que contiene la definición de un grupo de funciones y otros valores. Representan una biblioteca de funciones de un determinado tema. Por ejemplo, supongamos que comenzamos a formar una biblioteca simple de funciones de medidas (área, perímetro) geométrica. Guardamos en el fichero Geomet.py un grupo de funciones para calcular área, perímetro y volumen de una serie de figuras geométricas. Incluimos valor de la constante pi:

15 Puede consultarse google o [https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)

```
pi = 3.14159

def AreaCirc(radio):
    return pi*radio**2

def PerimCirc(radio):
    return 2*pi*radio

def VolCilindro(radio, h):
    return h*pi*radio**2

def AreaRect(b, h):
    return b*h

def PerimRect(b, h):
    return 2*(b+h)

def VolOrtoed(b, h, a):
    return b*h*a
```

Igual que los módulos internos y *math*, el módulo *Geomet.py* lo podemos importar para luego hacer uso de sus funciones y constantes. Por ejemplo,

```
from Geomet import pi, AreaCirc, AreaRect
print(AreaCirc(0.5))
print(AreaRect(3, 4))
```

Produce:

```
>>>
0.7853975
12
```

Se pueden importar todas las funciones y valores del módulo, en lugar de cada una de ellas, con \*,

```
from Geomet import *
```

O se puede importar el módulo completo y usar los valores o funciones de éste:

```
import Geomet
x = Geomet.pi*4
print(x)
print(Geomet.AreaCirc(0.5))
```

## Tema 5.

### Tipos de datos estructurados: homogéneos y heterogéneos, dinámicos y estáticos

Para la mayoría de los ejemplos hemos usado datos simples, que tiene asociado un único valor: un entero, un real o un booleano. Se dice que son objetos escalares por ser indivisibles, es decir no tienen una estructura interna accesible. También hemos introducido datos compuestos como los textos o cadena de caracteres, representados con *strings*, así como las secuencias en forma de listas que utilizamos para recorrer elementos en las composiciones iterativas *for*. Estos tipos de datos no son escalares pues se pueden dividir en elementos y acceder a ellos, son datos estructurados.

Los **datos estructurados o compuestos** contienen elementos, que pueden ser datos simples u otros datos compuestos. Recordamos que tanto los datos simples como compuestos en Python son tratados como un objeto.

Los elementos pueden ser todos del mismo tipo, como los string que contiene caracteres, y en este caso se llaman **datos estructurados homogéneos**. Otros lenguajes (C/C++, Matlab, Pascal) disponen de estructuras homogéneas como el array (arreglo o tabla), muy útiles para operaciones

con vectores o matrices. El Python estándar no dispone de una estructura como *array* de C o Pascal aunque la librería de Python numérico (NumPy)<sup>16</sup> sí incluye estas opciones.

También los elementos pueden ser de distinto tipo, como en las listas que hemos introducido previamente. Este tipo se suele llamar datos estructurados heterogéneos. En Python, excepto los string, los demás tipos de **datos compuestos son heterogéneos**, logrando una alta flexibilidad para el tratamiento de datos de problemas de todo tipo. En lenguajes como el Pascal, C/C++ y Matlab, se usan registros (record en Pascal), y estructuras (struc en C/C++, structure en Matlab) para el tratamiento de datos estructurados heterogéneos.

En Python los datos compuestos o estructurados los podemos clasificar en dos grupos, de acuerdo a la característica de si sus elementos pueden o no ser cambiados, reducidos o ampliados: datos estructurados **mutables** e **inmutables**.

## 5.1. Datos estructurados inmutables (estáticos)

Los datos estructurados inmutables, llamados también estáticos o de valores/tamaño fijos (Peña, 2015), se caracterizan porque los elementos de su secuencia no pueden ser cambiados ni eliminados. Tampoco se pueden añadir elementos nuevos a la estructura de datos. Si se quiere modificar este tipo de datos se utiliza el recurso de crear un nuevo valor.

En Python tenemos como datos estructurados inmutables las cadenas de caracteres (**string**) y las **tuplas** (*tuple*), como secuencias de datos, y los **conjuntos congelados** (*frozenset*).

### 5.1.1. Cadena de caracteres (string)

Los string, introducidos ya en la sección 2.1, son una secuencia de caracteres de cualquier tipo (letras, números, caracteres especiales; cualquier carácter Unicode) que forman un objeto de Python.

#### Indexación o acceso y longitud de las secuencias

Se pueden acceder a los elementos (caracteres) de los string (o de cualquier secuencia) a través de un **índice** que se pone entre corchetes, [*índice*], y dice en qué posición se encuentra el elemento dentro del string. Por ejemplo,

```
>>> s = 'casa de madera'
>>> letra_1 = s[0]
>>> long = len(s)
>>> letra_ultima = s[long-1]      # alternativa: s[-1]
>>> print(letra_1, letra_ultima, long)
c a 14
```

<sup>16</sup> <http://www.numpy.org/> Disponible en Spyder de Winpython y Anaconda.



El valor *'casa de madera'* es un objeto tipo string, que incluye una secuencia de 14 caracteres. Este valor se asigna a la variable *s*, que se referencia al mismo objeto. Accedemos al primer elemento con el índice 0 (*letra\_1 = s[0]*). Tal como lo indicamos en la introducción de las listas, recordemos que en Python el primer elemento de las secuencias está en la posición 0 cuando se indexa (accede).

Para calcular el número de elementos, o longitud, de la secuencia de los datos estructurados usamos la función interna *len()*. La cadena tiene 14 elementos y su último elemento está en la posición 13 (*longitud - 1*) o -1. Se ilustra mejor en la figura 11.

c	a	s	a		d	e		m	a	d	e	r	a
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Figura 11. Elementos del string y los índices para accederlos (positivos y negativos). Fuente: elaboración propia.

Un string vacío es: *s = ''* (dos comillas simples sin espacio), *len(s) = 0*.

### Recorte o rebanado (slicing) de secuencias y otras operaciones

Para extraer un subconjunto de elementos (o segmento) de un string o de cualquier secuencia se usa el operador de corte (slice) *[n:m]*, donde *n* es el primer elemento a extraer y *m-1* el último.

Se presentan varios ejemplos de acceso a los elementos y recorte de segmentos del string de la figura 11. En el comentario se indica el resultado:

```
>>> s = 'casa de madera'
>>> segm1 = s[0:3] # segm1 <- 'cas'
>>> segm1 = s[:3] # segm1 <- 'cas' ,
equivale al slice anterior
>>> segm2 = s[8:len(s)] # segm2 <- 'madera'
>>> segm2 = s[8:] # segm2 <- 'madera'
, equivale al slice anterior
>>> segm3 = s[0:14:2] # segm3 <- 'cs emdr', slice 0:14 en pasos de 2 en 2
>>> letra_u = s[-1] # letra_u <- 'a',
equivale a acceso último elemento
>>> letra_penu = s[-2] # letra_penu <- 'r', equivale acceso penúltimo elem
```

En el operador de corte, si se omite el primer índice *[m]* (anterior a los dos puntos) el recorte comienza desde el primer elemento. Si se omite el segundo índice *[n:]* se recorta hasta el final de la secuencia. Los índices negativos son útiles para acceder al último elemento *[-1]* o últimos, sin requerir el uso de la función *len()*.

Los otros operadores como concatenar (+) o repetir (\*) strings son aplicables a cualquier secuencia de datos:

```
>>> s1='casa'
>>> s2 = s1 + ' grande'
>>> s2
'casa grande'
>>> s3 = 3*s1 + '!'
>>> s3
'casacasacasa!'
```

El operador **in** se considera un operador booleano sobre dos strings y devuelve *True* si el string de la izquierda es un segmento (o substring) del de la derecha. Si no lo es, devuelve *False*. El operador **not in** devuelve el resultado lógico opuesto. Ejemplos:

```
>>> s = 'casa de madera'
>>> 'asa' in s
True
>>> 'casade' in s
False
>>> 'casade' not in s
True
```

### Los string son inmutables

Recordemos que este tipo de dato se considera inmutable porque no podemos cambiar los valores de sus elementos ni cambiar su tamaño. Si queremos hacer eso debemos crear otra variable (y otro valor de string antes, claro). Veamos, si queremos poner en mayúscula la primera letra del string *s* del ejemplo anterior, nos da error:

```
>>> s = 'casa de madera'
>>> s[0] = 'C'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Esta acción de poner en mayúscula la primera letra del string se puede hacer de manera automática, como se muestra en la siguiente sección, pero creando una nueva variable.

## Métodos de los strings

Python es un lenguaje orientado a objetos y los datos en Python están en los objetos. En la programación orientada a objetos, los objetos tienen asociados métodos para que manipulen sus datos. Los métodos son similares a las funciones, ya que reciben argumentos y devuelven valores. Los strings tienen métodos que le son propios. Por ejemplo, el método `upper` toma un string y devuelve otro string pero con las letras en mayúsculas.

El método `upper` en lugar de ser aplicado al string `s = 'casa de madera'`, como una función, `upper(s)`, se aplica de la forma `s.upper()`. Es decir, se aplica un método sobre sus valores. Veamos varios métodos de los strings (hay métodos con y sin argumentos):

