

Fundamentos de Programación

Con aplicaciones en la Ingeniería

MSc. Ing. Alejandro Bolívar

2023

FUNDAMENTOS DE PROGRAMACIÓN

Con aplicaciones en la Ingeniería

Libro de estudio de la asignatura Computación I.

Fundamentos de Programación con Aplicaciones a la Ingeniería © 2023 by [Alejandro Bolívar](#) is licensed under [CC BY-NC 4.0](#)

Se permite la copia, uso y distribución de este ejemplar, bajo los términos de la licencia **Creative Commons Atribución 4.0**.

Tabla de contenido

1	Tabla de contenido	2
1	Unidad I. Fases en la resolución de problemas	7
1.1	Análisis del Problema	8
1.2	Diseño del algoritmo.....	11
1.3	Codificación de un programa.....	12
1.4	Compilación y ejecución de un programa	13
1.5	Verificación y depuración de un programa.....	13
1.6	Documentación y mantenimiento	14
2	Unidad II. Conceptos básicos de programación.....	17
2.1	La computadora	17
2.2	¿Cómo maneja la información un computador?	17
2.3	Unidades de medida de la información almacenada.....	20
2.4	Partes del computador.....	21
2.5	Algoritmo.....	28
2.6	Representación gráfica de los algoritmos	34
2.7	Programas y Programación	37
3	Unidad III. El lenguaje de Programación.....	42
3.1	Lenguaje de Programación Python	42
3.2	Características de Python	44
3.3	Ventajas	44
3.4	Desventajas	45
3.5	Instalación de Python.....	45
3.6	Depuración.....	46
3.7	Tipos de Datos	48
3.8	Elementos Básicos del Lenguaje	58
3.9	Operadores	64
3.10	Funciones predefinidas.....	80
3.11	Módulos Estándar.....	83
3.12	Entornos de desarrollo.....	88
3.13	Manejo de datos	96
3.14	Glosario	101
3.15	Ejercicios Resueltos de Estructura Secuencial.....	104
4	Unidad IV. Estructuras de Control	120
4.1	Control de flujo de datos.....	120
4.2	Sangría	120
4.3	Estructuras Selectivas	122
4.4	Ejercicios Resueltos con estructuras condicionales	126
4.5	Estructuras Repetitivas	130
4.6	Herramientas de programación	139
4.7	Secuencias y Series	144
4.8	Composición y descomposición de un número.	152
4.9	Glosario.....	157
4.10	Resumen	158
4.11	Recomendaciones para la escritura de código limpio.	159

5	Datos Estructurados	165
5.1	Manejo Básico de Cadenas	165
5.2	Listas	172
5.3	Manejo de Tuplas	174
5.4	Glosario	177
6	Archivos de Texto	180
6.1	Creación y Uso de Archivos de Texto	180
6.2	Campos	180
6.3	Registros	181
6.4	Creación de archivos de Texto	181
6.5	Uso de Archivos de Texto en un Programa	182
6.6	Definir el archivo que se usará y de qué modo se usará	182
6.7	Leer ó tomar información en archivos (texto, csv)	183
6.8	Escribir en un archivo	185
6.9	Cerrar el Archivo	186
6.10	Ejemplo de procesamiento de archivos	186
6.11	Lectura hasta el Fin de Archivo	187
6.12	Ejercicios de Archivo de Texto	189
6.13	Recorrido de un archivo según su estructura o contenido	192
7	Programación Modular	201
7.1	Ventajas de la programación modular	201
7.2	Funciones	202
7.3	Funciones definidas por el usuario	204
7.3.1	Definiendo funciones	204
7.4	Los parámetros	206
7.5	Sobre la finalidad de las funciones	210
7.6	Consideraciones prácticas	210
7.7	Documentación de funciones	211
7.8	Uso de DOCTEST para probar funciones	211
7.9	Recomendaciones de Escritura del Código	214
7.10	Ejercicios Resueltos	214
7.11	Ejercicios propuestos de funciones	219
7.12	Guía de estilo - PEP8	223
7.13	Resumen	224
8	Cálculo simbólico con Sympy	227
8.1	Importando SymPy	227
8.2	Declarando una variable simbólica	228
8.3	Números complejos	228
8.4	Manipulaciones algebraicas	229
8.5	Operaciones algebraicas	230
8.6	Ecuaciones algebraicas	231
8.7	Cálculo de límites	232
8.8	Cálculo de derivadas	232
8.9	Expansión de series	233
8.10	Integración	233
8.11	Ecuaciones diferenciales	237
8.12	Gráficos con Sympy	238

8.13	Exportando a LATEX	240
8.14	Ejemplos prácticos.	240
9	Referencias	242

Introducción

Este libro nace con la intención de proporcionar a los estudiantes del tercer semestre de la Facultad de Ingeniería de la Universidad de Carabobo, en la asignatura Computación I del Departamento de Computación una comprensión sólida y completa de los conceptos fundamentales de la programación enfocado en la resolución de problemas mediante el computador. Partiendo desde la lógica de programación, el estudiante aprenderá a desarrollar algoritmos eficientes y efectivos para resolver problemas complejos en un lenguaje de programación. Además, se abordan temas esenciales como el control de flujo, las estructuras de datos, el diseño modular, y la depuración de código.

Este texto representa una guía detallada y accesible para desarrollar habilidades en una de las disciplinas más relevantes e influyentes de nuestro tiempo. Esperamos que este libro proporcione la base necesaria para avanzar en su formación como programador, así como también permitirle desarrollar soluciones tecnológicas innovadoras e impactantes.

Unidad 1

Análisis de un problema

Los problemas se resolverán implementando tres etapas: La entrada definida por la pregunta: ¿qué se conoce?, el proceso dándole respuesta a ¿cómo lograrlo? Y la salida especificando ¿qué se quiere?

Objetivos de la unidad:

- Estudiar los conceptos básicos de computación.
- Conocer las unidades de medidas de almacenamiento de la información
- Estudiar las fases de resolución de problemas en computadora.

Unidad I. Fases en la resolución de problemas

La resolución de un problema en computadora inicia con comprender el planteamiento del problema y luego realizar la escritura de un programa y su ejecución en la misma.

Las características más sobresalientes de la resolución de problemas son [1]:

- a) Análisis. El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.
- b) Diseño. Una vez analizado el problema, se diseña una solución que conducirá a un algoritmo que resuelva el problema.
- c) Codificación (implementación). La solución se escribe en la sintaxis del lenguaje de alto nivel (por ejemplo, Pascal) y se obtiene un programa fuente que se compila a continuación.
- d) Ejecución, verificación y depuración. El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados “bugs”, en inglés) que puedan aparecer.
- e) Mantenimiento. El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.
- f) Documentación. Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (codificación) se implementa el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

Las fases de compilación y ejecución traducen y ejecutan el programa. En las fases de verificación y depuración el programador busca errores de las etapas anteriores y los elimina (ver Figura 1-1). Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración del programa. Por último, se debe realizar la documentación del programa.

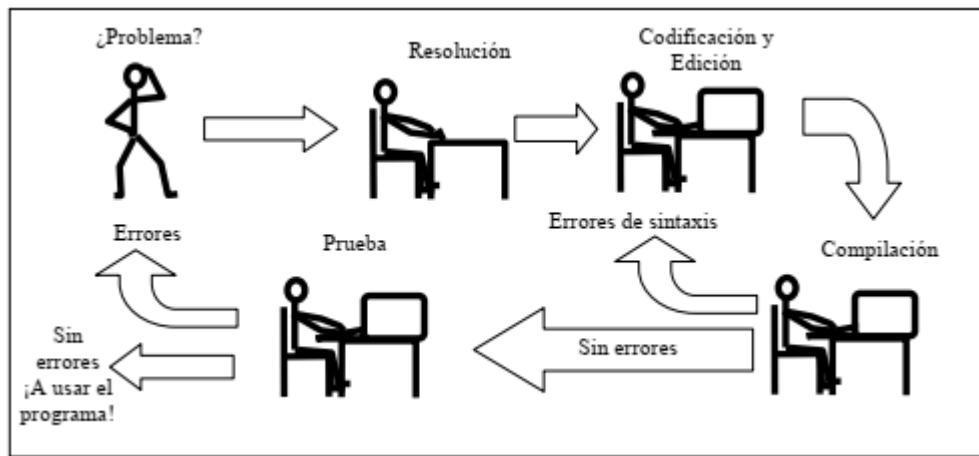


Figura 1-1. El proceso de resolución de problemas e implementación de un programa que lo resuelve [2].

Antes de conocer las tareas a realizar en cada fase, se considera el concepto y significado de la palabra algoritmo. La palabra algoritmo se deriva de la traducción al latín de la palabra Alkhô-warîzmi², nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX. Un algoritmo es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos.

Ejemplos de algoritmos son: instrucciones para montar en una bicicleta, hacer una receta de cocina, obtener el máximo común divisor de dos números, etc. Los algoritmos se pueden expresar por *fórmulas*, *diagramas de flujo* o *N-S* y *pseudocódigos*. Esta última representación es la más utilizada para su uso con lenguajes estructurados como Pascal.

1.1 Análisis del Problema

El proceso de Análisis consiste en la división de un todo en sus partes. En este caso, el todo es el problema planteado en forma de enunciado, pero ¿en cuales partes debe dividirse

como resultado del análisis? Inicialmente, el enunciado del problema se debe dividir en dos partes, las cuales se obtienen al responder las siguientes preguntas:

1. ¿Qué información útil se suministra dentro del enunciado?
2. ¿Qué resultados exige el enunciado?

Ejemplo: Dadas las longitudes de la base y la altura de un triángulo isósceles, determine el área que encierra el triángulo.

1. ¿Qué información útil se suministra dentro del enunciado?
 - a. La longitud de la Base
 - b. La longitud de la altura
 - c. El hecho que el triángulo es isósceles IRRELEVANTE
2. ¿Qué resultados exige el enunciado?
 - a. El área encerrada por el triángulo

La información de que el triángulo es isósceles es irrelevante para la solución del problema, ya que, si se conoce la longitud de la base y altura de un triángulo, no importa su forma, ya que la fórmula de cálculo requiere solamente estas longitudes.

Ejemplo: Dado las longitudes de los catetos de un triángulo rectángulo, determine la longitud de la hipotenusa.

1. ¿Qué información útil se suministra dentro del enunciado?
 - a. La longitud del primer cateto
 - b. La longitud del segundo cateto
 - c. El hecho de que el triángulo es rectángulo
2. ¿Qué resultados exige el enunciado?
 - a. La longitud de la hipotenusa

En este caso la información de que el triángulo es rectángulo es relevante, porque en el proceso de síntesis de la solución se usa el teorema de Pitágoras para encontrar el resultado. Sin esa información del enunciado, no existe una solución única al problema.

Antes del Diseño del Algoritmo primero es necesario realizar una pequeña síntesis del análisis a fin de establecer las fórmulas matemáticas, los procesos básicos de programación o la secuencia que deberá usarse dentro del diseño del algoritmo. Como es un paso intermedio entre el análisis y el diseño de la solución, esta pequeña síntesis se colocará dentro del análisis del problema añadiendo como tercera pregunta ¿Cómo logro obtener los resultados a partir de la información útil suministrada en el enunciado?

Para los ejemplos dados, el análisis completo sería:

Ejemplo: Dadas las longitudes de la base y la altura de un triángulo isósceles, determine el área que encierra el triángulo.

1. ¿Qué información útil se suministra dentro del enunciado?
 - a. La longitud de la Base
 - b. La longitud de la altura
 - c. El hecho que el triángulo es isósceles IRRELEVANTE
2. ¿Qué resultados exige el enunciado?
 - a. El área encerrada por el triángulo
3. ¿Cómo logro obtener los resultados a partir de la información útil suministrada en el enunciado?
 - a. Usando la fórmula $\text{area} = \text{base} * \text{altura} / 2$ por lo tanto la información de la forma del triángulo es irrelevante.

Ejemplo: Dadas las longitudes de los catetos de un triángulo rectángulo, determine la longitud de la hipotenusa.

1. ¿Qué información útil se suministra dentro del enunciado?

- a. La longitud del primer cateto
 - b. La longitud del segundo cateto
 - c. El hecho de que el triángulo es rectángulo
2. ¿Qué resultados exige el enunciado?
3. ¿Cómo logro obtener los resultados a partir de la información útil suministrada en el enunciado?
 - a. Con el teorema de Pitágoras, donde $\text{hip}^2 = \text{cat1}^2 + \text{cat2}^2$. El teorema de Pitágoras sólo se aplica si el triángulo es rectángulo, por lo tanto, la información de la forma es RELEVANTE.

1.2 Diseño del algoritmo

En la etapa de análisis del proceso de programación se determina *qué* hace el programa. En la etapa de diseño se determina *cómo* hace el programa la tarea solicitada. Los métodos más eficaces para el proceso de diseño se basan en el conocido *divide y vencerás*. Es decir, la resolución de un problema complejo se realiza dividiendo el problema en subproblemas y a continuación dividiendo estos subproblemas en otros de nivel más bajo, hasta que pueda ser *implementada* una solución en la computadora. Este método se conoce técnicamente como **diseño descendente** (*top-down*) o **modular**. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina *refinamiento sucesivo*. Cada subprograma es resuelto mediante un **módulo** (*subprograma*) que tiene un solo punto de entrada y un solo punto de salida [1].