```
>>> s = 'casa de madera'
>>> sM = s.upper()      # convierte las letras en mayúsculas
>>> sM
'CASA DE MADERA'
>>> sM.lower()          # convierte las letras en minúsculas
'casa de madera'
>>> s.capitalize()      # primera letra del string en mayúscula
'Casa de madera'
>>> s.title()           # primera letra de cada palabra del string en
mayúscula
'Casa De Madera'
>>> i = s.find('e')      # busca el índice (posición) del primer string
'e'
>>> i                    # si no encontrara el string devuelve -1
6
>>> s.count('a')         # cuenta las veces que aparece el elemento o string
4
>>> s.count('de')
2
>>> s.replace('a','e')   # reemplaza el primer string por el segundo
'cese de medere'
>>> s.split(' ')         # parte s usando el string ' ' produciendo lista
['casa', 'de', 'madera']
```

En el cuadro anterior se muestran un grupo de métodos típicos sobre los valores de los string. Resolvimos el problema de poner la primera letra en mayúscula con el método `capitalize()`. El método `Split()` divide el string en segmentos de acuerdo al delimitador que se use como argumento, que en este caso es el espacio en blanco. El argumento '' se usa por omisión, por lo que puede usarse `s.split()` para separar palabras en un texto. Los substring (palabras, en este caso) resultantes son devueltos en una lista con los substring como elementos.

### 5.1.2. Tuplas

Las tuplas, como los string, son una secuencia de elementos ordenados en un objeto de Python. A diferencia de los strings (elementos son caracteres) las tuplas pueden contener elementos de cualquier tipo, incluso elementos de diferente tipo. Los elementos se indexan igual que los string, a través de un número entero. La sintaxis de las tuplas es una secuencia de valores separados por comas. Aunque no son necesarios, se suelen encerrar entre paréntesis,

```
# Ejemplo de tuplas
>>> a = 1, 2, 3
>>> a
(1, 2, 3)
>>> b = (3, 4, 5, 'a')
>>> b
(3, 4, 5, 'a')
>>> type(a)
<class 'tuple'>
>>> type(b)
<class 'tuple'>
```

Los objetos asignados a las variables a y b son tipo tuplas. Lo importante es incluir las comas entre los elementos. Por ejemplo,

```
>>> t = 'k',
>>> t
('k',)
>>> type(t)
<class 'tuple'>
>>> t2 = 'k'
>>> t2
'k'
>>> type(t2)
<class 'str'>
```

El objeto 'k', es una tupla, sin embargo 'k' es un string. Se puede crear una tupla vacía usando paréntesis sin que incluya nada: (). Podemos también usar la función interna tuple() para convertir una secuencia iterable, como un string o una lista, a tupla, o crear una tupla vacía:

```
>>> tuple('Hola')
('H', 'o', 'l', 'a')
>>> tuple([1, 2])
(1, 2)
>>> tuple()
()
```

### Indexación, recorte y otras operaciones de tuplas

El acceso a los elementos de las tuplas, la extracción de elementos y las operaciones se realizan de forma análoga a los strings. Veamos varios ejemplos:

```
>>> b = (3, 4, 5, 'a')
>>> b[0]
3
>>> b[-1]
'a'
>>> b[0:3]
(3, 4, 5)
>>> t = ('las', 'tuplas', 'son', 'inmutables')
>>> t[0]
'las'
>>> t[1] = 'listas'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Se observa la característica estática o inmutable de las tuplas, al igual que los string. Podemos incluir tuplas dentro de las tuplas y concatenarlas y repetirlas, como los string,

```
>>> b = (3, 4, 5, 'a')
>>> c = (b, 2)
>>> b + c
(3, 4, 5, 'a', (3, 4, 5, 'a'), 2)
>>> 3*b
(3, 4, 5, 'a', 3, 4, 5, 'a', 3, 4, 5, 'a')
```

La composición iterativa for - in de Python puede usar cualquier secuencia iterable, incluyendo las tuplas:

```
>>> juegos = ('tennis', 'baseball', 'football', 'volleyball',  
'natación')  
>>> for deporte in juegos:  
...     print(deporte)  
tennis  
baseball  
football  
volleyball  
natación
```

### Asignaciones múltiples y funciones con devoluciones múltiples

Python permite asignaciones múltiples mediante **asignaciones con tuplas**. Estas acciones permiten que a una tupla de variables a la izquierda de una asignación le sea asignada una tupla de valores a la derecha de ésta. La condición a cumplir es que el número de variables de la tupla de variables sea igual al número de elementos de la tupla de valores. Incluso, el objeto a asignar de forma múltiple a la tupla de variables puede ser un string o una lista, siempre que el número de caracteres o elementos sea igual al número de variables de la tupla a la que se le asignan los valores. Veamos unos ejemplos

```
>>> a,b,c = (1,2,3)  
>>> a  
1  
>>> type(a)  
<class 'int'>  
>>> d,e,f = 'xyz'  
>>> d  
'x'  
>>> type(d)  
<class 'str'>
```

En la primera tupla de variables (*a,b,c*) las variables reciben valores enteros. Aunque este objeto es de tipo estructurado, tupla, sus elementos son variables de tipo entero. De igual forma, la tupla de variables (*d,e,f*) recibe cada una de ellas valores de tipo string y sus variables serán tipo string.

Esta característica de asignaciones con tuplas permite resolver de manera simple el típico **problema de intercambio de variables**, sin requerir de una variable auxiliar. Por ejemplo, si queremos intercambiar los valores de las variables  $x = 5$  e  $y = 7$ , en los lenguajes clásicos se haría:

```
>>> x = 5
>>> y = 7
>>> temp = x          # uso de variable auxiliar (temporal) temp
>>> x = y
>>> y = temp
>>> print(x, y)
```

Con asignaciones múltiples de tuplas, la solución es directa:

```
>>> x = 5
>>> y = 7
>>> x, y = y, x
>>> print(x, y)
7 5
```

En el caso de las funciones, éstas también pueden devolver múltiples resultados que pueden ser asignados a múltiples variables con el uso de tuplas. Siendo estrictos, las funciones solo devuelven un resultado. Pero si ese valor es una tupla, entonces ésta se puede asignar a una tupla de variables. Se requiere que el número de elementos coincida. Veamos la siguiente función como ejemplo:

```
def miFuncion(x):
    """
    Devuelve 2 valores: x incrementado y decrecido en 1
    """
    return x + 1, x - 1
a, b = miFuncion(10)
print(a, b)
print(miFuncion(20))
```

```
>>>
11 9
(21, 19)
```

La función devuelve una tupla de dos valores. En la primera instrucción del cuerpo principal del programa estos valores se asignan a la tupla con las variables a y b. Cada una de estas variables es de tipo entero y, para el argumento 10 de la función, reciben los valores 11 y 9, respectivamente. Estos valores los muestra el primer *print()*. El segundo *print()* muestra directamente la tupla que devuelve la función.

## Funciones con número arbitrario de parámetros, usando tuplas

En el tema anterior analizamos funciones con argumentos keywords. Existe la opción de definir una función con un número arbitrario (variable) de parámetros usando el operador `*` antes del nombre del parámetro. Veamos la función del siguiente ejemplo y sus diferentes llamados.

```
def media(*par):  
    suma = 0  
    for elem in par:  
        suma = suma + elem  
    return suma/len(par)  
print(media(3, 4))  
print(media(10.2, 14, 12, 9.5, 13.4, 8, 9.2))  
print(media(2))
```

```
>>>  
3.5  
10.9  
2.0
```

La función calcula el valor medio de la secuencia de números que se envía como argumento al parámetro de entrada, que espera recibir una tupla. Se puede mejorar la función para evitar dividir por 0, en caso de introducir una tupla vacía.

## Métodos de las tuplas

Al igual que en los string, existen métodos asociados a los objetos tipo tupla y listas. Pero solo los métodos: `s.index(x)` y `s.count(x)`. También se pueden usar las funciones internas *max* y *min*, cuando las tuplas (o listas) sean de valores numéricos. Si los elementos son strings, calcula el mayor o menor elemento, de acuerdo a la posición en la tabla ASCII del primer carácter. Veamos algunos ejemplos,



```
a = (2, 3, 4, 5, 79, -8, 5, -4)
>>> a.index(5)    # índice de la primera ocurrencia de 5 en a
3
>>> a.count(5)    # ocurrencias totales de 5 en a
2
>>> max(a)
79
>>> min(a)
-8
>>> b = ('az', 'b', 'x')
>>> max(b)
'x'
>>> min(b)
'az'
```

### 5.1.3. Conjuntos congelados (Frozenset)

En Python se dispone de otro grupo de datos estructurados heterogéneos que tratan de guardar cierta relación con la teoría de conjuntos. Estos datos son los conjuntos o Set y los Frozenset. Los primeros los presentamos en la siguiente sección de datos estructurados mutables o dinámicos.

Un conjunto congelado (Frozenset) es una colección de elementos no ordenados que sean únicos e inmutables. Es decir, puede contener números, string, tuplas, pero no listas. Que sean elementos únicos quiere decir que no estén repetidos. Los Set y Frozenset no son secuencias de datos.