Cualquier programa bien diseñado consta de un *programa principal* (el módulo de nivel más alto) que llama a subprogramas (módulos de nivel más bajo) que a su vez pueden llamar a otros subprogramas. Los programas estructurados de esta forma se dice que tienen un *diseño modular* y el método de romper el programa en módulos más pequeños se llama *programación modular*. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación

combinarlos entre sí. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

1. Programar un módulo.
2. Comprobar el módulo.
3. Si es necesario, depurar el módulo.
4. Combinar el módulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamientos sucesivos que permitan una posterior traducción a un lenguaje se denomina **diseño del algoritmo**.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente, usualmente se utiliza el diagrama de flujo o el pseudocódigo como representación alternativa del algoritmo previo a la codificación.

1.3 Codificación de un programa

La codificación de un programa se realiza sustituyendo, las palabras y expresiones del algoritmo o cualquier otra representación, por la sintaxis correspondiente en el lenguaje de programación. En el caso de programas sencillos se puede omitir el diseño del algoritmo para agilizar la codificación.

El programa también incluye una documentación que puede ser *interna* y *externa*. La documentación interna es la que se incluye dentro del código del programa fuente mediante comentarios significativos que ayudan a la comprensión del código, para ello se utiliza un carácter determinado que indique el inicio de los comentarios. El programa no los necesita y la computadora los ignora. Estas líneas de comentarios sólo sirven para hacer los programas más fáciles de comprender. El objetivo del programador debe ser escribir códigos sencillos y limpios. En cuanto a la documentación externa es la que se puede acceder externamente al código donde se especifica principalmente el objetivo del programa y los datos que maneja, adicionalmente del manual de usuario y de sistema.

1.4 Compilación y ejecución de un programa

Una vez que el algoritmo se ha convertido en un programa fuente, es preciso introducirlo en memoria mediante el teclado y almacenarlo posteriormente en un dispositivo de almacenamiento. Esta operación se realiza con un programa editor. Posteriormente el programa fuente se convierte en un *archivo de programa* que se guarda (graba) en un dispositivo de almacenamiento.

El **programa fuente** debe ser traducido a lenguaje máquina, este proceso se realiza con el compilador y el sistema operativo que se encarga prácticamente de la compilación.

Si tras la compilación se presentan errores (*errores de compilación*) en el programa fuente, es preciso volver a editar el programa, corregir los errores y compilar de nuevo. Este proceso se repite hasta que no se producen errores, obteniéndose el **programa objeto** que todavía no es ejecutable directamente. Suponiendo que no existen errores en el programa fuente, se debe instruir al sistema operativo para que realice la fase de **montaje o enlace** (*link*), carga, del programa objeto con las bibliotecas del programa del compilador. El proceso de montaje produce un **programa ejecutable**.

Una vez que el programa ejecutable se ha creado, ya se puede ejecutar (correr o rodar) desde el sistema operativo con sólo teclear su nombre (en el caso de DOS). Suponiendo que no existen errores durante la ejecución (llamados **errores en tiempo de ejecución**), se obtendrá la salida de resultados del programa.

Las instrucciones u órdenes para compilar y ejecutar un programa en C, C++, ... o cualquier otro lenguaje dependerá de su entorno de programación y del sistema operativo en que se ejecute Windows, Linux, Unix, etc.

1.5 Verificación y depuración de un programa

La *verificación* o *compilación* de un programa es el proceso de ejecución del programa con una amplia variedad de datos de entrada, llamados *datos de test o prueba*, que determinarán si el programa tiene o no errores (“*bugs*”). Para realizar la verificación se debe desarrollar una amplia gama de datos de test: valores normales de entrada, valores extremos

de entrada que comprueben los límites del programa y valores de entrada que comprueben aspectos especiales del programa.

La *depuración* es el proceso de encontrar los errores del programa y corregir o eliminar dichos errores. Cuando se ejecuta un programa, se pueden producir tres tipos de errores:

1. Errores de compilación. Se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación y suelen ser *errores de sintaxis*. Si existe un error de sintaxis, la computadora no puede comprender la instrucción, no se obtendrá el programa objeto y el compilador imprimirá una lista de todos los errores encontrados durante la compilación.

2. Errores de ejecución. Estos errores se producen por instrucciones que la computadora puede comprender, pero no ejecutar. Ejemplos típicos son: división por cero y raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error.

3. Errores lógicos. Se producen en la lógica del programa y la fuente del error suele ser el diseño del algoritmo. Estos errores son los más difíciles de detectar, ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y sólo puede advertirse el error por la obtención de resultados incorrectos. En este caso se debe volver a la fase de diseño del algoritmo, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

1.6 Documentación y mantenimiento

La documentación de un problema consta de las descripciones de los pasos a dar en el proceso de resolución de dicho problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Los programas pobremente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa puede ser *interna* y *externa*. La *documentación interna* es la contenida en líneas de comentarios. La *documentación externa* incluye análisis, diagramas de flujo y/o pseudocódigos, manuales de usuario con instrucciones para ejecutar el programa y para interpretar los resultados.

La documentación es vital cuando se desea corregir posibles errores futuros o bien cambiar el programa. Tales cambios se denominan *mantenimiento del programa*. Después de cada cambio la documentación debe ser actualizada para facilitar cambios posteriores. Es práctica frecuente numerar las sucesivas versiones de los programas **1.0, 1.1, 2.0, 2.1**, etc. (Si los cambios introducidos son importantes, se varía el primer dígito [**1.0, 2.0, ...**]; en caso de pequeños cambios sólo se varía el segundo dígito [**2.0, 2.1...**].)

Unidad 2

Conceptos básicos de programación.

En esta unidad se presenta una introducción a los conceptos básicos del área de la computación como el computador y sus partes, algoritmo y sus diferentes representaciones, y finalmente los paradigmas de programación.

Unidad II. Conceptos básicos de programación

2.1 La computadora

Una computadora es un dispositivo electrónico, utilizado para procesar información y obtener resultados, capaz de ejecutar cálculos y tomar decisiones a una velocidad de millones de veces más rápida en comparación a un ser humano. La computadora está conformada por dos partes: *hardware* y *software*. El *hardware* es la parte tangible mientras que el *software* es el conjunto de programas que indican a la computadora las tareas que debe realizar. Las computadoras procesan datos bajo el control de un conjunto de instrucciones denominadas programas de computadora. Estos programas controlan y dirigen a la computadora para que realice un conjunto de acciones (instrucciones) especificadas por personas especializadas, llamadas programadores de computadoras [1].

2.2 ¿Cómo maneja la información un computador?

En vista a la naturaleza de la computadora, ella no es capaz de manejar la información de la misma manera como lo hacen los seres humanos, es decir, en letras, palabras y números. Se hizo necesario, entonces, establecer una relación entre la manera que maneja la información una computadora y la manera como la maneja el ser humano. La computadora, como dispositivo electrónico, establece dos posibles valores de información representados simbólicamente a través de los dígitos 0 y 1; estos valores representan, dentro de la realidad, ausencia o presencia de voltaje o corriente dentro de la computadora.

Ausencia (0)



Presencia (1)



Desde el punto de vista matemático, la computadora maneja la información en un sistema de numeración BINARIO, es decir, dos dígitos; por otro lado, los seres humanos manejan la información numérica en un sistema de numeración DECIMAL, es decir, 10 dígitos. El dígito binario recibe el nombre de *BIT* en el contexto de la ciencia de la computación. La palabra ***Bit*** es el acrónimo de *BInary digiT*. (Dígito binario). La Real Academia Española (RAE) ha aceptado la palabra *bit* con el plural *bits*.

Si se toma en cuenta que la información que maneja el ser humano comprende un abecedario, con el cual se forman palabras, un sistema de numeración Decimal, con el cual se forman números y además, para la comprensión de la lectura, requiere de símbolos gramaticales y de ortografía, aunado a otros símbolos incluidos por las diversas ciencias exacta como la matemática, la física, etc., es fácil deducir que un dígito binario no puede representar a todo este cúmulo de información que día a día manejan los seres humanos. De la misma manera que el ser humano combina las letras para formar palabras o combina dígitos para formar números, los diseñadores iniciales de las computadoras decidieron formar números binarios como resultado de la combinación de dígitos binarios.

A diferencia de la forma como el ser humano crea sus palabras y números, la computadora se diseñó para trabajar con palabras de igual cantidad de dígitos binarios. Esta cantidad fue definida por Werner Buchholz en 1957 como números binarios de 8 dígitos y lo bautizó con el nombre de *BYTE*. La relación entre la computadora y el ser humano fue definida dentro de lo que se conoce como Tabla *ASCII* (*American Standard Code for Information Interchange*) que fue publicada como estándar por primera vez en 1967 y fue actualizado por última vez en 1986 (ver Tabla 2-1). En la actualidad define códigos para 33 caracteres no imprimibles, de los cuales la mayoría son caracteres de control obsoletos que tienen efecto sobre cómo se procesa el texto, más otros 95 caracteres imprimibles que les siguen en la numeración (empezando por el carácter espacio).

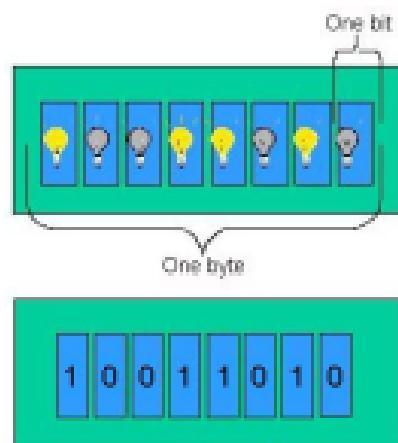


Tabla 2-1 ASCII de caracteres imprimibles.

Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car
32	20	espacio	48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
36	24	\$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
40	28	(56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
41	29)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
43	2B	+	59	3B	;	75	4B	K	91	5B	[107	6B	k	123	7B	{
44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	
45	2D	-	61	3D	=	77	4D	M	93	5D]	109	6D	m	125	7D	}
46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o			

Observe en la Tabla 2-1 la relación que existe entre los *Byte* y las letras del alfabeto. Desde el punto de vista del sistema numérico binario, la letra *A* mayúscula es menor que la letra *B* mayúscula y así sucesivamente; es decir que estas relaciones le permitirán a la computadora ordenar alfabéticamente una serie de nombres, ya que los conjuntos de *Byte* que las componen poseen la misma relación que el alfabeto.

A medida que la tecnología informática se difundió a lo largo del mundo, se desarrollaron diferentes estándares y las empresas desarrollaron muchas variaciones del código *ASCII* para facilitar la escritura de lenguas diferentes al inglés que usaran alfabetos latinos. Se pueden encontrar algunas de esas variaciones clasificadas como "ASCII Extendido". Estos nuevos símbolos corresponden a los bytes cuyo valor decimal está en el rango de 128 a 255. Dentro de estos nuevos símbolos está la eñe mayúscula y minúscula, las vocales acentuadas, símbolos gramaticales y símbolos de otras lenguas diferentes al castellano.

La Real Academia Española (RAE) define la palabra *Byte*, como la unidad de información compuesta generalmente de ocho bits

2.3 Unidades de medida de la información almacenada.

La memoria principal es uno de los componentes más importantes de una computadora y sirve para almacenamiento de información (datos y programas). Existen dos tipos de memoria y de almacenamiento: Almacenamiento principal (memoria principal o memoria central) y almacenamiento secundario o almacenamiento masivo (discos, cintas, etc.).

La memoria central de una computadora es una zona de almacenamiento organizada en centenares o millares de unidades de almacenamiento individual o celdas. La memoria central consta de un conjunto de celdas de memoria (estas celdas o posiciones de memoria se denominan también palabras, aunque no “guardan” analogía con las palabras del lenguaje). Cada palabra puede ser un grupo de 8 bits, 16 bits, 32 bits o incluso 64 bits, en las computadoras más modernas y potentes. Si la palabra es de 8 bits se conoce como byte. El término bit (dígito binario)¹⁰ se deriva de las palabras inglesas “*binary digit*” y es la unidad de información más pequeña que puede tratar una computadora. El término *byte* es muy utilizado en la jerga informática y, normalmente, las palabras de 16 bits se suelen conocer como palabras de 2 bytes, y las palabras de 32 bits como palabras de 4 bytes.

La memoria central de una computadora puede tener desde unos centenares de millares de bytes hasta millones de bytes. Como el byte es una unidad elemental de almacenamiento, se utilizan múltiplos para definir el tamaño de la memoria central: kilobyte (kB) igual a 1.024 bytes¹¹ (2¹⁰), Megabyte (MB) igual a 1.024×1.024 bytes (2²⁰ = 1.048.576), Gigabyte (GB) igual a 1.024 MB (2³⁰ = 1.073.741.824). Las abreviaturas MB, GB y TB se han vuelto muy populares como unidades de medida de la potencia de una computadora.