Los conjuntos congelados son inmutables porque no se pueden cambiar, ni quitar o añadir elementos. Ejemplos de datos congelados:

```
>>> FS1 = frozenset({25, 4, 'a', 2, 25, 'casa', 'a'})
>>> FS1
frozenset({2, 'a', 4, 'casa', 25})
>>> type(FS1)
<class 'frozenset'>
>>> len(FS1)
5
```

Los elementos repetidos (25 y 'a') que incluimos en el conjunto congelados fueron desechados.

## 5.2. Datos estructurados mutables (dinámicos)

A diferencia de los string, tuplas y conjuntos congelados, los datos estructurados mutables, llamados también dinámicos (Peña, 2015), se caracterizan porque sus elementos pueden cambiar de valor y se puede añadir o eliminar elementos.

En Python tenemos como datos estructurados mutables las **listas**, los **conjuntos** (Set) y los **diccionarios**. Se puede considerar que las listas y conjuntos (Set) son los equivalentes mutables a las tuplas y conjuntos congelados (FrozenSet), respectivamente.

### 5.2.1. Listas

Las listas, así como las tuplas y strings, están formadas por una secuencia de datos. Pero a diferencia de las tuplas sus elementos pueden ser modificados, eliminarse o aumentarse. Los elementos de las listas pueden ser datos simples (numéricos o booleanos), strings, tuplas u otras listas. Los elementos se indexan igual que las tuplas y string, a través de un número entero. La sintaxis de las listas es una secuencia de valores separados por comas encerrados entre corchetes. Ejemplos:

```
>>> v1 = [2, 4, 6, 8, 10]
>>> type(v1)
<class 'list'>
>>> v2 = [7, 8.5, 'a', 'Hola', (2, 3), [11, 12]]
>>> v2
[7, 8, 'a', 'Hola', (2, 3), [11, 12]]
>>> juegos=['tennis','baseball','football','volleyball','natación']
>>> juegos
['tennis', 'baseball', 'football', 'volleyball', 'natación']
```

La lista v1 está formada por números enteros, mientras que v2 incluye enteros, reales, strings, tuplas y una lista como su último elemento. La variable juegos refiere a un objeto lista con 5 elementos de tipo string. Es similar a la tupla definida previamente pero sus elementos pueden ser modificables. Es una estructura dinámica.

Podemos generar una lista con una secuencia de números enteros con el tipo de datos *range()*, de la forma,

```
>>> v = list(range(1,11))
>>> v
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

En las versiones de Python 2.x, `range()` es una función que genera directamente una lista. Sin embargo en las versiones 3.x, al ser `range()` un tipo de dato *range*, tenemos que convertirlo a lista con la función `list()`. Esta función también sirve para convertir tipos de datos iterables, como strings o tuplas a tipo lista. Se puede crear también una lista vacía. Ejemplos:

```
>>> t = (1, 2, 3)
>>> list(t)
[1, 2, 3]
>>> s = 'Hola'
>>> list(s)
['H', 'o', 'l', 'a']
>>> e = list()                                # lista vacía
>>> e = []                                    # lista vacía
```

### Indexación, recorte y otras operaciones de listas

En las listas, el acceso a sus elementos, la extracción de elementos y las operaciones se realizan de la misma forma que en los strings y las tuplas. Los operadores de corte (*slice*) `[n:m]` se usan también en las listas. Veamos varios ejemplos:

```
>>> v2 = [7, 8, 'a', 'Hola', (2,3), [11, 12]]
>>> v2[0]
7
>>> v2[-1]
[11, 12]
>>> v2[-2]
(2, 3)
>>> v2[0:3]
[7, 8, 'a']
>>> t = ['las', 'listas', 'son', 'mutables']
>>> t[3] = 'dinámicas'
>>> t
['las', 'listas', 'son', 'dinámicas']
>>> len(t)
4
```

Se puede observar la mutabilidad de las listas. Las listas pueden concatenarse y repetirse con los operadores `+` y `*`, respectivamente, como los string y tuplas,

```
>>> v1 = [2, 4, 6, 8, 10]
>>> v3 = [3, 5, 7]
>>> v1 + v3
[2, 4, 6, 8, 10, 3, 5, 7]
>>> 3*v3
[3, 5, 7, 3, 5, 7, 3, 5, 7]
```

La composición iterativa `for - in` de Python ya ha sido utilizada con listas en el tema de composiciones iterativas. Con la lista `juegos` definida previamente, obtenemos:

```
>>> for deporte in juegos:
...     print(deporte)
tennis
baseball
football
volleyball
natación
```

Los operadores booleanos **in** y **not in**, similar que en los strings, evalúan si un elemento pertenece o no a una secuencia (tupla o lista). Ejemplos

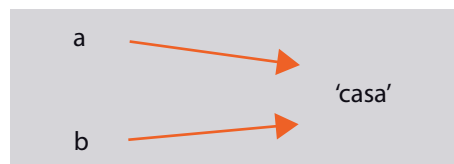
```
>>> v2 = [7, 8, 'a', 'Hola', (2,3), [11, 12]]
>>> 8 in v2
True
>>> 'Hola' in v2
True
>>> 'HOLA' not in v2
True
```

## Objetos, valores y referencias

Se dispone en Python del operador **is** que indica si dos variables están referidas al mismo objeto o no. Si ejecutamos las instrucciones

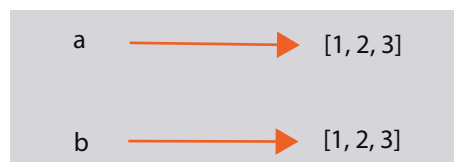
```
>>> a = 'casa'
>>> b = 'casa'
>>> id(a)
123917904
>>> id(b)
123917904
>>> a is b
True
```

Podemos ver que ambas variables *a* y *b* están referidas al mismo objeto, que tiene valor 'casa' y ocupa la posición de memoria 123917904 (esta posición es arbitraria). La instrucción *a is b* es cierta.



En los tipos de dato string, al ser inmutables, Python crea solo un objeto por economía de memoria y ambas variables están referidas al mismo objeto. Sin embargo con las listas, al ser mutables, aunque se formen dos listas con los mismos valores, Python crea dos objetos, que ocupan diferente posición de memoria:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
123921992
>>> id(b)
123923656
>>> a is b
False
```



Las listas asignadas a las variables *a* y *b*, aunque con el mismo valor, son objetos diferentes.

Pero hay que tener cuidado con las asignaciones de variable al mismo objeto mutable. En el siguiente ejemplo al copiar una variable no se crea otro objeto sino que el copiado se refiere al mismo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(b)          # a y b --> [1, 2, 3]
123921992
>>> a is b
True
```

Se puede decir que la variable `b` es un alias de `a` y que están **referenciadas**. Por lo tanto si modificamos o añadimos un valor al objeto `[1, 2, 3]`, a través de una de las variables, entonces modificamos la otra. Veamos

```
>>> b[0] = 15
>>> a
[15, 2, 3]
```

Este efecto puede dar resultados inesperados si no se maneja con cuidado. Sin embargo, esta propiedad sirve para pasar parámetros por referencia en funciones que se comporten como procedimiento. Si queremos copiar una variable de otra, se disponen del método `copy`, que se presentará a continuación.

### Métodos de las listas

Al ser las listas modificables, se dispone de un grupo amplio de métodos asociados a los objetos listas, que permiten añadir nuevos elementos, quitarle elementos, ordenarlos, etc. Hay que recordar que se usa el operador punto (.) para acceder a los métodos de los objetos. Veamos algunos ejemplos:

```
>>> v = [1, 2, 3, 4, 5]
>>> v.append(6)           # añade un objeto al final de la lista
>>> v
[1, 2, 3, 4, 5, 6]
>>> v2 = v.copy()         # copia la lista en otro objeto
>>> v.count(3)            # cuenta las veces que aparece el elemento (3)
1
>>> v.extend([7, 8, 9])   # extiende la lista con los elementos de otra
>>> v
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> v.index(5)            # índice de la primera ocurrencia del valor (5)
4
>>> v.insert(2, 4)        # insert(índice, valor), inserta 4 en posición 2
>>> v
[1, 2, 4, 3, 4, 5, 6, 7, 8, 9]
>>> v.pop(0)              # Remueve y devuelve elemento de posición
(0)
1                          # si se usa v.pop(-1) o v.pop() remueve el último
>>> v
[2, 4, 3, 4, 5, 6, 7, 8, 9]
>>> v.remove(4)           # remueve la 1ra ocurrencia del valor (4)
>>> v
[2, 3, 4, 5, 6, 7, 8, 9]
>>> v.reverse()          # invierte el orden de los elementos de la lista
>>> v
[9, 8, 7, 6, 5, 4, 3, 2]
>>> v1 = [4, -3, 5, 0, 1]
>>> v1.sort()             # ordena en orden ascendente
>>> v1
[-3, 0, 1, 4, 5]
>>> v1 = [4, -3, 5, 0, 1]
>>> v1.sort(reverse=True) # ordena en orden descendiente
>>> v1
[5, 4, 1, 0, -3]
>>> v.clear()             # remueve todos los elementos de la lsita
>>> v
[]
```

## Listas anidadas (tablas o matrices)

Una lista anidada es una lista donde sus elementos son a su vez listas. Este tipo de objeto es útil para representar tablas o matrices de datos. Por ejemplo, la matriz

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Se puede escribir en Python como

```
>>> m = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

En un editor de Python se puede escribir saltando a la siguiente línea después de la coma que separa cada elemento de la lista (que representa una fila de la matriz):

```
m = [[1, 2, 3, 4],  
      [5, 6, 7, 8],  
      [9, 10, 11, 12]]
```

Si accedemos al primer elemento de la lista, realmente accedemos a la primera fila. Para acceder al valor 7, que sería el elemento m<sub>23</sub>, en Python es la posición [1][2]:

```
>>> m[0]          # fila 1  
[1, 2, 3, 4]  
>>> m[1][2]       # elemento m_23 de la matriz (en Python m_12)  
7
```

Probemos dos matrices definidas con listas e intentemos sumarlas; por ejemplo,

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

```
>>> m1 = [[1, 1],  
...       [1, 1]]  
>>> m2 = [[1, 0],  
...       [0, 1]]  
>>> m1 + m2  
[[1, 1], [1, 1], [1, 0], [0, 1]]
```



¡No se ha realizado la suma de las matrices, sino la concatenación de la listas!

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

(¡No es lo que se esperaba al sumar dos matrices).

Se presenta a continuación un programa que lee dos matrices (N×N) utilizando listas y realiza su suma, a través de la composición iterativa *for - in*.

```
# Programa que suma 2 matrices NxN
N = int(input('Matriz cuadrada, entra e número de filas: '))
A = [[0]*N for i in range(N)] # se inicializan matrices N x N de 0's
B = [[0]*N for i in range(N)]
C = [[0]*N for i in range(N)]
for i in range(N):
    for j in range(N):
        A[i][j] = float(input('A[' +str(i)+'']['+str(j)+'']: '))
for i in range(N):
    for j in range(N):
        B[i][j] = float(input('B[' + str(i)+'']['+str(j)+'']: '))
for i in range(N):
    for j in range(N):
        C[i][j] = A[i][j] + B[i][j]
print('La suma de A\n', A , '\n y B\n', B, '\n es \n',C)
```

Este ejemplo muestra cómo recorrer una estructura de listas anidadas. Deben usarse dos índices para recorrer las listas dentro de las listas, que representan las columnas de cada fila. Cuando se asignan valores desde el teclado hay que prefijar el tamaño de la lista (inicializarla), para luego asignarle valores. Alternativamente se puede usar el método `append`. Resultado al introducir un número de filas igual a 2:

```
Matriz cuadrada, entra e número de filas: 2
A[0][0]: 1
A[0][1]: 1
A[1][0]: 1
A[1][1]: 1
B[0][0]: 1
B[0][1]: 0
B[1][0]: 0
B[1][1]: 1
La suma de A
[[1.0, 1.0], [1.0, 1.0]]
y B
[[1.0, 0.0], [0.0, 1.0]]
es
[[2.0, 1.0], [1.0, 2.0]]
```

### 5.2.2. Vectores y matrices con NumPy

Los arreglos de números o tablas se pueden trabajar mucho más eficientemente con los objetos *array* del paquete (package) de NumPy<sup>17</sup>. Estos módulos de cálculo numérico son una extensión del Python, pero muy útiles para el tratamiento de vectores y matrices, de forma similar al Matlab.

Los *arrays* de NumPy son datos estructurados homogéneos, similares a las listas (éstas son heterogéneas), pero enfocados al cálculo usando vectores y matrices. Los *arrays* suelen contener números enteros, reales y complejos. Los números complejos, que aún no los habíamos introducido, es otro tipo de dato numérico en Python. Su sintaxis es:

```
>>> c = 2+3j          # se pueden escribir con espacios: 2 + 3j
>>> type(c)
<class 'complex'>
```

Los *arrays* de NumPy que se formen con listas de enteros y reales se transforman todos sus elementos a reales. Si algún elemento de la lista es complejo, todos serán complejos. Para crear un *array* se importa la librería de cualquiera de las formas que hemos indicado previamente. Por ejemplo (realizado en Spyder de WinPython):

17 <http://www.numpy.org/> Disponible en Spyder de Winpython, <http://winpython.sourceforge.net/>, anaconda, <https://anaconda.org/anaconda/numpy> y en <http://scipy.org/>.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
```

Los **vectores** se utilizan en matemáticas y procesamiento de datos y señales para agrupar valores, como las tres coordenadas de un punto en el espacio  $(x, y, z)$ , o  $(x_1, x_2, x_3)$ . Las soluciones de un conjunto de  $n$  ecuaciones pueden agruparse en el vector  $(x_1, x_2, x_3, \dots, x_n)$ . Los datos numéricos de una señal muestreada en el tiempo se pueden agrupar en un vector.

Haremos un par de programas con las funciones de NumPy para sumar 2 vectores. En el primer programa se declaran los valores de los vectores. En el segundo, se pide el tamaño del vector al usuario, se leen los valores del teclado, se realiza el cálculo y se muestra el resultado. La inicialización de los vectores se hace para prefijar su tamaño. Es una buena opción crear un vector de ceros o de unos, que luego se le añaden los elementos.

```
import numpy as np
v1 = np.array([1, 2, 3.2])
v2 = np.array([5, 5, 5])
v3 = v1 + v2
print(v3)
```

```
[ 6.   7.   8.2]      # observar que los números son todos reales
```

```
import numpy as np
v1 = np.zeros(3)      # se crea un vector de 3 ceros (reales)
print("Introduce vector 1:")
for i in range(len(v1)):
    v1[i]=int(input("Elemento "+str(i)+" : "))
v2 = np.zeros(3)
print("Introduce vector 2:")
for i in range(len(v1)):
    v2[i]=int(input("Elemento "+str(i)+" : "))
v3 = v1 + v2
print(v3)
```

```
Introduce vector 1:
Elemento 0: 3
Elemento 1: 3
Elemento 2: 3
Introduce vector 2:
Elemento 0: 4
Elemento 1: 4
Elemento 2: 4
[ 7.  7.  7.]
```

Los objetos *array* se indexan y recortan de igual forma que las listas. También incluyen métodos asociados para hallar varias funciones matemáticas sobre sus elementos, como la suma, la media, desviación estándar, el máximo, etc. En el ejemplo anterior, `v3.sum()` devuelve el valor 21.0. Convertir un objeto *array* a lista se hace con el método `tolist`. Por ejemplo, `v3.tolist()` retorna la lista `[ 7. 7. 7.]`.

Las matrices que se presentaron con listas anidadas pueden también operarse con *arrays*. Veamos varios ejemplos:

```
>>> m1 = np.array([[1, 1],[1, 1]])
>>> m1
array([[1, 1],
       [1, 1]])
>>> m2 = np.array([[1, 0],[0, 1]])
>>> m2
array([[1, 0],
       [0, 1]])
>>> m3 = m1 + m2
>>> m3
array([[2, 1],
       [1, 2]])
```

La suma de matrices se hace directamente. Se propone que se modifique el ejemplo de suma de matrices con listas para realizarlo con *arrays*.

### Operaciones con vectores en lugar de iteraciones

En los *arrays* de Numpy, se pueden realizar directamente operaciones con funciones en lugar de recorrerlas con una composición iterativa. Se puede **vectorizar** la operación. Algunas funciones internas de Python, como `max()`, `min()` y `sum()`, también se pueden aplicar sobre una secuencia tipo tupla o lista. Supongamos que queremos aplicar una función matemática a un vector. En lugar de utilizar un bucle para recorrer el vector, podemos hacer:

```
import numpy as np
def f(x):
    return 4*2**x
a = np.array(range(1,11))
y = f(a)
print('El vector',a)
print('Aplicada la función f(x) = 4*2**x se obtiene')
print(y)
```

La secuencia de la dicotomía de Zenon representada anteriormente  $\sum 2^{-i}$ , puede calcularse con una función definida y calcular la **sumatoria** con la función interna `sum()` sobre un array:

```
import numpy as np
def f(x):
    return 1/2**x
N = int(input('Entra el número de términos: ' ))
a = np.array(range(1,N+1))
y = f(a)
print('El vector',y)
print('La suma es', sum(y))
```

```
Entra el número de términos: 7
El vector [0.5      0.25    0.125    0.0625    0.03125    0.015625
 0.0078125]
La suma es 0.9921875
```

El cálculo de **sumatorias** realizado con esquemas de recorrido con iteraciones definidas, se puede sustituir y realizar con más eficacia con la función interna `sum()` en una secuencia numérica (tupla, lista o array). También se puede hallar su mayor o menor valor:

```
>>> Lista = [1, 2, 3, 4]
>>> sum(Lista)          # devuelve valor 10
>>> max(Lista)          # devuelve valor 4
>>> min(Lista)          # devuelve valor 1
>>> Tupla = (1, 2, 3, 4)
>>> sum(Tupla)          # devuelve valor 10
>>> max(Tupla)          # devuelve valor 4
>>> min(Tupla)          # devuelve valor 1
```

### 5.2.3. Conjuntos (Set)

Los conjuntos (Set) son equivalentes a los Frozenset pero son estructuras mutables. Los conjuntos son una colección de elementos únicos e inmutables que no están ordenados. Siguen la idea de los conjuntos en matemáticas, donde los elementos no deben repetirse. Pueden contener números, string, tuplas, pero no listas.

Los conjuntos no pueden indexarse ni recortarse, pero se le pueden añadir o quitar elementos aunque solo a través de sus métodos asociados. Ejemplos conjuntos:

```
>>> S1 = {25, 4, 'a', 2, 25, 'casa', 'a'}
>>> S1
{'casa', 'a', 2, 4, 25} # los elementos repetidos fueron desechados
>>> type(S1)
<class 'set'>
>>> S1.add('b')
>>> S1
{'casa', 'a', 2, 4, 'b', 25}
>>> S1.remove('a')
>>> S1
{'casa', 2, 4, 'b', 25}
```

### 5.2.4. Diccionarios

Un diccionario es un objeto de Python similar a una lista, excepto que sus elementos pueden accederse a través de índices que no necesariamente tienen que ser enteros. Pueden ser de tipo texto (string), que es lo frecuente. Los índices se llaman claves (**keys**). Así, el **diccionario está formado por pares de clave-valor**. Su sintaxis se aprecia en el ejemplo:

```
>>> temperatura = {'Sevilla': 25.5, 'Londres': 15.4, 'Lisboa': 17.5} # o
>>> temperatura = dict(Sevilla=25.5, Londres=15.4, Lisboa=17.5)
>>> type(temperatura)
<class 'dict'>
>>> Dic = {} # Diccionario vacío
>>> Dic = dict() # Diccionario vacío
```

Son datos estructurados mutables, por lo que podemos añadirle un nuevo par clave-valor:

```
>>> temperatura['Valencia'] = 26
>>> temperatura
{'Sevilla': 25.5, 'Lisboa': 17.5, 'Valencia': 26, 'Londres': 15.4}
```

Se accede al valor los elementos a través de la clave y se pueden borrar elementos:

```
>>> temperatura['Londres']
15.4
>>> del temperatura['Sevilla']
>>> temperatura
{'Lisboa': 17.5, 'Valencia': 26, 'Londres': 15.4}
```

### Operaciones con diccionarios

Se puede recorrer un diccionario con una composición *for - in*, como otro dato estructurado, usando las claves como índices, veamos:

```
>>> for ciudad in temperatura:
...     print('Temperatura en',ciudad,'es',temperatura[ciudad])
...
Temperatura en Lisboa es 17.5
Temperatura en Valencia es 26
Temperatura en London es 15.4
```

El operador booleano *in* se puede emplear también en estas estructuras, pero se usa para averiguar si una clave forma parte del diccionario:

```
>>> 'Paris' in temperatura
True
>>> 'Lima' not in temperatura
True
```

Los objetos diccionarios tienen sus propios métodos asociados. Por ejemplo, se pueden extraer las claves y los valores de un diccionario con los métodos *keys* y *values*. La función *len()* devuelve el número de elementos (o pares clave-valor):

```
>>> temperatura.keys()
dict_keys(['Valencia', 'Lisboa', 'Londres'])
>>> temperatura.values()
dict_values([26, 17.5, 15.4])
>>> len(temperatura)
3
```

### 5.3. Funciones nulas (procedimientos) y paso de parámetros por referencia

Las funciones de Python que usen variables con datos mutables (como las listas) entre sus parámetros tiene la característica equivalente al **paso de parámetros por referencia** usado en otros lenguajes de programación, como el Pascal.

El **paso de parámetros** significa que el valor del argumento se copia al parámetro de la función. Si el parámetro es modificado dentro de la función, su valor no se referencia a la variable del argumento si esta variable es inmutable (enteros, reales, string, tuplas). En este caso, equivale a **paso de parámetros por valor**.

Pero, si las variables son mutables, como las listas, y la función modifica el valor de esta variable, entonces el nuevo valor sí se referencia al objeto original de la variable que se usó de argumento, tomando este nuevo valor. Equivale a **paso de parámetros por referencia**. Esta referencia se hace inmediatamente, sin esperar que acabe la función.

Se muestra un ejemplo de una función nula o void (procedimiento) que lee del teclado los valores de una lista de elementos de enteros, que es referenciada del programa principal.

```
""" Lectura de lista del teclado con procedure o función void
Programa que suma y halla la media de los elementos de una lista
"""
def leer_vector(v):
    for i in range(len(v)):          # v parámetro por referencia
        v[i]=int(input('Elemento '+str(i)+ ': '))
N = int(input('Entra el número de elementos: '))
a = N*[0]
leer_vector(a)
suma = 0
for i in range(N):
    suma = suma + a[i]
print('La lista',a)
print('La suma de sus elementos es',suma,'y el valor medio es', suma/N)
```



Si se introduce el valor  $N = 3$ , el programa inicializa la lista `a` con 3 elementos de valor 0 y esta lista es el argumento de la función. La llamada a la función hace que su parámetro se referencie al mismo objeto del argumento. La función lee del teclado los valores de los elementos de este objeto (llamado `a` en el programa y `v` en la función) y al acabar el bloque de instrucciones del cuerpo de la función, el objeto al que se referencia el argumento y el parámetro ha cambiado de valores. Esto equivale a **paso de parámetros por referencia**. Veamos:

```
Entra el número de elementos: 3
Elemento 0: 3
Elemento 1: 5
Elemento 2: 7
La lista [3, 5, 7]
La suma de sus elementos es 15 y el valor medio es 5.0
```

En Python el paso de parámetros viene ser “**por objetos**”. Si el objeto es inmutable, entonces el paso es por valor. Si el objeto es mutable el paso es por referencia.

El ejemplo anterior de procedimiento para leer los valores de una lista y calcular la suma y valor medio de los elementos, lo adaptamos a leer un vector (*array*) con NumPy. La suma de elementos y el cálculo valor medio están incluidos en los métodos asociados a los objetos *array* de NumPy. El programa se simplifica, pues el bucle `for - in` se sustituye por una vectorización:

```
""" Paso de parámetros por objeto
    Lectura de lista del teclado con procedure o función void
    Programa que suma y halla la media de los elementos de un vector
"""
import numpy as np
def leer_vector(v):
    for i in range(len(v)):          # v parámetro por referencia
        v[i]=int(input('Elemento '+str(i)+ ': '))
N = int(input('Entra el número de elementos: '))
a = np.zeros(N)                    # función zeros(N) genera array de N 0's
leer_vector(a)
print('La lista',a)
print('Suma de elementos es',a.sum(), 'y valor medio es',a.mean())
```

Hay que señalar que el paso de parámetros por referencia (paso de parámetros por objetos mutables en Python) puede realizarse también en funciones productivas, aunque esto no es lo usual en programación.

## 5.4. Ficheros (*files*)

Los datos y variables, o cualquier objeto, que se han usado en los ejemplos vistos hasta ahora se pierden al salir del programa (Python, o cualquier otro) o al apagar el computador. Esta información se almacena en la memoria RAM durante la ejecución del programa. Es decir, no tiene persistencia. La forma de guardar estos datos de manera permanente es mediante ficheros (archivos, *files*) en la memoria secundaria del ordenador. Hoy en día esta memoria puede ser el disco duro, memoria flash como los *pen drive* y tarjetas SD (Secure Digital) o las memorias (confusamente llamadas) ROM<sup>18</sup> en los teléfonos móviles actuales, etc.

Podemos decir que un **fichero** es un bloque de datos o información estructurada, que puede ser utilizado por los programas del computador. Los ficheros tienen un nombre y una ruta (*path*) para poder ser accedidos.

Como el lector seguramente habrá experimentado, en un computador se guardan muchos tipos de ficheros: texto (.txt), audio (.mp3, .wav), imágenes (.jpg) y un largo etcétera. La información guardada en los archivos es binaria (secuencia de bytes). Esta información binaria puede representar caracteres de un string (fichero de texto) o datos codificados de audio, imágenes, procesadores de texto, de hojas de cálculo, etc. (ficheros binarios).

Los ficheros binarios tienen una secuencia de bytes con la información codificada. Para leerlos o escribir sobre ellos, se debe conocer el formato de cómo se ha guardado la información. Y recorrerlos byte a byte o en bloques de bytes.

Los ficheros de texto guardan la secuencia de bytes conforme a la codificación de caracteres (ASCII latin, UTF-8, los primeros 256 caracteres de Unicode, etc.). Se estructuran como una secuencia de líneas de texto formadas por una secuencia de caracteres. Se cambia de línea con un byte codificado como salto de línea o nueva-línea (*newline*).

Python permite acceder a ficheros de texto o binarios para su lectura o escritura. Se presenta a continuación la forma de operar ficheros en Python y su comparación con lenguajes como C y Matlab.

---

<sup>18</sup> El origen de las memorias flash persistentes son las memorias EEPROM (Electrically Erasable Programmable Read-Only Memory) o ROM programable y borrrable eléctricamente.

	Python (solo función: open)	C, Matlab (todas funciones)
Abrir ficheros texto	<code>f = open(nf, modo_texto)</code>	<code>f = fopen(nf, modo)</code>
Lectura	<code>text = f.read()</code>	<code>A = fscanf(f, format)</code>
	<code>linea = f.readline()</code>	<code>linea = fgets(f)</code>
(Métodos en Python)	<code>lista = f.readlines()</code>	
Escritura	<code>f.write(string)</code> <code>f.writelines(sequencia)</code>	<code>fprintf(f, format, datos)</code>
Abrir ficheros binarios	<code>f = open(nf, modo_binario)</code>	<code>f = fopen(nf, modo)</code>
lectura	<code>d = f.read()</code>	<code>d = fread()</code>
escritura	<code>f.write(b'datos')</code>	<code>fwrite(f, datos, format)</code>
movimiento	<code>posicion = f.tell()</code> <code>f.seek(offset, origen)</code>	<code>posicion = ftell(f)</code> <code>fseek(f, offset, origen)</code>
Cerrar ficheros	<code>f.close()</code>	<code>fclose(f)</code>

Tabla 18. Funciones y métodos para manejar ficheros en Python, comparado con C y Matlab.

Los modos de apertura de ficheros de la función *open()* en Python y *fopen()* de C y Matlab se muestran en la siguiente tabla:

Python	C, Matlab
función open: modo texto (t)	función open: modo general
'r' abre y lee (por omisión)	'r' abre y lee (por omisión)
'w' abre y escribe (borra si hay algo)	'w' abre y escribe (borra si hay algo)
'r+' abre (si existe), lee y escribe	'r+' abre (si existe), lee y escribe
'w+' abre, lee y escribir (borra si hay algo)	'w+' abre, lee y escribir (borra si hay algo)
'a' abre y escribe (añade al final)	'a' abre y escribe (añade al final)
'a+' abre, lee y escribe (añade al final)	'a+' abre, lee y escribe (añade al final)
función open: modo binario (b)	
'rb''wb''rb+'''wb+'''ab''ab+'	

Tabla 19. Modos de apertura de ficheros en Python, comparado con C y Matlab.

En Matlab o C tanto la apertura de ficheros como la lectura, escritura o movimiento sobre el fichero se hace con funciones. La función *fopen()* devuelve un número entero que se asigna a una variable que

se usa como identificador del fichero abierto. En Python la función *open()* devuelve un identificador del fichero que es un objeto tipo entrada/salida de texto o binario. A este identificador también se le llama **manejador del fichero**. En Python las operaciones de lectura, escritura o movimiento sobre el fichero se hacen con métodos asociados al objeto.

El siguiente ejemplo muestra la creación de un fichero para guardar la información de un paciente al que se le ha registrado un electrocardiograma (ECG) ambulatorio de 24 horas, tipo Holter. Esta información puede ser generada directamente por el programa del equipo registrador. En estos casos se usa un fichero de texto con los datos del proceso de registro del ECG y otro(s) fichero(s) con los propios ECG. Los valores del ECG se suelen guardar en formato binario. La información de texto del paciente y el proceso de adquisición es la siguiente:

```
Sistema = HOLTER-ECG
Nombre = AGRCB
Fecha Prueba = 02/07/2016
Fecha Análisis = 03/07/2016
Número de canales = 3
Bits por muestra = 12
Muestras por segundo = 200
Rango dinámico = -5.0mV to +5.0mV
LSB = 10 microvolts
Sex = V
Fecha Nacimiento = 16/01/1950
Hora de conexión = 14:27
```

Se presentan dos programas que guardan la información en un fichero de texto. En el primero probamos guardar solo las dos primeras líneas. En el segundo guardamos el texto completo:

```
f = open('pac1.txt', 'w')
dos_lineas = 'Sistema = HOLTER-ECG\nNombre = AGRCB'
f.write(dos_lineas)
f.close()
```

Es importante cerrar el objeto de entrada y salida (f) para que la acción de crear el fichero se complete. Se puede observar que hay que incluir los caracteres especiales `\n` de escape para introducir el salto de línea. Este programa salva el fichero `pac1.txt` que contiene las dos primeras líneas de la información que queremos. Un texto de varias líneas conviene más crearlo con las triples comillas. Pondremos toda la información del paciente en un fichero:

```
f = open('pac1.txt', 'w')
texto = """Sistema = HOLTER-ECG
Nombre = AGRCB
Fecha Prueba = 02/07/2016
Fecha Analisis = 03/07/2016
Número de canales = 3
Bits por muestra = 12
Muestras por segundo = 200
Rango dinámico = -5.0mv to +5.0mv
LSB = 10 microvolts
Sex = V
Fecha Nacimiento = 16/01/1950
Hora de conexión = 14:27
"""
f.write(texto)
f.close()
```

Este programa utiliza el texto de información como un string entre comillas triples. Esta sintaxis incluye automáticamente el carácter `\n` al finalizar la línea escrita.

Vamos a leer lo escrito en ambos ficheros. El primero incluye solo las dos primeras líneas. Se presentan varias opciones de lectura.

```
f2 = open('pac1.txt', 'r') #
se crea el manejador de fichero f2
>>> f2.readline() # el método readline() lee una línea
'Sistema = HOLTER-ECG\n' # y salta al comienzo de la siguiente
>>> print(f2.readline())
Nombre = AGRCB
>>> f2.readline() # No hay más líneas
'' # devuelve string vacío
>>> f2.close()
```

Si se quiere leer línea a línea, el método `readline()` es muy práctico. Cuando se llega al final del fichero y se intenta leer algo nuevo, se devuelve un string vacío. La instrucción `open('pac1.txt','rt')` equivale a `open('pac1.txt','r')` y a `open('pac1.txt')`. La apertura por omisión es de lectura y de ficheros de texto. Si queremos leer el fichero entero:

```
>>> f = open('pac1.txt')
>>> f.read()
'Sistema = HOLTER-ECG\nNombre = AGRCB'
>>> f.tell()
36
>>> f.close()
```

Después de leer el todo el fichero (que contiene 35 caracteres) el método *tell()* nos dice la posición en que se ha quedado sobre el fichero para seguir actuando sobre él. En este caso, estamos al final (o en la posición que sigue al último carácter).

Probemos el segundo fichero con el texto completo para leer algunas partes y cambiar otras:

```
>>> f = open('pac2.txt', 'r+')
>>> f.seek(10)          # se busca posición 10
10
>>> f.read(6)          # se leen 6 caracteres y la posición es la 16
'HOLTER'
>>> f.tell()           # dice la posición actual
16
>>> f.seek(31)
>>> f.write('Maria')    # escribe desde posición 31 del fichero
>>> f.seek(0)           # se posiciona en el inicio
0
>>> print(f.read(62))   # imprime lectura de primeros 62 caracteres
Sistema = HOLTER-ECG
Nombre = Maria
Fecha Prueba = 02/07/2016
```

El método *seek()* solo busca desde la posición 0 en el modo texto.

También se puede leer todo un fichero de texto iterando sobre el objeto *file*:

```
f = open('pac2.txt', 'r')
for linea in f:
    print(linea, end='')
f.close()
```

```
Sistema = HOLTER-ECG
Nombre = Maria
Fecha Prueba = 02/07/2016
Fecha Analisis = 03/07/2016
Número de canales = 3
Bits por muestra = 12
Muestras por segundo = 200
Rango dinámico = -5.0mv to +5.0mv
LSB = 10 microvolts
Sex = V
Fecha Nacimiento = 16/01/1950
Hora de conexión = 14:27
>>>
```

Los ficheros binarios guardan elementos tipo bytes de Python. Los **bytes** son objetos inmutables con elementos de un byte (8 bits) y valores entre [0, 256), equivalentes a la codificación binaria de caracteres ASCII. A continuación se muestra un ejemplo donde se crea un fichero binario, se escribe con 3 bytes, se lee uno de ellos y luego se leen los tres:

```
>>> f = open('datos.bin', 'wb')
>>> datos = bytes([12, 17, 254])
>>> datos
b'\x0c\x11\xfe'
>>> f.write(datos)
3
>>> f.close()
>>> f = open('datos.bin', 'rb+')
>>> f.seek(1)                                # se posiciona en el 2do byte
1
>>> d2 = f.read(1)                            # se lee el 2do byte, entero 17
>>> d2
b'\x11'                                     # byte del entero 17, en Hexadecimal: 11
>>> print(list(d2))
[17]
>>> f.seek(0)                                # se posiciona en el 1er byte
0
>>> list(f.read())                            # lista de la lectura de los 3
bytes
[12, 17, 254]                               # convertidos a enteros
>>> f.close()
```

La escritura de ficheros binarios en Python puede realizarse de una manera más eficiente y flexible aprovechando las características de módulos de struct, NumPy y ScyPy<sup>19</sup>. Estos módulos incorporan el uso de estructuras o arreglos de vectores o matrices para escribirlos directamente en ficheros binarios. NumPy, por ejemplo, permite importar datos de Matlab. En el ejemplo anterior, la lectura de los datos binarios se puede hacer con NumPy como:

```
>>> import numpy as np
>>> f = open('datos.bin', 'rb')
>>> np_datos = np.fromfile(f, dtype=np. uint8)
>>> np_datos
array([ 12,  17, 254], dtype=uint8)
>>> f.close()
```

El ejemplo siguiente muestra la facilidad de guardar y leer una matriz en un fichero binario con NumPy:

```
>>> import numpy as np
>>> m1 = np.ones([3,3])           # se crea una matriz 3x3 de unos
>>> m1
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.save('datos.npy', m1)
>>> matriz = np.load('datos.npy')
>>> matriz
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

19 <http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.io.html>



## Glosario

### Algoritmo

Una sucesión ordenada de acciones elementales que se han de realizar para conseguir la solución correcta de un problema en un tiempo finito, a partir de unos datos dados.