La aplicación de estos prefijos representa un mal uso de la terminología de medidas, ya que en otros campos las referencias a las unidades son potencias de 10. Por ejemplo, las medidas en distancias, kilómetro (km) se refiere a 1.000 metros, las medidas de frecuencias, Megahercio (MHz) se refieren a 1.000.000 de hercios. En la jerga informática popular para igualar terminología, se suele hablar de 1 kB como 1.000 bytes y 1 MB como 1.000.000 de bytes y un 1 GB como 1.000 millones de bytes, sobre todo para correspondencia y fáciles

cálculos mentales, aunque como se observa en la Tabla 2-2 estos valores son sólo aproximaciones prácticas [1].

Tabla 2-2. Unidades de medida de almacenamiento

<i>Byte</i>	<i>Byte</i>	(B)	<i>equivale</i>	a	8	<i>bits</i>
<i>Kilobyte</i>	<i>Kbyte</i>	(kB)	<i>equivale</i>	a	1.024	<i>bytes</i> (10^3)
<i>Megabyte</i>	<i>Mbyte</i>	(MB)	<i>equivale</i>	a	1.024	<i>Kbytes</i> (10^6)
<i>Gigabyte</i>	<i>Gbyte</i>	(GB)	<i>equivale</i>	a	1.024	<i>Mbytes</i> (10^9)
<i>Terabyte</i>	<i>Tbyte</i>	(TB)	<i>equivale</i>	a	1.024	<i>Gbytes</i> (10^{12})
<i>Petabyte</i>	<i>Pbyte</i>	(PB)	<i>equivale</i>	a	1.024	<i>Tbytes</i> (10^{15})
<i>Exabyte</i>	<i>Ebyte</i>	(EB)	<i>equivale</i>	a	1.024	<i>Pbytes</i> (10^{18})
<i>Zettabyte</i>	<i>Zbyte</i>	(ZB)	<i>equivale</i>	a	1.024	<i>Ebytes</i> (10^{21})
<i>Yotta</i>	<i>Ybyte</i>	(YB)	<i>equivale</i>	a	1.024	<i>Zbytes</i> (10^{24})

1 Tb = 1.024 Gb; 1 GB = 1.024 Mb = 1.048.576 kb = 1.073.741.824 b

2.4 Partes del computador

2.4.1 Hardware de una computadora

El **hardware** se refiere a todos los componentes físicos (que se pueden tocar), en el caso de una computadora personal serían el monitor, teclado, la placa base, el microprocesador, la tarjeta de memoria etc. *Hardware* es un neologismo proveniente del inglés definido por la Real Academia Española como el conjunto de elementos materiales que conforman una computadora; no tiene equivalente en el castellano.

De manera muy general, el *Hardware* de una computadora se puede clasificar según la función que elabora, en tres grandes grupos:

- 1) Microprocesador:
 - a. **Unidad Central de Proceso:** es la encargada de ejecutar las órdenes dadas a la computadora.
 - b. **Unidad Aritmética y Lógica:** es la encargada de realizar las operaciones aritméticas elementales, así como determinar la relación entre dos valores escalares.

- c. Partes misceláneas que sirven para el manejo de la información que entra y sale del microprocesador.

2) Memoria:

- a. **Random Access Memory (RAM):** es el lugar donde la computadora coloca la información y los procesos que la están manipulando a fin de obtener resultados. Su contenido se pierde al apagar la computadora, por eso se dice que esta memoria es volátil.
- b. **Read Only Memory (ROM):** contiene las órdenes iniciales que ejecuta la computadora al encenderse. Contiene, también, los procesos básicos para manejar y controlar todos las partes electrónicas y dispositivos que acompañan al microprocesador. Hoy en día se están sustituyendo por memorias flash para facilitar su actualización.

3) Periféricos:

Se denominan **periféricos** tanto a las unidades o dispositivos a través de los cuales la computadora se comunica con el mundo exterior, como a los sistemas que almacenan o archivan la información, sirviendo de memoria auxiliar de la memoria principal (RAM).

- a. **Unidades de Entrada:** Captan y envían los datos al dispositivo que los procesará, según las ordenes ejecutadas por la Unidad Central de proceso. A través de estas unidades el ser humano introduce información en la computadora. Ejemplos de unidades de entrada: Teclado, Mouse, Escáner, Cámara Web, Micrófono, Joystick, entre otros.
- b. **Unidades de Salida:** Son dispositivos que muestran o proyectan información hacia el exterior de la computadora. La mayoría son para informar, alertar, comunicar, proyectar o dar al ser humano cierta información. Ejemplos de unidades de salida: Monitor, Impresora, Altavoces, Auriculares, entre otros.
- c. **Unidades de comunicación:** Son los dispositivos que se encargan de comunicarse con otras máquinas o computadoras, ya sea para trabajar en conjunto, o para enviar y

recibir información. Ejemplos de unidades de comunicación: Fax-Modem, Tarjetas de Red, Wireless y Bluetooth, Controladores USB e Infrarrojos, entre otros.

- d. **Unidades de almacenamiento:** Son los dispositivos que almacenan información por bastante tiempo. La memoria RAM no puede ser considerada un periférico de almacenamiento, ya que su memoria es volátil y temporal. Ejemplos de unidades de almacenamiento: Disco Duro, Grabador o lector de CD, DVD, HD-DVD y Blu-ray, Memoria Flash, Memoria portátil, entre otros.

2.4.2 Software

Se denomina **software** (palabra de origen ánglico), programática, equipamiento lógico o soporte lógico a todos los componentes intangibles de una computadora, es decir, al conjunto de programas y procedimientos necesarios para hacer posible la realización de una tarea específica, en contraposición a los componentes físicos del sistema (hardware).

El software es quién al ejecutar una tarea da un ambiente de inteligencia similar a la del ser humano (y a veces hasta sorprendente) por parte de la computadora, a la que se le conoce como inteligencia aparente. Según su función, el software se clasifica en tres grandes grupos:

- a) **Software de sistema:** El software del sistema coordina las diferentes partes de un sistema de computadora y conecta e interactúa entre el software de aplicación y el hardware de la computadora. Otro tipo de software del sistema que gestiona, controla las actividades de la computadora y realiza tareas de proceso comunes, se denomina *utility* o utilidades (en algunas partes de Latinoamérica, utilerías). El software del sistema que gestiona y controla las actividades de la computadora se denomina sistema operativo. Otro software del sistema son los programas traductores o de traducción de lenguajes de computadora que convierten los lenguajes de programación, entendibles por los programadores, en lenguaje máquina que entienden las computadoras.

El software del sistema es el conjunto de programas indispensables para que la máquina funcione; se denominan también programas del sistema. Estos programas son,

básicamente, el sistema operativo, los editores de texto, los compiladores/intérpretes (lenguajes de programación) y los programas de utilidad [1].

- b) **Software de programación:** proporciona herramientas para ayudar al programador a escribir programas informáticos y a usar diferentes lenguajes de programación de forma práctica. Incluye entre otros: Editores de Texto, compiladores, intérpretes, enlazadores, depuradores.
- c) **Software de aplicación:** permite a los seres humanos llevar a cabo una o varias tareas más específicas, en cualquier campo de actividad susceptible de ser automatizado o asistido, con especial énfasis en los negocios.

Por otro lado, el software puede existir dentro de una computadora en tres formas diferentes:

- a) **Código fuente:** escrito por programadores. Contiene el conjunto de instrucciones destinadas a la computadora. El conjunto de instrucciones no está escrito en un lenguaje que la computadora pueda entender directamente, ya que está escrito en lenguaje del ser humano.
- b) **Código objeto:** resultado del uso de un compilador sobre el código fuente. Consiste en una traducción de este último hacia el lenguaje nativo de la computadora (byte). El código objeto no es directamente inteligible por el ser humano, pero tampoco es puede ser ejecutado por la computadora ya que le falta añadir fragmentos de código objeto para ser directamente funcional en la computadora donde quiere colocarse a trabajar.
- c) **Código ejecutable:** resultado de enlazar uno o varios fragmentos de código objeto. Constituye un archivo binario con un formato tal que el sistema operativo es capaz de cargarlo en la memoria de una computadora, y proceder a su ejecución.

2.4.3 Traductores de lenguaje: el proceso de traducción de un programa.

El proceso de traducción de un programa fuente escrito en un lenguaje de alto nivel a un lenguaje máquina comprensible por la computadora, se realiza mediante programas

llamados “traductores”. Los traductores de lenguaje son programas que traducen a su vez los programas fuente escritos en lenguajes de alto nivel a código máquina. Los traductores se dividen en compiladores e intérpretes [1].

2.4.3.1 Intérpretes

Un *intérprete* es un traductor que toma un programa fuente, lo traduce y, a continuación, lo ejecuta. Los programas intérpretes clásicos como BASIC, prácticamente ya no se utilizan, más que en circunstancias especiales. Sin embargo, está muy extendida la versión interpretada del lenguaje Smalltalk, un lenguaje orientado a objetos puro. El sistema de traducción consiste en: traducir la primera sentencia del programa a lenguaje máquina, se detiene la traducción, se ejecuta la sentencia; a continuación, se traduce la siguiente sentencia, se detiene la traducción, se ejecuta la sentencia y así sucesivamente hasta terminar el programa (Figura 2-1).



Figura 2-1. Interprete

2.4.3.2 Compiladores

Un *compilador* es un programa que traduce los programas fuente escritos en lenguaje de alto nivel a lenguaje máquina. La traducción del programa completo se realiza en una sola operación denominada **compilación** del programa; es decir, se traducen todas las instrucciones del programa en un solo bloque. El programa compilado y depurado (eliminados los errores del código fuente) se denomina *programa ejecutable* porque ya se puede ejecutar directamente y cuantas veces se desee; sólo deberá volver a compilarse de nuevo en el caso de que se modifique alguna instrucción del programa. De este modo el programa ejecutable no necesita del compilador para su ejecución. Los traductores de lenguajes típicos más utilizados son: **C, C++, Java, C#, Pascal, FORTRAN y COBOL** (Figura 2-2).

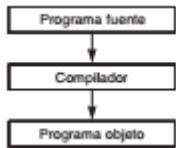


Figura 2-2. Compilación

2.4.4 Programa y dato

Los términos programa y dato se han utilizado sin que se definieran previamente, ya que son ambos son términos conocidos dentro de otros contextos. Antes de seguir avanzando en nuevos temas, es necesario dar una definición formal de estos términos dentro del contexto de estudio: La Computación.

Un **Programa** es una secuencia de pasos, órdenes o instrucciones que al ejecutarse logran cumplir una tarea específica. Un **Dato** es la información del ser humano representada simbólicamente a través de letras y dígitos en la computadora. De esta manera, la Programación de Computadoras es el proceso de crear programas que permitan automatizar los procesos de cálculo manual, así como también la automatización de procesos que incluyan la obtención de información para su posterior procesamiento.

La asignatura Computación I comprende la enseñanza de las técnicas básicas de programación para la resolución de problemas matemáticos o no, que generen en los estudiantes las habilidades de razonamiento para la automatización de los procesos de ingeniería que posteriormente serán dictados a lo largo de la carrera.

2.4.5 Resumen

Una computadora es una máquina para procesar información y obtener resultados en función de unos datos de entrada.

Hardware: parte física de una computadora (dispositivos electrónicos).

Software: parte lógica de una computadora (programas).

Las computadoras se componen de:

- Dispositivos de Entrada/Salida (E/S).
- Unidad Central de Proceso (Unidad de Control y Unidad Lógica y Aritmética).
- Memoria central.
- Dispositivos de almacenamiento masivo de información (memoria auxiliar o externa).

El *software del sistema* comprende, entre otros, el sistema operativo **Windows**, **Linux**, en computadoras personales y los lenguajes de programación. Los *lenguajes de programación* de alto nivel están diseñados para hacer más fácil la escritura de programas que los lenguajes de bajo nivel. Existen numerosos lenguajes de programación cada uno de los cuales tiene sus propias características y funcionalidades, y normalmente son más fáciles de transportar a máquinas diferentes que los escritos en lenguajes de bajo nivel. Los programas escritos en lenguaje de alto nivel deben ser traducidos por un compilador antes de que se puedan ejecutar en una máquina específica. En la mayoría de los lenguajes de programación se requiere un compilador para cada máquina en la que se desea ejecutar programas escritos en un lenguaje específico.

Los lenguajes de programación se clasifican en:

- *Alto nivel*: Pascal, FORTRAN, Visual Basic, C, Ada, Modula-2, C++, Java, Delphi, C#, etc.
- *Bajo nivel*: Ensamblador.
- *Máquina*: Código máquina.
- *Diseño de Web*: SMGL, HTML, XML, PHP.

Los programas traductores de lenguajes son:

- Compiladores.
- Intérpretes.

Un método general para la resolución de un problema con computadora tiene las siguientes fases:

1. Análisis del programa.

2. Diseño del algoritmo.
3. Codificación.
4. Compilación y ejecución.
5. Verificación.
6. Documentación y mantenimiento.

El sistema más idóneo para resolver un problema es descomponerlo en módulos más sencillos y luego, mediante diseños descendentes y refinamiento sucesivo, llegar a módulos fácilmente codificables. Estos módulos se deben codificar con las estructuras de control de programación estructurada.

1. *Secuenciales*: las instrucciones se ejecutan sucesivamente una después de otra.
2. *Repetitivas*: una serie de instrucciones se repiten una y otra vez hasta que se cumple una cierta condición.
3. *Selectivas*: permite elegir entre dos alternativas (dos conjuntos de instrucciones) dependiendo de una condición determinada)

2.5 Algoritmo.

2.5.1 Concepto y propiedades

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático.

Un **algoritmo** es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, algoritmo proviene, de Mohammed Al-Khôwârizmi, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra algorismus derivó posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo IV a. C.), quien inventó un método para encontrar el máximo común divisor de dos números, se considera junto con Al-Khôwârizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth, inventor de Pascal, Modula-2 y Oberon, tituló uno de sus más famosos libros, *Algoritmos + Estructuras de datos = Programas*, significándonos que solo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas para el inicio del aprendizaje de la programación.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo*, que describe la secuencia ordenada de pasos, sin ambigüedades, que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo*.)
2. Expresar el algoritmo como un programa en un lenguaje de programación adecuado. (*Fase de codificación*.)
3. *Ejecución y validación* del programa por la computadora.



Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan solo un medio para expresar un algoritmo y una computadora es solo un

procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

2.5.2 Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser **preciso** e indicar el orden de realización de cada paso.
- Un algoritmo debe estar **definido**. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser **finito**. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.
- Asimismo, **un algoritmo tiene que ser legible**. Esto significa que el texto que describe debe ser claro y conciso, de una manera tal que permita su comprensión inmediata, es decir sin procedimientos rebuscados o poco claros.
- Por último, **un algoritmo debe estar definido en tres partes fundamentales**, las cuales son: Entrada, Proceso y Salida.

2.5.3 Componentes de un algoritmo

Conceptualmente, un algoritmo tiene tres componentes:

1. la **entrada**: son los datos sobre los que el algoritmo opera;
2. el **proceso**: son los pasos que hay que seguir, utilizando la entrada;
3. la **salida**: es el resultado que entrega el algoritmo.



Nota: En programación, el proceso es una secuencia de **sentencias**, que debe ser realizada en orden. El proceso también puede tener **ciclos** (grupos de sentencias que son ejecutadas varias veces) y **condicionales** (grupos de sentencias que sólo son ejecutadas bajo ciertas condiciones).

2.5.4 Como hacer un algoritmo.

En el caso que necesitemos realizar un algoritmo para poder resolver problemas o mejorar algún proceso en nuestra actividad, lo podemos llevar a cabo de manera bastante sencilla, ya que lo único que tenemos que hacer es poner claro que necesitamos y como lo podemos resolver.

Para ello se debe realizar la serie de pasos necesarios y sus derivaciones para poder llegar al resultado esperado del problema que tenemos entre manos. Los pasos para definir y concretar un algoritmo son los siguientes:

- Primer paso: Análisis previo del problema o necesidad. Lo primero que tenemos que hacer, antes de comenzar con el desarrollo de cualquier algoritmo es llevar a cabo un análisis pormenorizado de la situación o problema.
- Segundo paso: Definir los requerimientos. En este paso se debe definir exactamente el problema que tenemos que solucionar y desglosarlo, incluyendo todas las derivaciones que puedan surgir.
- Tercer paso: La identificación de los módulos. En este paso, identificar claramente los módulos es tan importante como la identificación de los requerimientos. Esto es así debido a que identificar correctamente los módulos nos va servir para simplificar considerablemente la puesta en marcha de los pasos del algoritmo correcto para nuestro problema, y que hemos identificado en el paso anterior.
- Cuarto paso: La creación del algoritmo. En este punto debemos asegurarnos que el algoritmo cumpla con todos los requerimientos adecuados para llevar a cabo la función encomendada. Esto es similar tanto para los algoritmos no computacionales como para los algoritmos computacionales. En el caso de tratarse de un algoritmo computacional, además deberá cumplir con ciertas características para poder luego implementarse en cualquier lenguaje de programación.
- Quinto paso: La implementación del algoritmo. En el caso de los algoritmos computacionales, la implementación de los mismos se debe llevar a cabo traduciendo el mismo a un lenguaje de programación con el propósito de que cualquier computadora pueda interpretar sus instrucciones y enviar a su hardware la información necesaria para poder completar los pasos correspondientes y de esta manera obtener el resultado esperado.
- Sexto paso: Creación de las herramientas para llevar a cabo el algoritmo. En este último, y si pudimos cumplimentar correctamente con todos los pasos anteriores, ya estaremos en posición para poder crear las herramientas necesarias para poder ejecutar el algoritmo desarrollado. En el caso tratarse de un algoritmo computacional, podemos desarrollar a través de cualquier lenguaje de programación una aplicación para poder llevarlo a cabo, la cual contará con una serie de instrucciones que

ordenadas una detrás de la otra podrá representar el algoritmo que hemos diseñado y poder ofrecer una solución a los requerimientos identificados. En los casos en que se trate de un algoritmo no computacional, podemos desarrollar lo necesario teniendo en cuenta los pasos que debe seguir el algoritmo, como por ejemplo una línea de producción.

(Extraído de: <https://www.tecnologia-informatica.com/algoritmo-definicion/>)

Un ejemplo de Algoritmo de la vida diaria podría ser:

Preparación de una masa para pan:

1. cuente con manteca, harina, agua, levadura, y sal
2. mezcle los ingredientes
3. amase hasta formar un bolo
4. espere hasta leudar
5. caliente el horno
6. lleve el bolo al horno
7. retire del horno luego de X minutos

Observación: En este algoritmo no quedan claras muchas instrucciones, como por ejemplo cantidad de los ingredientes, temperatura del horno, tiempo de cocción, etc. Es decir, se requiere mayor precisión. También se puede decir que el lenguaje no es muy claro y no podrá ser interpretado por una persona que hable otro idioma, por ejemplo. Entonces es necesario definir ciertos acuerdos tanto en la forma de expresar un algoritmo sintaxis y en las palabras (o símbolos) que se utilizan para expresarlo.

Ejemplo: calcular la media de dos números

Algoritmo

1. Ingrese dos números, llamados a y b
2. Calcule la media de acuerdo a la fórmula $(a+b)/2$ y deje el resultado en m
3. Mostrar el valor de m

Observación: este algoritmo es más preciso y puede llevarse a un “programa” que obtenga la solución.

Para expresar un algoritmo podemos utilizar variadas técnicas como: **textual** mediante pseudocódigo, o **gráfica** mediante “diagramas de flujo”, u otras técnicas como

UML (lenguaje de modelamiento Unificado). A continuación, se le presentan varios ejemplos adicionales de algoritmo:

Ejemplo 2-1

Un cliente ejecuta un pedido a una fábrica. La fábrica examina en su banco de datos la ficha del cliente, si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario, lo rechazará. Redactar el algoritmo correspondiente.

Los pasos del algoritmo son:

1. Inicio.
2. Leer el pedido.
3. Examinar la ficha del cliente.
4. Si el cliente es solvente, aceptar pedido;
5. en caso contrario, rechazar pedido.
6. Fin.

Ejemplo 2-2

Se desea diseñar un algoritmo para saber si un número es primo o no.

Un número es primo si solo puede dividirse entre sí mismo y entre la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles entre números distintos a ellos mismos y a la unidad. Así, 9 es divisible entre 3, 8 lo es entre 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número entre 2, 3, 4..., etcétera.

1. Inicio.
2. Poner X igual a 2 ($X \leftarrow 2$, X variable que representa a los divisores del número que se busca N).
3. Dividir N entre X (N/X).
4. Si el resultado de N/X es entero, entonces N no es número primo y se bifurca al punto 7; en caso contrario continuar el proceso.
5. Suma 1 a X ($X \leftarrow X + 1$).
6. Si X es igual a N, entonces N es primo; en caso contrario bifurcar al punto 3.
7. Fin.

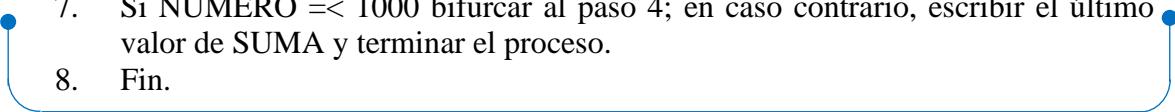
Por ejemplo, si N es 131, los pasos anteriores serían:

- 
1. Inicio.
 2. $X = 2$.
 3. $131/X$. Como el resultado no es entero, se continúa el proceso.
 4. $X \leftarrow 2 + 1$, luego $X = 3$.
 5. Como X no es 131, se continúa el proceso.
 6. $131/X$ resultado no es entero.
 7. $X \leftarrow 3 + 1$, $X = 4$.
 8. Como X no es 131 se continúa el proceso.
 9. $131/X \dots$, etc.
 10. Fin.

Ejemplo 2-3

Realizar la suma de todos los números pares entre 2 y 1000.

El problema consiste en sumar $2 + 4 + 6 + 8 \dots + 1\,000$. Utilizaremos las palabras SUMA y NÚMERO (variables, serán denominadas más tarde) para representar las sumas sucesivas $(2 + 4)$, $(2 + 4 + 6)$, $(2 + 4 + 6 + 8)$, etc. La solución se puede escribir con el siguiente algoritmo:

- 
1. Inicio.
 2. Establecer SUMA a 0.
 3. Establecer NÚMERO a 2.
 4. Sumar NÚMERO a SUMA. El resultado será el nuevo valor de la suma (SUMA).
 5. Incrementar NÚMERO en 2 unidades.
 6. Si NÚMERO ≤ 1000 bifurcar al paso 4; en caso contrario, escribir el último valor de SUMA y terminar el proceso.
 7. Fin.

2.6 Representación gráfica de los algoritmos

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitirá que un algoritmo pueda ser codificado de manera indistinta en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado gráfica o numéricamente, de modo que las sucesivas acciones no dependan de la sintaxis de ningún lenguaje de programación, sino que

la descripción pueda servir fácilmente para su transformación en un programa, es decir, su codificación.

Los métodos usuales para representar un algoritmo son:

1. diagrama de flujo,
2. lenguaje de especificación de algoritmos: pseudocódigo,
3. lenguaje español, inglés...
4. fórmulas.

Los métodos anteriores no suelen ser fáciles de transformar en programas. Una descripción en español narrativo no es satisfactoria, ya que es demasiado prolífica y generalmente ambigua. Una fórmula, sin embargo, es un buen sistema de representación. Por ejemplo, las fórmulas para la solución de una ecuación cuadrática (de segundo grado: $ax^2 + bx + c = 0$) son un medio sencillo de expresar el procedimiento algorítmico que se debe ejecutar para obtener las raíces de dicha ecuación.

1. Elevar al cuadrado b.
2. Tomar a; multiplicar por c; multiplicar por 4.
3. Restar el resultado obtenido de 2 del resultado de 1, etcétera.

Sin embargo, no es frecuente que un algoritmo pueda ser expresado por medio de una simple fórmula.

2.6.1 Pseudocódigo

El pseudocódigo es una forma de escribir algoritmos de manera informal y sin utilizar un lenguaje de programación específico. El objetivo es describir la secuencia de pasos necesarios para resolver un problema de manera clara y concisa, utilizando una serie de convenciones y símbolos que permiten representar estructuras comunes como bucles, condicionales, asignaciones de variables, entre otras operaciones.

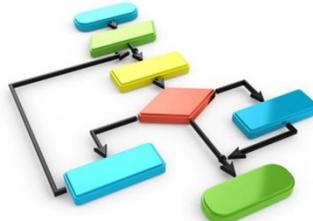
El pseudocódigo original utiliza para representar las acciones sucesivas palabras reservadas en inglés, similares a sus homónimas en los lenguajes de programación, como *start*, *end*, *stop*, *if-then-else*, *while-end*, *repeat-until*, etc. seguidos de expresiones matemáticas y/o lógicas que definen la acción a ejecutar. La escritura de pseudocódigo exige

normalmente la indentación (sangría en el margen izquierdo) de diferentes líneas. El objetivo es que otras personas puedan entender el algoritmo sin necesidad de conocer un lenguaje de programación específico. La ventaja del pseudocódigo es que, en su uso, en la planificación de un programa, el programador se puede concentrar en la lógica y en las estructuras de control y no preocuparse de las reglas de un lenguaje específico.

Por fortuna, aunque el pseudocódigo nació como un sustituto del lenguaje de programación y, por consiguiente, sus palabras reservadas se conservaron o fueron muy similares a las del idioma inglés, el uso del pseudocódigo se ha extendido en la comunidad hispana con términos en español como: **inicio, fin, parada, leer, escribir, si-entonces-si_no, mientras, fin_mientras, repetir, hasta_ que**, etc. Sin duda, el uso de la terminología del pseudocódigo en español ha facilitado y facilitará considerablemente el aprendizaje y uso diario de la programación.

2.6.2 Diagramas de Flujo

Un **diagrama de flujo (flowchart)** es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos estándar mostrados en la Tabla 2-3 y que tiene los pasos de algoritmo escritos en esas cajas unidas por flechas, denominadas líneas de flujo, que indican la secuencia en que se debe ejecutar.



Los símbolos estándar normalizados por **ANSI** (American National Standard Institute) son muy variados. En la Tabla 2-3 se representa una plantilla de dibujo típica donde se contemplan los símbolos principales utilizados en el diagrama.

Tabla 2-3. Símbolos de diagrama de flujo.