### Argumento

Valor pasado a una función cuando ésta es llamada. El valor se pasa al parámetro correspondiente de la función.

### Asignación

Acción en los lenguajes de programación que le da un valor a una variable. El valor de la derecha del símbolo de asignación se asigna a la variable a la izquierda del símbolo. En Python significa que la variable es referenciada al valor del objeto de la derecha del símbolo.

### Bit

Unidad primaria de la información en computación (combinación de binary digit).

### Byte

Palabras de 8 bits que forman las unidades de información usadas hoy en día en computadores. El término octeto para palabras de 8 bits también fue usado en España en el comienzo de la era de la informática, como traducción de byte.

### Compilador

Programa que lee un código fuente (secuencia de instrucciones) y lo traduce completamente al lenguaje de máquina creando un programa ejecutable.

### Computador o computadora

Máquina electrónica, hoy en día con tecnología digital, que mediante programas realiza cálculos, almacena y procesa información para resolver problemas de diversas naturalezas.

### Datos compuestos (o datos estructurados)

Comprenden datos formados de elementos con valores de un mismo tipo o de diferentes tipos que se representan unificados para ser guardados o procesados. Los elementos pueden ser datos simples o compuestos. Los datos compuestos pueden clasificarse en homogéneos y heterogéneos, de acuerdo al tipo de elementos de su estructura: de un solo tipo o de diversos tipos. En Python estos datos son objetos y se clasifican en datos compuestos mutables e inmutables, dependiendo de si los elementos de la estructura pueden ser cambiados, eliminados o agregados, o no.

### Datos simples

Tiene asociado un único valor: un entero, un real o un booleano (en Python). En otros lenguajes el carácter es un tipo de dato simple. Se les llama elementos u objetos escalares por ser indivisibles, es decir no tienen una estructura interna accesible.

### Expresiones

En programación se refieren a los mecanismos para hacer cálculos y se componen de combinaciones de valores e identificadores con operadores. Incluyen los clásicos cálculos aritméticos, pero también los booleanos o lógicos y los de relaciones o comparaciones.

### Fichero (archivo, file)

Bloque de datos o información estructurada, con un nombre, que puede ser utilizado por los programas del computador y se guardan en la memoria permanente del computador, como el disco duro, memoria flash, etc.

### Función

Subprograma que incluye una instrucción o un bloque de instrucciones que realizan un cálculo o una tarea, y que tiene un identificador como nombre. Pueden ser productivas, es decir, que devuelven un resultado y funciones nulas (void) que realizan acciones o modifican objetos, sin devolver un resultado.

### Identificador

Nombres usados para identificar las variables, funciones, módulos, etc. Pueden contener letras y números pero deben empezar siempre con una letra o el carácter guion bajo o subrayado “\_”. Se deben evitar las palabras reservadas propias del lenguaje de programación (keywords).

### Índice

Variable o valor con la cual se accede a los elementos de una secuencia de un dato estructurado como, por ejemplos, a un carácter de un string o a un elementos de una listas o una tupla.

## Informática

Área del conocimiento que estudia todo lo que hace referencia a la obtención y procesado de información por medios automatizados, como los computadores, a partir de unos datos determinados.

## Intérprete

Programa que lee una a una las instrucciones del código fuente y las traduce a las apropiadas del lenguaje de máquina. Va traduciendo y ejecutando las sentencias del programa a medida que las encuentra.

## Lenguaje de máquina

Es el lenguaje básico que entiende la unidad de procesamiento central (CPU) de un ordenador, que comprende el conjunto de instrucciones que se usa para programar el microprocesador ( $\mu P$ ). Es decir, se hace la programación de bajo nivel o lenguaje ensamblador, a nivel de la máquina.

## Lenguaje de programación

Es un lenguaje formal diseñado por humanos para realizar algoritmos e instrucciones que puedan ser procesados por una máquina, como un computador.

## Memoria

Lugar físico donde se almacenan datos e instrucciones en forma binaria para recuperarlos y utilizarlos posteriormente por los diversos sistemas de un computador.

## Memoria flash

Memoria no volátil (permanente) de electrónica de estado sólido, basada en tecnología de compuertas NOR o NAND.

## Memoria RAM

Memoria de acceso aleatorio para escribir y leer datos e instrucciones de programas (RAM, de *Random Access Memory*), que se borra al cerrar el programa o el computador.

## Memoria ROM

Memoria solo de lectura de las instrucciones y datos (ROM, de *read-only-memory*), que permanecen grabadas. En las especificaciones de los móviles actuales se suele indicar confusamente memoria ROM a la memoria flash que almacena los programas en general (sistema operativo y programas descargados por el usuario). El origen de las memorias flash persistentes son las memorias EEPROM (*Electrically Erasable Programmable Read-Only Memory*) o ROM programable y borrrable eléctricamente.

## Módulo

En Python son las bibliotecas de funciones de una especialidad dada. Se diseña en un fichero de extensión .py que contiene la definición de un grupo de funciones y otros valores. En otros lenguajes de programación se utilizan otros términos como paquete (*package*), unidades (*unit*) o caja de herramientas (*toolbox*). Los *package* en Python son una colección de módulos guardados en una carpeta.

## Octeto

Ver byte.

## Ordenador

Ver Computador.

## Parámetro

Es el nombre usado en la definición de la función, a modo de variable, que se relaciona con el valor que se le pasa como argumento desde el programa principal u otra función. Dentro de la función será usado como variable local.

## Procedimiento (*procedure*)

Subprograma que realiza acciones como leer datos, mostrar resultados, manejar ficheros, pero que no devuelven un resultado (como las funciones productivas). En lenguajes como Pascal o modula-3 se llama procedure. En Python son las funciones nulas (void).

## Python

Lenguaje de programación de alto nivel, considerado de muy alto nivel y de propósitos generales, que es ejecutado por un intérprete, y su licencia es de código abierto. Fue creado por Guido van Rossum a finales de la década de 1980 en Holanda y se publicó la primera versión a comienzos de la década de 1990.

## Procesos informáticos

Comprenden básicamente de la entrada de datos, el tratamiento o cálculo de datos mediante una secuencia de acciones preestablecidas por determinado programa, y la salida de datos.

## Programa

Secuencia de instrucciones para realizar un cálculo o conseguir la solución correcta de un problema en un tiempo finito, a partir de unos datos dados, en un computador. Se entiende como la codificación de un algoritmo en un lenguaje entendido por el computador.

## Programación estructurada

Paradigma de programación propuesto en la década de 1970 que contempla el uso de estructuras o composiciones secuenciales, de selección o alternativas e iterativas y con ellas se puede resolver cualquier función computable.

## Programación modular

Paradigma de programación para crear programas de mejor claridad y, principalmente, para reutilizar subprogramas. Esta programación facilita modificar y corregir los códigos por separado y también crear una librería de subprogramas utilizables por otros programas.

## RAM

Ver Memoria RAM.

## Recursión

Es un método de programar en la cual una función se llama a sí misma una o más veces en el cuerpo de la función.

## ROM

Ver Memoria ROM.

## Sistemas operativos

Son programas complejos o sistema de programas que sirven para relacionar la unidad de procesamiento central (CPU), es decir, el microprocesador ( $\mu$ P), memorias, etc., con otros programas y el usuario. Están diseñados sobre los  $\mu$ P's de los dispositivos, utilizando sus respectivos lenguajes de máquina o ensambladores.

## Shell de Python

Interfaz tipo consola interactiva del intérprete de Python. Sobre el Shell se puede ejecutar en modo línea de comando o modo inmediato una expresión como una calculadora.

### **Subprograma (o subrutina)**

Son un conjunto de instrucciones separadas del programa principal que realizan una tarea o un cálculo y pueden devolver un resultado. Son llamados del programa principal o de otro subprograma. En Python se refieren a las funciones productivas o nulas (void). En otros lenguajes de programación incluyen a las funciones que devuelven resultados y a los procedimientos (procedures) que realizan tareas.

### **Variable**

En la mayoría de lenguajes de programación, una variable se considera a un lugar de memoria donde se guarda un valor de tipo simple o compuesto, y tiene un nombre (identificador). En Python, variable se considera un nombre referido a un objeto, es decir a los valores contenidos en los objetos.

### **Variable global**

Se declaran dentro de una función con la palabra reservada global. Si una variable del programa principal tiene el mismo identificador que la definida como global dentro de una función, entonces será modificada por cualquier asignación que reciba la variable global.

### **Variable local**

Son las variables definidas y usadas dentro de las funciones, que desaparecen al terminar de ejecutarse la función. Los parámetros de las funciones son también una especie de variable local.

## Enlaces de interés

### Consejo General de Colegios Profesionales de Ingeniería en Informática

Organismo que agrupa todos los Colegios Profesionales de Ingeniería Informática de las distintas Comunidades Autónomas de España. Se encarga, en el ámbito de su competencia, del ejercicio de la profesión de ingeniería en informática.

<http://www.ccii.es/>

### Python 3 tutorial

Curso de Python 3 de diseñado por Denise Mitchinson y adaptado para python-course.eu por Bernd Klein. Es una introducción al lenguaje Python a nivel principiante e intermedio, con una gran cantidad de ejemplos y ejercicios. Es muy apropiado para el autoaprendizaje.

[http://www.python-course.eu/python3\\_course.php](http://www.python-course.eu/python3_course.php)

### Python para principiantes

Libro tutorial de Python de acceso libre, en español, del sitio web hispano de referencia sobre diseño y programación "Librosweb.es".

<http://librosweb.es/libro/python/>

### The Python tutorial

Tutorial sobre Python 3 de la página web de "Python Software Foundation" que administra el lenguaje Python y su licencia de código abierto.

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

### Zen de Python

Describe la "filosofía" del lenguaje de programación Python con diversos aforismos, escritos por Tim Peters. Describen claramente lo que es Python y lo que cada programador debe de cuidar al diseñar un programa informático. Se puede invocar desde la consola (Shell) de Python como: `import this`

<http://docs.python-guide.org/en/latest/writing/style/#zen-of-python>

<http://www.python.org.ar/wiki/PythonZen>





# Bibliografía

## Referencias bibliográficas

Böhm, C., & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM, 9(5), 366-371.

Copeland, B.J. (2008). The Modern History of Computing. In The Stanford Encyclopedia of Philosophy, E.N. Zalta, Ed. Recuperado de: <http://plato.stanford.edu/archives/fall2008/entries/computing-history/>

Diakopoulos, N. & Cass, S. (2016). Interactive: The Top Programming Languages 2016. IEEE Spectrum. Recuperado de: <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016>

Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. Communications of the ACM, 11(3), 147-148.

Dijkstra, E. W. (1969). Notes on Structured Programming. Eindhoven: Technische Hogeschool Eindhoven (THE).

Downey, A. (2015). Think Python. Needham, MA: Green Tea Press.

Downey, E.K., A. B., Elkner, J. & Meyers, C. (2002). How to Think Like a Computer Scientist: Learning with Python. Wellesley, MA: Green Tea Press.

Guo, P. (2014). Python is now the most popular introductory teaching language at top U.S. Universities. Communications of the ACM. Recuperado de <http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popularintroductory-teaching-language-at-top-usuniversities/fulltext>

IEEE Standards Committee. (2008). 754-2008 IEEE standard for floating-point arithmetic. IEEE Computer Society Std, 2008.

Kaelbling, L, White, J., Abelson, H., Lozano-Perez, T., Finney, S., Canelake, S., Grimson, E., Chuang, I., Horn, B. & Freeman, D. (2011). Introduction to Electrical Engineering and Computer Science I. 6.01 Course Notes Spring 2011. MIT OpenCourseWare. Recuperado de: [http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/Syllabus/MIT6\\_01SCS11\\_notes.pdf](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/Syllabus/MIT6_01SCS11_notes.pdf)

Klein, B. (2016). Python 3 Tutorial. Formatted output. Bodenseo. Recuperado de: [http://www.python-course.eu/python3\\_formatted\\_output.php](http://www.python-course.eu/python3_formatted_output.php)

Lomont, C. (2012). Introduction to x64 Assembly. Recuperado de: <https://software.intel.com/en-us/articles/introduction-to-x64-assembly>

Peña, R. (2015). Paseo por la programación estructurada y modular con Python. ReVisión, 8(1), 17-27. Recuperado de: <http://www.aenui.net/ojs/index.php?journal=revisión&page=article&op=viewArticle&path%5B%5D=184&path%5B%5D=293>

Real Academia Española (2014). Diccionario de la lengua española (23ª ed.). Recuperado de: <http://dle.rae.es/>

Wentworth, P., Elkner, J., Downey, A. B., & Meyers, C. (2012). How to think like a computer scientist: learning with Python 3. Recuperado de: <http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/thinkcspy3.pdf>

## Bibliografía recomendada

Downey, A. (2015). Think Python. How to Think Like a Computer Scientist. Needham, MA: Green Tea Press. Recuperado de: <http://greenteapress.com/wp/think-python-2e/>

Guttag, J. V. (2013). Introduction to computation and programming using Python. Cambridge, MA: The MIT Press.

Harrington, A. N. (2015). Hands-on Python Tutorial. Recuperado de: <http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/>

Langtangen, H. P. (2009). A primer on scientific programming with Python (Vol. 2). Berlin Heidelberg: Springer. Recuperado de: <http://link.springer.com/book/10.1007/978-3-642-54959-5>

Marzal-Varó, A., García-Luengo, I., & García-Sevilla, P. (2014). Introducción a la programación con Python 3. Castellón, España: Universitat Jaume I. Recuperados de: <http://repositori.uji.es/xmlui/handle/10234/102653>

Severance, C. (2015). Python para Informáticos: Explorando la información (Version 2.7.2). Licencia Creative Commons. Recuperado de: <http://do1.dr-chuck.net/py4inf/ES-es/book.pdf>

## Anexo A

**Tabla ASCII (codificación con 7 bits) y ASCII extendido (8 bits)**

Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]
1	1	[STAR OF HADING]	33	21	!
2	2	[START OF TEXT]	34	22	"
3	3	[END OF TEXT]	35	23	#
4	4	[END OF TRANSMISSION]	36	24	\$
5	5	[ENQUIRY]	37	25	%
6	6	[ACKNOWLEDGE]	38	26	&
7	7	[BELL]	39	27	'
8	8	[BACKSPACE]	40	28	(
9	9	[HORIZONTAL TAB]	41	29	(
10	A	[LINE FEED]	42	2A	*
11	B	[VERTICAL TAB]	43	2B	+
12	C	[FORM FEED]	44	2C	,
13	D	[CARRIAGE RETURN]	45	2D	-
14	E	[SHIFT OUT]	46	2E	.
15	F	[SHIFT IN]	47	2F	/
16	10	[DATA LINK ESCAPE]	48	30	0
17	11	[DEVICE CONTROL 1]	49	31	1
18	12	[DEVICE CONTROL 2]	50	32	2
19	13	[DEVICE CONTROL 3]	51	33	3
20	14	[DEVICE CONTROL 4]	52	34	4
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5
22	16	[SYNCHRONOUS IDLE]	54	36	6
23	17	[ENG OF TRANS BLOCK]	55	37	7
24	18	[CARCEL]	56	38	8
25	19	[END OF MEDIUM]	57	39	9
26	1A	[SUBSTITUTE]	58	3A	:
27	1B	[ESCAPE]	59	3B	;
28	1C	[FILE SEPARATOR]	60	3C	<
29	1D	[GROUP SEPARATOR]	61	3D	=
30	1E	[RECORD SEPARATOR]	62	3E	>
31	1F	[UNIT SEPARATOR]	63	3F	?

Decimal	Hex	Char	Decimal	Hex	Char
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	123	7B	{
92	5C	\	124	7C	
93	5D	]	125	7D	}
94	5E	^	126	7E	-
95	5F	_	127	7F	[DEL]

Tabla A.1. ASCII. Fuente: Wikimedia commons,  
<https://commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg>

**Tabla ASCII extendida (código de página 347)**

decimal	carácter	decimal	carácter	decimal	carácter	decimal	carácter
128	Ç	160	á	192	Ł	224	Ó
129	ü	161	í	193	⊥	225	β
130	é	162	ó	194	⌈	226	Ô
131	â	163	ú	195	⌋	227	Ò
132	ä	164	ñ	196	—	228	õ
133	à	165	Ñ	197	†	229	Õ
134	å	166	ª	198	ã	230	μ
135	ç	167	º	199	Ä	231	þ
136	ê	168	¿	200	ℓ	232	ƒ
137	ë	169	®	201	ℝ	233	Ú
138	è	170	¬	202	ℙ	234	Û
139	ï	171	½	203	⌞	235	Ù
140	î	172	¼	204	⌗	236	ý
141	ì	173	¡	205	=	237	Ý
142	Ä	174	«	206	⌘	238	˘
143	Å	175	»	207	α	239	˙
144	É	176	⋮	208	ð	240	–
145	æ	177	⋮	209	Ð	241	±
146	Æ	178	⋮	210	Ê	242	=
147	ô	179		211	Ë	243	¾
148	ö	180	‡	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	í	246	÷
151	ù	183	À	215	î	247	ˆ
152	ÿ	184	©	216	ï	248	°
153	Ö	185	¶	217	Ĵ	249	ˆ
154	Ü	186		218	ƒ	250	˙
155	ø	187	¶	219	■	251	¹
156	£	188	¶	220	■	252	³
157	Ø	189	ç	221		253	²
158	×	190	¥	222	ì	254	■
159	f	191	ƒ	223	■	255	

Tabla A.2. ASCII extendido. Fuente: Elaboración propia.

# Agradecimientos

## **Autor**

Dr. D. Pedro Gomis Román

## **Departamento de Metodología e Innovación**

*Coordinadora*

D.<sup>a</sup> Mercedes Romero Rodrigo

## *Diseñadores*

D.<sup>a</sup> Carmina Gabarda López

D.<sup>a</sup> Ana Gallego Martínez

D.<sup>a</sup> Cristina Ruiz Jiménez

D.<sup>a</sup> Sara Segovia Martínez

Reservados todos los derechos VIU - 2016 ©.



**viu**  
**.es**