Símbolos principales	Función
----------------------	---------

	Terminador: regresa el inicio y fin de un programa
	Lineas de conexión y dirección de flujo: indica el sentido de ejecución de las operaciones.
	Entrada: Cualquier tipo de introducción de datos desde los periféricos.
	Salida: es utilizado para mostrar datos o resultados.
	Proceso general: permite realizar cálculos matemáticos y asignación de valores a las variables.
	Toma de decisiones: indica la evaluación de expresiones lógicas para decidir cual camino seguir.
	Bloque de procedimiento: Ciclo que repite un conjunto de instrucciones.
	Conector dentro de la página.
	Conexión fuera de la página.

2.7 Programas y Programación

La **Programación** es un proceso de resolución de problemas. Las técnicas más comunes incluyen análisis del problema, definición de los requerimientos del problema y diseño, que, en la práctica es la realización de un algoritmo, que es un método para resolver dicho problema.

Las computadoras procesan datos con el control de un conjunto de instrucciones denominadas programa de computadora. Los programas señalan a la computadora las tareas o acciones a realizar en función de un conjunto de instrucciones (acciones) especificadas por el constructor del programa llamado programador.

Un *programa de computadora* o *programa* es una secuencia de sentencias diseñadas para ejecutar una tarea. La **Programación** es un proceso de planeación y creación de un programa. Desde un punto de vista práctico, un programa se escribe primero con un algoritmo en pseudocódigo o en otra herramienta de programación y luego se traduce a un lenguaje de programación tal como Python, C, C++ o Java.

2.7.1 Paradigmas de Programación

Uno de los elementos básicos a la hora de realizar un programa es la modelización del problema que pretende resolver. Es preciso localizar las variables y los aspectos relevantes, así como comprender los pasos necesarios para obtener el resultado a partir de los datos iniciales, etc. Es decir, abstraer el problema, reduciendo sus detalles de forma que podamos trabajar con pocos elementos cada vez.

La algoritmia plantea la forma óptima de resolver problemas concretos, tales como la ordenación de una lista de elementos, la búsqueda de un elemento en un conjunto, la manipulación de conjuntos de datos, etc. Sin embargo, no proporcionan un marco general, un enfoque, que nos permita plantear y formular las soluciones a un problema de forma coherente.

Los paradigmas de programación llenan ese hueco, proporcionando guías tanto sobre cómo realizar la abstracción de los datos como sobre el control de la ejecución. Es decir, **los paradigmas de programación son herramientas conceptuales para analizar, representar y abordar los problemas**, presentando sistematizaciones alternativas o complementarias para pasar del espacio de los problemas al de las implementaciones de una solución.

Es muy recomendable y productivo comprender el enfoque de distintos paradigmas, puesto que presentan estrategias alternativas, incrementando nuestras herramientas

disponibles y haciéndonos reflexionar sobre muchas de las tareas que realizamos al crear un programa. Además, a menudo ocurre que unos problemas se formulan de forma muy clara si se los analiza según una perspectiva determinada, mientras que producen una gran complejidad vistos de otra manera.

2.7.1.1 Algunos paradigmas habituales

Algunas de las formas de pensar los problemas que ha llegado a sistematizarse como paradigmas de programación son:

1. **La programación modular:** Los programas se forman por partes separadas llamadas módulos. Estos funcionan de forma independiente entre sí, se relacionan a través de interfaces bien definidas.
2. **La programación procedural:** Un programa se compone de procedimientos (subrutinas, métodos o funciones) que son fragmentos de código que pueden llamarse desde cualquier punto de la ejecución de un programa, incluidos otros procedimientos o él mismo. (ej. ALGOL)
3. **La programación estructurada:** Se puede expresar cualquier programa utilizando únicamente tres estructuras de control: secuencial, condicional e iterativa. Los programas se componen de partes menores con un único punto de entrada, y se trata de aislarlos para evitar la complejidad que introducen los efectos colaterales. (ej. Pascal, C, Ada)
4. **La programación imperativa:** Un programa se puede definir en términos de estado, y de instrucciones secuenciales que modifican dicho estado. (ej. C, BASIC)
5. **La programación declarativa:** Es posible expresar un programa a través de condiciones, proposiciones o restricciones, a partir de los que se obtiene la solución mediante reglas internas de control. (ej. PROLOG)
6. **La programación funcional:** Se puede expresar un programa como una secuencia de aplicación de funciones. Elude el concepto de estado del cómputo y no precisa de las estructuras de control de la programación estructurada. (ej. LISP, Haskell)
7. **La programación orientada a objetos:** Los programas se definen en términos de “clases de objetos” que se comunican entre sí mediante el envío de mensajes. Es una evolución de los paradigmas de la programación procedural, estructurada y modular, y se implementa en lenguajes como Java, Smalltalk, Python o C++.

2.7.1.2 Programación multiparadigma

Los paradigmas de programación son idealizaciones, y, como tales, no siempre se presentan de forma totalmente ‘pura’, ni siempre resultan incompatibles entre sí. Cuando se

utilizan diversos paradigmas se produce lo que se conoce como **programación multiparadigma**.

El uso de distintos modelos, según se adapten mejor a las diversas partes de un problema o a nuestra forma de pensamiento, resulta también más natural y permite expresar de forma más clara y concisa nuestras ideas.

Algunos lenguajes, como *Haskell* o *Prolog* están diseñados para encajar perfectamente en la visión de un paradigma particular, mientras que otros, como *Python*, se adaptan especialmente bien a la programación multiparadigma y dan gran libertad a la hora de resolver un problema, al no imponen un mismo patrón para todos los casos.

Unidad 3. Conceptos básicos de programación

Esta unidad es fundamental para el inicio en la programación, se presentan los tipos de datos a utilizar, los operadores y como escribir expresiones aritméticas y lógicas.

Unidad III. El lenguaje de Programación

Objetivos de la unidad:

- Aprender a asignar el identificador adecuado a las variables y constantes.
- Identificar las palabras reservadas del lenguaje de programación.
- Asignar valores a variables y constantes.
- Aprender a escribir comentarios en el código.
- Identificar los diferentes tipos de errores durante la elaboración de un programa.
- Conocer los diferentes tipos de datos primitivos y derivados.
- Utilizar los operadores aritméticos, relacionales, de texto y lógicos.
- Realizar expresiones de tipo aritméticas y lógicas.
- Evaluar expresiones utilizando la jerarquía de operadores.
- Utilizar las funciones incorporadas en el lenguaje de programación.
- Importar módulos de terceros.
- Solicitar información del módulo.
- Utilizar funciones incorporadas en los módulos.
- Conocer los entornos de desarrollo integrado (IDE) y los entornos interactivos de desarrollo (IDLE (por sus siglas en inglés *Integrated Development Environment for Python*))
- Utilizar la función de impresión en pantalla para mostrar datos, incluyendo caracteres especiales y formatos.
- Utilizar la función de entrada de datos para la asignación de valores a las variables que utilizará el programa.
- Realizar ejercicios de estructura secuencial o lineal para la práctica de entrada de datos, elaboración de expresiones e impresión de resultados.

3.1 Lenguaje de Programación Python

Dentro de los **lenguajes informáticos**, Python, pertenece al grupo de los **lenguajes de programación** y es clasificado como un **lenguaje interpretado, de alto nivel, multiplataforma, de tipado dinámico, multiparadigma y orientado a objeto**.



Lenguaje interpretado o de script: Un lenguaje interpretado o de script es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (lenguajes compilados).

La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo, los lenguajes interpretados son más flexibles y más portables.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es semi interpretado. En Python, como en Java y muchos otros lenguajes, el código fuente se traduce a un pseudocódigo máquina intermedio llamado *bytecode* la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

Tipado dinámico: La característica de tipado dinámico se refiere a que *no es necesario declarar el tipo de dato* que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.

Fuertemente tipado: No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, *es necesario convertir de forma explícita dicha variable al nuevo tipo previamente*. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o *string*) no podremos tratarla como un número (sumar la cadena "9" y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.

Multiplataforma: El intérprete de Python está disponible en multitud de plataformas (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios.

Orientado a objetos: La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos.

Python también permite la programación imperativa, programación funcional y programación orientada a aspectos.

A diferencia de la mayoría de los lenguajes de programación, **Python provee reglas de estilos**, a fin de poder escribir código fuente más legible y de manera estandarizada. Estas reglas de estilo, son definidas a través de la ***Python Enhancement Proposal N° 8 (PEP 8)*** (<https://www.python.org/dev/peps/pep-0008/>).

3.2 Características de Python

Algunas características en la que se destaca Python son las siguientes:

- a. Fácil de aprender y de programar
- b. Fácil de leer (similar a pseudocódigo)
- c. Interpretado (rápido para programar)
- d. Datos de alto nivel (listas, diccionarios, sets, etc.)
- e. Libre y gratuito
- f. Multiplataforma (Windows, Linux y Mac)
- g. Pilas incluidas (biblioteca estándar incluida en el paquete)
- h. Cantidad de bibliotecas con funciones extras
- i. Comunidad

3.3 Ventajas

Las ventajas del lenguaje Python son las siguientes:

- 1. **Simplificado y rápido:** Este lenguaje simplifica mucho la programación “hace que te adaptes a un modo de lenguaje de programación, Python te propone un patrón”. Es un gran lenguaje para scripting, si usted requiere algo rápido (en el sentido de la ejecución del lenguaje), con unas cuantas líneas ya está resuelto.
- 2. **Elegante y flexible:** El lenguaje le da muchas herramientas, si usted quiere listas de varios tipos de datos, no hace falta que declares cada tipo de datos. Es un lenguaje tan flexible usted no se preocupa tanto por los detalles.
- 3. **Programación sana y productiva:** Programar en Python se convierte en un estilo muy sano de programar: es sencillo de aprender, direccionado a las reglas perfectas, le hace como dependiente de mejorar, cumplir las reglas, el uso de las líneas, de variables”. Además, es un lenguaje que fue hecho con productividad en mente, es decir, Python le hace ser más productivo, le permite entregar en los tiempos que me requieren.
- 4. **Ordenado y limpio:** El orden que mantiene Python, es de lo que más les gusta a sus usuarios, es muy legible, cualquier otro programador lo puede leer y trabajar sobre el programa escrito en Python. Los módulos están bien organizados, a diferencia de otros lenguajes.

5. **Portable:** Es un lenguaje muy portable (ya sea en Mac, Linux o Windows) en comparación con otros lenguajes. La filosofía de baterías incluidas, son las librerías que más usted necesita al día a día de programación, ya están dentro del interprete, no tiene la necesidad de instalarlas adicionalmente con en otros lenguajes.
6. **Comunidad:** Algo muy importante para el desarrollo de un lenguaje es la comunidad, la misma comunidad de Python cuida el lenguaje y casi todas las actualizaciones se hacen de manera democrática.

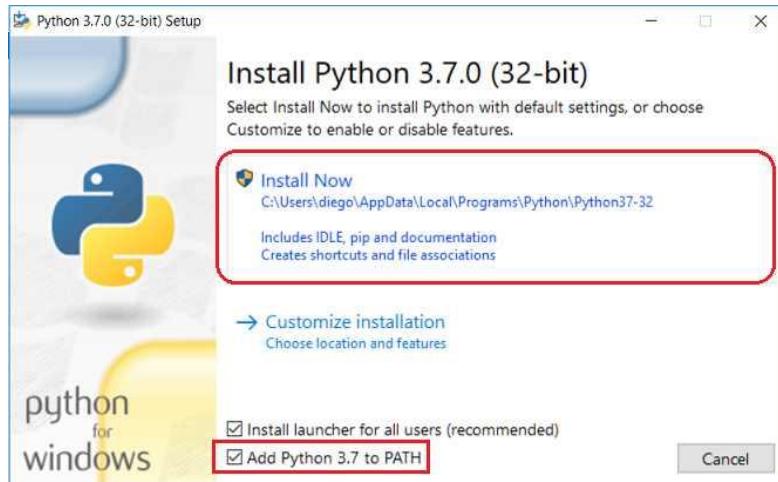
3.4 Desventajas

Las desventajas del lenguaje Python son las siguientes:

1. **Curva de aprendizaje:** La “curva de aprendizaje cuando ya estás en la parte web no es tan sencilla”.
2. **Hosting:** La mayoría de los servidores no tienen soporte a Python, y si lo soportan, la configuración es un poco difícil.
3. **Librerías incluidas:** Algunas librerías que trae por defecto no son del gusto de amplio de la comunidad, y optan a usar librerías de terceros.

3.5 Instalación de Python

Para la descarga del lenguaje Python se ingresa al sitio: python.org (descargar la versión más actual 3.X). Ejecutamos el programa que descargamos y procedemos a instalarlo (marcamos 'Add Python 3.X to PATH').



Luego de haberse instalado se puede ejecutar desde el menú de opciones de Windows.

3.6 Depuración

La programación es un proceso complejo y, por ser realizado por humanos, a menudo desemboca en errores. Por razones caprichosas, esos errores se llaman *bugs* y el proceso de buscarlos y corregirlos se llama depuración (en inglés “*debugging*”).

Hay tres tipos de errores que pueden ocurrir en un programa: de sintaxis, en tiempo de ejecución y semánticos. Es muy útil distinguirlos para encontrarlos más rápido.

3.6.1 Errores sintácticos

Python sólo puede ejecutar un programa si el programa es correcto sintácticamente. En caso contrario, es decir si el programa no es correcto sintácticamente, el proceso falla y devuelve un mensaje de error. El término sintaxis se refiere a la estructura de cualquier programa y a las reglas de esa estructura.

Para la mayoría de lectores, unos pocos errores sintácticos no son significativos, y por eso puede leerse el código sin anunciar errores de sintaxis. Python no es tan permisivo. Si hay, aunque sea un solo error sintáctico en el programa, Python mostrará un mensaje de error y abortará la ejecución del programa. Durante las primeras semanas de su carrera como programador pasará, seguramente, mucho tiempo buscando errores sintácticos. Sin embargo, tal como adquiera experiencia tendrá menos errores y los encontrará más rápido.

```
>>> 4/*8SyntaxError: invalid syntax
>>> pirnt('Hola mundo!')
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    pirnt('Hola mundo!')
NameError: name 'pirnt' is not defined
```

3.6.2 Errores en tiempo de ejecución

El segundo tipo de error es un error en tiempo de ejecución. Este error no aparece hasta que se ejecuta el programa. Estos errores también se llaman excepciones porque indican que algo excepcional (y malo) ha ocurrido.

Con los programas que vamos a escribir al principio, los errores en tiempo de ejecución ocurrirán con poca frecuencia, así que puede pasar bastante tiempo hasta que vea uno.

```
>>> alumnos =56
>>> grupos =0
>>> alum_grupo =alumnos/grupos
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    alum_grupo = alumnos/grupos
ZeroDivisionError: division by zero
```

3.6.3 Errores semánticos

El tercer tipo de error es el error semántico. Si hay un error de lógica en su programa, el programa se ejecutará sin ningún mensaje de error, pero el resultado no será el deseado. Será cualquier otra cosa. Concretamente, el programa hará lo que usted le dijo.

A veces ocurre que el programa escrito no es el programa que se tenía en mente. El sentido o significado del programa (su semántica) no es correcto. Es difícil detectar los errores de lógica, porque requiere trabajar al revés, observando el resultado del programa para averiguar lo que hace.

```
# dada la base y la altura de un triángulo rectángulo,
# determine el área.
>>> base = 4
>>> altura = 2
>>> area = base / altura
>>> print(area)
2.0
```

3.6.4 Depuración experimental

Una de las técnicas más importantes que usted aprenderá es la depuración. Aunque a veces es frustrante, la depuración es una de las partes más intelectualmente ricas, interesantes y estimulantes de la programación.

La depuración es una actividad parecida a la tarea de un investigador: se tienen que estudiar las claves para inducir los procesos y eventos que llevaron a los resultados que tiene a la vista.

La depuración también es una ciencia experimental. Una vez que se tiene la idea de cuál es el error, se modifica el programa y se intenta nuevamente. Si su hipótesis fue la correcta se pueden predecir los resultados de la modificación y estará más cerca de un programa correcto. Si su hipótesis fue errónea tendrá que idearse otra hipótesis.

Para algunas personas, la programación y la depuración son lo mismo: la programación es el proceso de depurar un programa gradualmente hasta que haga lo que usted quiera. La idea es que debería usted comenzar con un programa que haga algo y hacer pequeñas modificaciones, depurándolas sobre la marcha, de modo que siempre tenga un programa que funcione.

3.7 Tipos de Datos

Una característica de Python es la cantidad y versatilidad de sus tipos de datos. Conocerlos en profundidad sirve para tener un buen criterio a la hora de elegir como modelar nuestros datos. En líneas generales los tipos de datos (ver Figura 3-1) se dividen en *primarios o simple* y *compuesto o derivados*:

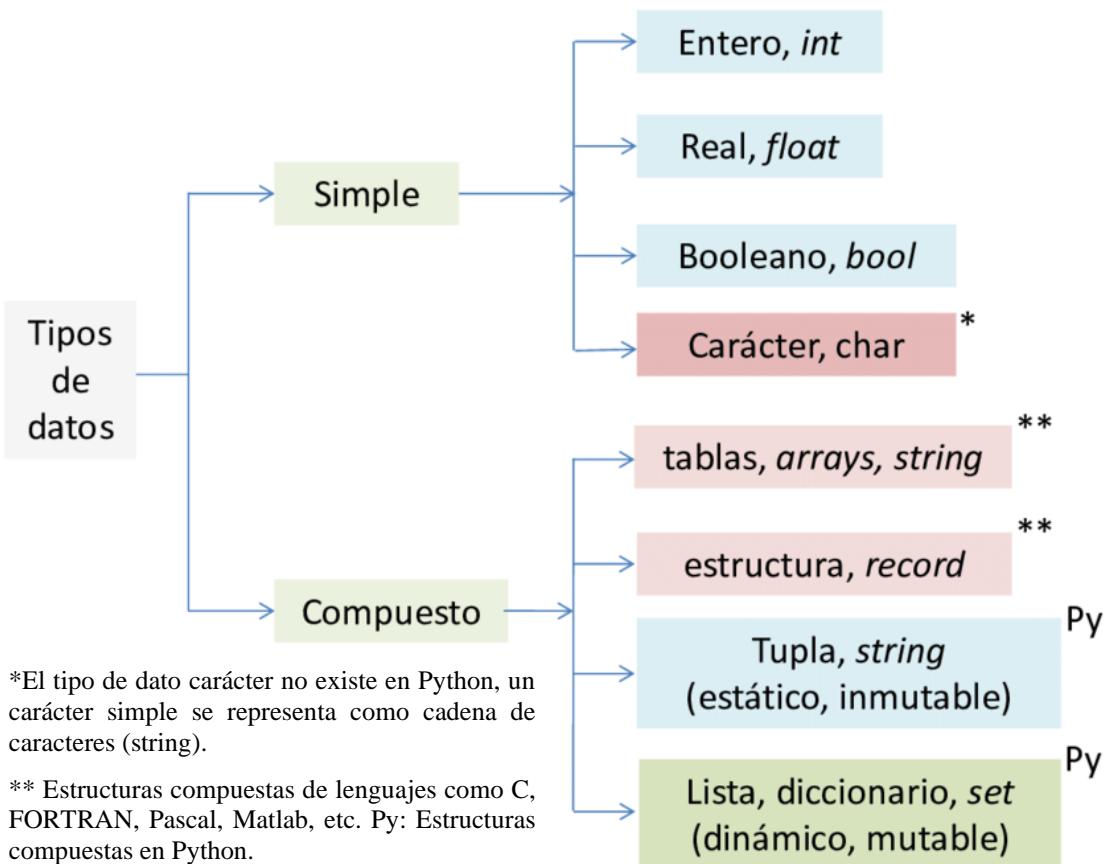


Figura 3-1. Tipos de Datos [3].

3.7.1 Primarios (o primitivos):

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*).

3.7.1.1 Números Enteros

En matemáticas los números enteros (*integer*) son los números naturales, sus negativos y el cero. Ejemplos de enteros: 5, -20, 0, -104. En Python los enteros se definen con la palabra **int** y dispone de una función interna **type** que devuelve el tipo de dato dado:

```

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

```

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

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
```

3.7.1.2 Números 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 flotante⁸, 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 separado 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

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

Así, con 64 bits se pueden representar los números (decimales) del $\pm 5.0 \cdot 10^{-324}$ (precisión) hasta $\pm 1.7 \cdot 10^{308}$ (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.

```
>>> 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
```

El "problema" de los números flotantes

El resultado de las operaciones con números flotantes puede ser inesperado:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

La causa de este problema es que Python no esconde el hecho que las operaciones que involucran números de punto flotante no son exactas debido a que hay un error inherente al pasar internamente los números de la base 2 (la que usa realmente el hardware) a la base 10. Este error ocurre con todos los lenguajes, la diferencia es que la mayoría de los lenguajes oculta este hecho usando algún tipo de redondeo. En Python este redondeo hay que hacerlo de manera explícita (si hace falta para la aplicación que se está haciendo). Una manera de contrarrestar esto es con la función incorporada (*built-in*) *round()*. Esta función requiere 2 parámetros. El primero es el número que se quiere redondear y el segundo es la precisión con

la que se quiere mostrar dicho valor. En el siguiente ejemplo se redondea el resultado de la suma a un solo decimal:

```
>>> round(0.1 + 0.1 + 0.1 - 0.3,1)  
0.0
```

Alternativamente, para no perder precisión, existe el módulo decimal. A diferencia de las operaciones de punto flotante, con este módulo las operaciones se realizan directamente en base 10. Si bien el resultado en este caso es exacto, la operación es más lenta debido a que se resuelve por software mientras que las de punto flotante aprovechan mejor el hardware. Uso del módulo decimal:

```
>>> from decimal import Decimal  
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -  
Decimal('0.3')  
Decimal('0.0')
```

Más información: <http://docs.python.org/library/decimal.html> y <http://floating-point-gui.de>

También, se pueden expresar números complejos o imaginarios, por ejemplo:

```
>>> type(5 + 5.0)  
<type 'float'>  
>>> 5 + 5.0  
10.0  
>>> type(2+3j)  
<type 'complex'>  
>>> (2+3j).real  
2.0  
>>> (2+3j).imag  
3.0
```

3.7.1.3 Cadenas.

Los textos se llaman cadena de caracteres (en inglés *string*). Los caracteres están ordenados de acuerdo a la tabla ASCII. Por ejemplo, los caracteres ASCII ordenados del valor decimal 20 al 127 son:

!	"	#	\$	%	&	'	()	*	,	-.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A	B	
C	D	E	F	G	H	I	J	K	L	M	N	Ó	P	Q	R	S	T	U	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).

3.7.1.4 Valores booleanos.

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
```

3.7.2 Derivados

- a) 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. Las listas, diccionarios, tuplas, entre otros serán tratados en temas posteriores. Estos datos se pueden subclasicar según distintos parámetros: Ordenados (o secuenciales) / Desordenados y Mutables / Inmutables.

Los tipos de datos en Python se pueden resumir en la Tabla 3-1

Tabla 3-1. Tipos de datos en Python

Tipo	Clase	Notas	Ejemplo
int	Número entero	Precisión fija, convertido en long en caso de overflow.	42
long	Número entero	Precisión arbitraria	42L ó 456966786151987643L
float	Número decimal	Coma flotante de doble precisión	3.1415927
complex	Número complejo	Parte real y parte imaginaria j.	(4.5 + 3j)
bool	Booleano	Valor booleano verdadero o falso	True o False
str	Cadena	Inmutable	'Cadena'
list	Secuencia	Mutable, puede contener objetos de diversos tipos	[4.0, 'Cadena', True]
tuple	Secuencia	Inmutable, puede contener objetos de diversos tipos	(4.0, 'Cadena', True)
set	Conjunto	Mutable, sin orden, no contiene duplicados	set([4.0, 'Cadena', True])

frozen set	Conjunto	Inmutable, sin orden, no contiene duplicados	frozenset([4.0, 'Cadena', True])
dict	Mapping	Grupo de pares clave:valor	{'key1': 1.0, 'key2': False}

Mutable: si su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.

Inmutable: si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

3.7.3 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 tengan 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 presenten 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 '
```

A continuación, algunos ejemplos del tipo de dato cadena:

```
>>> # Comillas simples
>>> cadenaa = 'Texto entre comillas simples'
>>> cadena
>>> type(cadena)
>>>
>>> # Comillas dobles
>>> cadenab = "Texto entre comillas dobles"
>>> cadenab
>>> type(cadenab)
>>>
>>> # Repetición de cadena
>>> cadrep = "Cadena" * 3
>>> cadrep
>>> type (cadrep)
>>>
>>> # Concatenación de cadena
>>> nombre = "Leonardo"
>>> apellido = "Caballero"
>>> nombre_completo = nombre + " " + apellido
>>> nombre_completo
>>> type (nombre_completo)
>>>
>>> "Tamaño de cadena '", nombre_completo, "' es:",
len(nombre_completo)
>>>
>>> # Acceder a rango de la cadena
>>> nombre_completo[3:13]
>>>
>>> # Cadena con código escapes
>>> cadenaesc = 'Texto entre \n\tcomillas simples'
>>> cadenaesc
>>> type(cadenaesc)
```

No pueden mezclarse valores numéricos con cadenas:

```
>>> 'hola' + 2
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
  'hola' + 2
TypeError: cannot concatenate 'str' and 'int' objects
>>> 'hola' + str(2)
'hola2'
```

Este error se debe a que Python es un lenguaje de tipado dinámico pero fuerte. Es dinámico porque en cualquier momento se puede definir una variable con un tipo de dato distinto. En los lenguajes de tipado estático una vez que se crea un dato de un tipo particular, esto no puede alterarse. Python es de tipado fuerte porque una vez definido, este no puede convertirse automáticamente en otro tipo de datos. Por ejemplo, en PHP '1' + 1 es igual a 2. En Python sumar un *string* con un *int* es imposible, porque Python no presupone nada con respecto a cómo hacer esta operación no definida. Lo que sí es posible es "convertir" el '1' en 1 y obtener 2 (o convertir el 1 en '1' y obtener '11').

3.7.4 Listas

Se trata de conjuntos ordenados de elementos, encerrados por corchetes y separados por comas. El orden comienza con el índice 0 para el primer lugar de la Lista. Pueden agruparse valores de distintos tipos de datos básicos, y es posible agregar, eliminar o modificar elementos de las listas en cualquier momento (las listas son mutables en Python).

```
In [7]:  
# Definición de una lista y referencia a un índice  
lista = [10,20,30,40]  
print(lista)  
print(lista[0])  
print(lista[3])  
[10, 20, 30, 40]  
10  
40  
  
In [8]:  
# Modificación de una lista  
lista[1] = 25  
print(lista)  
[10, 25, 30, 40]  
  
In [9]:  
# Generación de porciones a partir de una lista  
sublista = lista[1:3]  
print(lista)  
print(sublista)  
print(lista[:-1])  
print(lista[:])  
[10, 25, 30, 40]  
[25, 30]  
[10, 25, 30]  
[10, 25, 30, 40]
```

3.7.5 Tuplas

Las Tuplas son básicamente listas de elementos estáticas, es decir, que no pueden modificarse (decimos que las Tuplas son inmutables en Python). Para su definición, en lugar de corchetes se encierran valores separados por comas entre paréntesis.

```
In [10]:  
# Definición de una tupla y referencia a un índice  
tupla = (6, 7, 8, 9)  
print(tupla)  
print(tupla[0])  
print(tupla[3])  
(6, 7, 8, 9)  
6  
9  
In [11]:  
# Generación de porciones a partir de una tupla  
subtupla = tupla[1:3]  
print(tupla)  
print(subtupla)  
print(tupla[:-1])  
print(tupla[:])  
(6, 7, 8, 9)  
(7, 8)  
(6, 7, 8)  
(6, 7, 8, 9)
```

La similitud entre listas y tuplas es tan explícita que se puede bloquear una lista transformándola en una tupla con la función tuple() o bien desbloquear una tupla para transformarla en una lista con la función list()

```
In [12]:  
print(tuple(lista))  
print(list(tupla))  
(10, 25, 30, 40)  
[6, 7, 8, 9]
```

3.7.6 Diccionarios

En los Diccionarios cada elemento se compone de un par *clave*-*valor*, y para su definición es necesario encerrar los elementos entre llaves. Es posible acceder a un *valor* utilizando su *clave*, pero no al revés. Por este motivo, no se pueden repetir las claves para elementos distintos, pero sí es posible agregar, eliminar o modificar valores (Los diccionarios son mutables).

```
In [13]:
```

```
# Definición de un diccionario
diccionario = {"Codigo":7512,"Materia":"Análisis Numérico I"}
print(diccionario)
{'Codigo': 7512, 'Materia': 'Análisis Numérico I'}
In [14]:
# Referencia a un índice
print(diccionario["Codigo"])
print(diccionario["Materia"])
7512
Análisis Numérico I
```

3.8 Elementos Básicos del Lenguaje

3.8.1 Identificador

El **identificador** es el nombre que se le asigna a una variable, constante y otros elementos (función, clase, módulo u otro objeto) para referenciarlos en el código. Para crearlo se debe tomar en cuenta las siguientes convenciones:

- a. Debe comenzar por una letra preferiblemente en minúscula, seguida de letras y números.
- b. No puede contener caracteres especiales, excepto el subrayado.
- c. No deben tener más de 255 caracteres.
- d. No deben existir dos iguales en el mismo ámbito
- e. Python distingue entre mayúsculas y minúsculas. Es decir, que A y a son para Python variables distintas.
- f. Las palabras reservadas del lenguaje están prohibidas. En caso de que intente dar un nombre incorrecto a una variable, Python mostrará un mensaje de error al ejecutar el programa.
- g. Python permite identificadores con vocales acentuadas, como á, é, í, ó, ú y las letras griegas. Aunque no es recomendable esta práctica por si se cambia de lenguaje de programación.

Aunque no es obligatorio, normalmente conviene que el nombre de la variable esté relacionado con la información que se almacena en ella, para que sea más fácil entender el programa. Mientras se escribe un programa, esto no parece muy importante, pero si consulta un programa que ha escrito hace tiempo (o que ha escrito otra persona), te resultará mucho más fácil entender el programa si los nombres están bien elegidos.

En el caso de definir una cadena, es decir una variable que contiene valores alfanuméricos (letras, números y caracteres especiales), debe escribir su valor entre comillas ("") o apóstrofes (''), como prefiera.

Hay que tener cuidado también con las funciones internas o predefinidas del lenguaje (*built-in functions*), se pueden consultar la lista actual de funciones internas en: <https://docs.python.org/3.3/library/functions.html>. 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.

3.8.2 Palabras reservadas en Python

Los lenguajes de programación, contienen un conjunto de palabras reservadas que no pueden usarse como nombre de variable. Una lista parcial de palabras reservadas en Python es la siguiente:

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

and	as	assert	break	class	continue	def	del	elif
else	except	exec	finally	for	from	global	if	import
in	is	lambda	not	or	pass	print	raise	return
try	while	with	yield					

Indique si los siguientes identificadores son válidos.

a)Identificador	g)desviación	m)UnaVariable	s)área
b)Indice\dos	h)año	n)a(b)	t)area-rect
c)Dos palabras	i)from	o)12	u)x_____
d)___	j)var!	p)uno.dos	v)_ I
e)12horas	k)'var'	q)x	w)_____ I
f)hora12	l)import from	r)π	x)_x_x_x

Anote sus observaciones.

3.8.3 Variables

Una **variable** es un objeto o tipo de datos cuyo valor puede cambiar durante el desarrollo del algoritmo o ejecución del programa. Dependiendo del lenguaje, hay diferentes tipos de variables, tales como *enteras, reales, carácter, lógicas y de cadena*. Una variable que es de un cierto tipo puede tomar únicamente valores de ese tipo. Una variable de carácter, por ejemplo, puede tomar como valor sólo caracteres, mientras que una variable entera puede tomar sólo valores enteros [1].

Si se intenta asignar un valor de un tipo a una variable de otro tipo se producirá *un error de tipo*. Una variable se identifica por los siguientes atributos: *nombre* que lo asigna y *tipo* que describe el uso de la variable.

En Python, una variable se define con la sintaxis:

```
>>> identificador = valor_de_la_variable
```

Cada variable, tiene un nombre y un valor, el cual define a la vez, el tipo de datos de la variable.

Las variables en Python se crean cuando se definen, es decir, cuando se les asigna un valor. Para crear una variable, se escribe la variable, el signo igual y el valor que se asignará la variable. A continuación, se tiene algunos ejemplos de definiciones de variables:

```
>>> x = 42
>>> nombre = 'Pedro Pérez'
>>> cantidad = 10 + 5
>>> importe = 2.5
>>> caracter = 'p'
>>> edad = 23
>>> encontrado=True
```

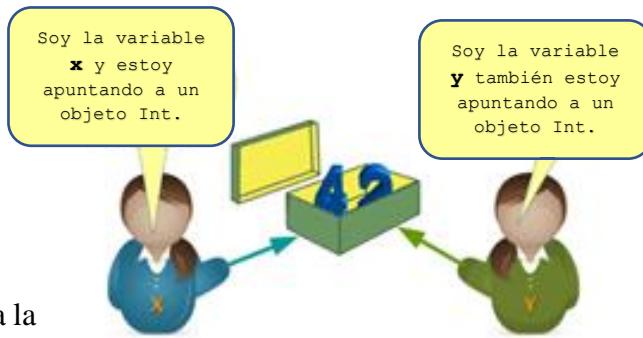


Demostrar algo más ahora. Veamos el siguiente código:

```
>>> x = 42
>>> y = x
```

Asignamos a la variable **x** el número 42. Luego, asignamos **x** a la variable **y**. Esto significa que ambas variables hacen referencia al mismo número.

¿Qué sucederá si le asignamos a la variable **y** un nuevo valor?



`y = 78`

Python creará un nuevo número entero con el valor 78 y luego la variable **y** hará referencia a este número recién creado.

En resumen: en Python una variable no se guarda directamente en la memoria; se crea un objeto (cuya posición en la memoria se llama identidad); y se asocia el identificador de la variable a la identidad de ese objeto. La Figura 3-2 muestra un ejemplo del uso de variables de un mismo valor en Python, comparado con otros lenguajes de programación [3].

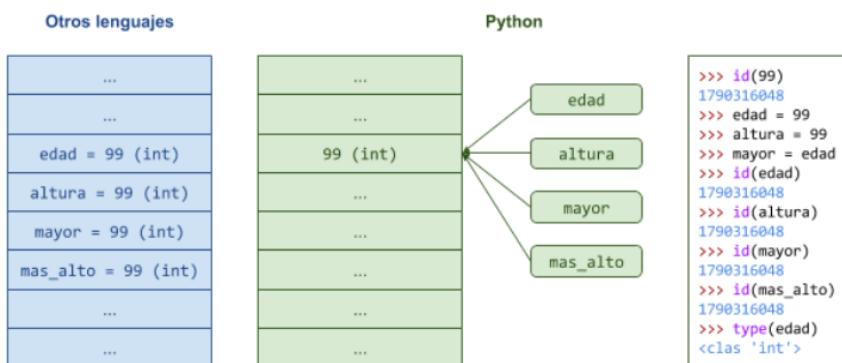
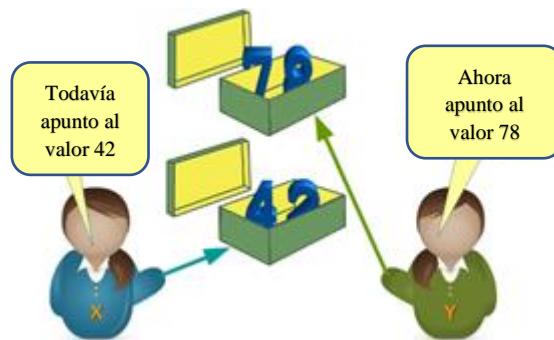


Figura 3-2. Identificador de variable y su identidad.

3.8.4 Constante

Una **constante** es un dato que permanece sin cambios durante todo el desarrollo del algoritmo o durante la ejecución del programa [1].

En Python no se puede utilizar constantes, para no modificar su valor en el código, se debe utilizar como norma nombres descriptivos y en mayúsculas separando las palabras por guiones bajos, antes y después del signo igual “=”, coloque un espacio en blanco, por ejemplo:

```
>>> ACELERACION_GRAVEDAD = 10
```

3.8.5 Variables no inicializadas

En Python, la primera operación sobre una variable debe ser la asignación de un valor. No se puede usar una variable a la que no se ha asignado previamente un valor:

```
>>> a + 2
Traceback (most recent call last)
File "<stdin>", line 1, in ?
NameError: name 'a' is not defined
```

Como puedes ver, se genera una excepción *NameError*, es decir, de “*error de nombre*”. El texto explicativo precisa aún más lo sucedido: “*name 'a' is not defined*”, es decir, “*el nombre a no está definido*”. La asignación de un valor inicial a una variable se denomina *inicialización* de la variable. Decimos, pues, que en Python no es posible usar variables no inicializadas.

3.8.6 Asignación múltiple

Otra de las ventajas que Python provee, es la de poder asignar en una sola instrucción, múltiples variables:

```
nombre, cedula, edad = 'José', 35125569, 25
```

En una sola instrucción, se declaran tres variables: nombre, cédula y edad a las cuales se le asigna un valor determinado a cada una.

3.8.7 Comentarios

La documentación de un programa es el conjunto de información interna y externa al programa, que facilitará su posterior mantenimiento y puesta a punto. La documentación puede ser *interna* y *externa*. La *documentación externa* es aquella que se realiza externamente al programa y con fines de mantenimiento y actualización; es muy importante

en las fases posteriores a la puesta en marcha inicial de un programa. La *documentación interna* es la que se acompaña en el código o programa fuente y se realiza a base de comentarios significativos. Estos comentarios se representan con diferentes notaciones, según el tipo de lenguaje de programación.

En Python, los comentarios comienzan con el signo numeral (#) seguido de un espacio. La siguiente línea representa un comentario y, por lo tanto, sería ignorada por el intérprete de Python:

```
>>> # Esto es un comentario
```

Un comentario también puede estar a continuación de código interpretable, en la misma línea, de este modo:

```
>>> GRAVEDAD = 9.8 # Aceleración de gravedad, en m/s^2
```

A continuación, se muestra un ejemplo típico de comentarios en un programa en Python denominado cabecera.

```
# Nombre: ejemplo_estructura_programa.py
# Propósito: Ejemplo de archivo con la estructura de un programa
#
# Origen: Propio
# Autor: José María Herrera-Fernández y Luis Miguel Sánchez Brea
#
# Creación: 12 de septiembre de 2013
# Historia:
#
# Dependencias: scipy, matplotlib
# Licencia: GPL
```

Esta sección es estrictamente voluntaria, sin embargo, es muy útil cuando varios usuarios programan en el mismo archivo.

Nombre	El nombre del archivo que vamos a crear cuando guardemos por primera vez. En nuestro caso <code>ejemplo_estructura_programa.py</code> .
Propósito	La finalidad que se persigue al crear el programa, * Crear gráfica del seno*, * Calcular los coeficientes de un polinomio*, etc.
Origen	Si el origen del programa no es propio conviene saber su procedencia por si se debe consultar de nuevo la fuente original en un futuro.
Autor(es)	Los nombres de los creadores del programa.

Creación	Fecha de creación del programa.
Historia	En caso de modificaciones posteriores, es en esta sección donde se indica la modificación, la fecha de la realización y el autor de la misma.
Dependencias	Módulos necesarios para la ejecución del programa. Por ejemplo: <code>scipy</code> , <code>matplotlib</code> .
Licencia	Apartado correspondiente a los derechos de autor.

PEP 8: comentarios, Comentarios en la misma línea del código deben separarse con dos espacios en blanco. Luego del símbolo `#` debe ir un solo espacio en blanco.



Punto py

Hay un convenio por el que los archivos que contienen programas Python tienen extensión `py` en su nombre. La extensión de un nombre de archivo son los caracteres del mismo que suceden al (último) punto. Un archivo llamado `ejemplo.py` tiene extensión `py`. La idea de las extensiones viene de antiguo y es un mero convenio. Puedes prescindir de él, pero no es conveniente. En entornos gráficos (como KDE, Gnome o Microsoft Windows) la extensión se utiliza para determinar qué ícono va asociado al archivo y qué aplicación debe arrancarse para abrir el archivo al hacer clic (o doble clic) en el mismo.

3.9 Operadores

Los **operadores** son aquellos elementos del lenguaje que combinan variables, constantes, valores literales, instrucciones, etc., para obtener un valor numérico, lógico, de cadena, etc., como resultado.

La combinación de operadores con variables, instrucciones, etc., se denomina **expresión**, mientras que a los elementos integrantes de una expresión y que no son operadores, se les denomina **operandos**.

En función de la complejidad de la operación a realizar, o del tipo de operador utilizado, una expresión puede ser manipulada a su vez como un operando dentro de otra expresión de mayor nivel.

Los operadores se clasifican en las categorías detalladas a continuación, según el tipo de expresión a construir.

- a) aritméticas,

- b) relacionales,
- c) lógicas,
- d) carácter.

El resultado de la expresión aritmética es de tipo numérico; el resultado de la expresión relacional y de una expresión lógica es de tipo lógico, el resultado de una expresión de cadena es de tipo cadena.

3.9.1 Operadores Aritméticos

Un operador aritmético (Tabla 3-2) toma dos operandos como entrada, realiza un cálculo y devuelve el resultado.

Considera la expresión, “ $a = 2 + 3$ ”. Aquí, **2** y **3** son los *operandos* y **+** es el *operador aritmético*. El resultado de la operación se almacena en la variable **a**.

Tabla 3-2. Operadores Aritméticos.

Nombre	Símbolo	Ejemplo	Resultado	Resultado tipo
Suma	+	$a = 10 + 5$	a es 15	Entero si a y b enteros; real si alguno es real
Resta	-	$a = 12 - 7$	a es 5	Entero si a y b enteros; real si alguno es real
Negación		$a = -5$	a es -5	De igual tipo, con signo contrario
Multiplicación	*	$a = 7 * 5$	a es 35	Entero si a y b enteros; real si alguno es real
División	/	$a = 12.5 / 2$	a es 6.25	Siempre es real
División entera	//	$a = 12.5 // 2$	a es 6.0	Devuelve la parte entera del cociente $a \div b$
Módulo	%	$a = 27 \% 4$	a es 3	Devuelve el resto de la división $a \div b$
Exponente	**	$a = 2 ** 3$	a es 8	Entero si a y b enteros; real si alguno es real

PEP 8: operadores Siempre colocar un espacio en blanco, antes y después de un operador



Operadores aritméticos

Todos los operadores aritméticos no existen en todos los lenguajes de programación; por ejemplo, en Visual Basic y en Pascal no se utiliza `//` y `%` para la división entera y el residuo. El operador exponenciación es diferente según sea el tipo de lenguaje de programación clásico elegido (`^` en Visual Basic, en Pascal para calcular la potencia $z=x^n$ se calcula aplicando propiedades de logaritmo como la siguiente: $z:=\text{Exp}(n*\text{Ln}(x))$).

Los operadores se evalúan en una expresión siguiendo la tabla de jerarquías (Tabla 3-3) que se presenta a continuación:

Tabla 3-3. Prioridad de los operadores aritméticos.

Prioridad de operadores aritméticos
Potenciación (<code>**</code>)
Negación (<code>-</code>)
Multiplicación y división real (<code>*</code> , <code>/</code>)
División entera (<code>//</code>)
Resto de división (<code>%</code>)
Suma y resta (<code>+</code> , <code>-</code>)

Nota: La potenciación es *asociativa por la derecha*. La expresión $2^{**}3^{**}2$ equivale a $2^{(3^2)} = 2^9 = 512$, y no a $(2^3)^2 = 8^2 = 64$

3.9.1.1 División Entera

En Python 2 el resultado de dividir dos números enteros se redondea al entero más cercano. Como resultado de dividir $3/2$ se obtendrá 1. Para obtener una división con punto flotante tanto el numerador como el denominador deben ser explícitamente *float*. Por tanto, $3.0/2$ o $3/2.0$ o $3.0/2.0$ su resultado es 1.5; Python 3 evalúa $3 / 2$ como 1.5 por defecto, lo cual es más intuitivo para los programadores.

3.9.1.2 Ejemplo de operadores numéricos

```
>>> a = 26
>>> b = 11.3
>>> c = 5
>>> d = 3.5

>>> # +, Añade valores a cada lado del operador
>>> a + b

>>> # -, Le resta el operando de la derecha al de la izquierda
>>> c - a
```

```

>>> # *, Multiplica los valores de ambos lados del operador
>>> d * c

>>> # **, Multiplica el operando de la izquierda tantas veces el
operador de la derecha.
>>> c ** 2

>>> # /, División real
>>> float(c) / a

>>> 7 / 3

>>> # %, Divide el operando de la izquierda por el operador del lado
derecho y devuelve el resto o residuo.
>>> 7 % 3

```

3.9.1.3 Evaluación de expresiones aritméticas.

¿Evalúe las siguientes expresiones? Presta especial atención al tipo de datos que resulta de cada operación individual. Realiza los cálculos paso a paso y comprueba el resultado con el computador.

a)	1 / 2 / 4.0
b)	1 / 2.0 / 4.0
c)	1 / 2.0 / 4
d)	1.0 / 2 / 4
e)	4 ** .5
f)	4.0 ** (1 / 2)

g)	4.0 ** (1 / 2) + 1 / 2
h)	4.0 ** (1.0 / 2) + 1 / 2.0
i)	3e3 / 10
j)	10 / 5e-3
k)	10 / 5e-3 + 1
l)	3 / 2 + 1

3.9.2 Operadores de Relación o Comparación

Los operadores de relación o de comparación permiten comparar valores de tipo numérico o cadena. Los operadores de relación sirven para expresar las condiciones en los programas.

Los operadores de relación se muestran en la Tabla 3-4. El formato general para las comparaciones es:

Expresión1 **operador de relación** Expresión2

y el resultado de la operación será True o False. Así, por ejemplo, si $a = 4$ y $b = 3$, entonces

$a > b$ el resultado es **True**

mientras que

$$(a - 2) < (b - 4) \quad \text{el resultado es } \mathbf{False}$$

Los operadores de relación (Tabla 3-4) se pueden aplicar a cualquiera de los cuatro tipos de datos estándar: enteros, real, lógico, cadena. La aplicación a valores numéricos es evidente.

Tabla 3-4. Operadores de Relación o Comparación.

Símbolo	Significado	Ejemplo	Resultado
<code>==</code>	Igual que	<code>5 == 7</code>	Falso
<code>!=</code>	Distinto que	<code>rojo != verde</code>	Verdadero
<code><</code>	Menor que	<code>8 < 12</code>	Verdadero
<code>></code>	Mayor que	<code>12 > 7</code>	Falso
<code><=</code>	Menor o igual que	<code>12 <= 12</code>	Verdadero
<code>>=</code>	Mayor o igual que	<code>4 >= 5</code>	Falso

Todos los operadores tienen el mismo nivel de jerarquía.

En Python los operadores relacionales pueden ser encadenados, tal como se acostumbra en matemáticas, de la siguiente manera:

```
>>> x = 4
>>> 0 < x <= 10
True
>>> 5 <= x <= 20
False
```

La expresión $0 < x \leq 10$ es equivalente a $(0 < x) \text{ and } (x \leq 10)$.

3.9.2.1 Ejemplo de operadores relacionales

```
>>> a = 5
>>> b = 5
>>> a1 = 7
>>> b1 = 3
>>> c1 = 3

>>> cadena1 = 'Hola'
>>> cadena2 = 'Adiós'

>>> # igual
>>> c = a == b
>>> c

>>> cadenas = cadena1 == cadena2
>>> cadenas

>>> # diferente
>>> d = a1 != b
>>> d

>>> cadena0 = cadena1 != cadena2
>>> cadena0
```

```
>>> # mayor que
>>> e = a1 > b1
>>> e

>>> # menor que
>>> f = b1 < a1
>>> f

>>> # mayor o igual que
>>> g = b1 >= c1
>>> g

>>> # menor o igual que
>>> h = b1 <= c1
>>> h
```

3.9.3 Operaciones de texto

Los operadores + y * tienen otras interpretaciones cuando sus operandos son *strings*.

+ es el operador de **concatenación** de *strings*: une dos *strings* uno después del otro:

```
>>> 'perro' + 'gato'
'perrogato'
```

La concatenación no es una suma ni una operación conmutativa.

* es el operador de **repetición** de *strings*. Recibe un operando *string* y otro entero, y entrega como resultado el *string* repetido tantas veces como indica el entero:

```
>>> 'waka' * 2
'wakawaka'
```

Otras operaciones que pueden ser útiles son: obtener el *i*-ésimo carácter de un *string* (partiendo desde cero) usando los corchetes:

```
>>> nombre = 'Pedro'
>>> nombre[0]
'P'
>>> nombre[1]
'e'
>>> nombre[2]
'd'
```

Comparar *strings* alfabéticamente con los operadores relacionales (lamentablemente no funciona con acentos y eñes):

```
>>> 'a' < 'abad' < 'abeja'
True
>>> 'zapato' <= 'alpargata'
False
```

Obtener el largo de un *string* con la función **len**:

```
>>> len('paralelepípedo')
14
>>> len('')
0
```

Verificar si un *string* está dentro de otro con el operador **in**:

```
>>> 'pollo' in 'repollos'
True
>>> 'pollo' in 'gallinero'
False
```

3.9.4 Operadores Lógicos

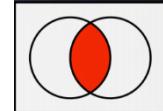
Hay tres operadores lógicos en Python: la *y lógica* o *conjunción (and)*, ver Tabla 3-5), la *o lógica* o *disyunción (or)*, ver Tabla 3-6) y el *no lógico* o *negación (not)*, ver Tabla 3-7).

El operador *and* devuelve como resultado el valor verdadero si y sólo si son ciertos sus dos operandos. Esta es su *tabla de verdad*:

Tabla 3-5. Operador lógico and.

and		resultado
operando	derecho	
izquierdo	derecho	
True	True	True
True	False	False
False	True	False
False	False	False

and	True	False
True	True	False
False	False	False

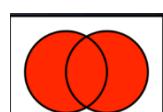


El operador *or* proporciona *True* si cualquiera de sus operandos es *True*, y *False* sólo cuando ambos operandos son *False*. Esta es su tabla de verdad:

Tabla 3-6. Operador lógico or.

or		resultado
operando	derecho	
izquierdo	derecho	
True	True	True
True	False	True
False	True	True
False	False	False

or	True	False
True	True	True
False	True	False



El operador **not** es unario, y proporciona el valor *True* si su operando es *False* y viceversa. He aquí su tabla de verdad:

Tabla 3-7. Operador lógico not.

not	
Operando	Resultado
True	False
False	True

not	
True	False
False	True

La precedencia está dada por la siguiente lista (Tabla 3-8), en que los operadores lógicos han sido listados en orden de mayor a menor precedencia:

Tabla 3-8. Prioridad de operadores lógicos.

Prioridad de operadores Lógicos
not
and
or

3.9.5 Evaluación en cortocircuito de expresiones lógicas

Cuando Python está procesando una expresión lógica, como $x \geq 2 \text{ and } (x/y) > 2$, evalúa la expresión de izquierda a derecha. Debido a la definición de *and*, si x es menor de 2, la expresión $x \geq 2$ resulta ser falsa, de modo que la expresión completa ya va a resultar falsa, independientemente de si $(x/y) > 2$ se evalúa como verdadera o falsa.

Cuando Python detecta que no se gana nada evaluando el resto de una expresión lógica, detiene su evaluación y no realiza el cálculo del resto de la expresión. Cuando la evaluación de una expresión lógica se detiene debido a que ya se conoce el valor final, eso es conocido como cortocircuitar la evaluación.

El funcionamiento en cortocircuito nos descubre una ingeniosa técnica conocida como patrón guardián. Examina la siguiente secuencia de código en el intérprete de Python:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
```

```
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

La tercera operación ha fallado porque Python intento evaluar (x/y) e y era cero, lo cual provoca un *runtime error* (error en tiempo de ejecución). Pero el segundo ejemplo *no* falló, porque la primera parte de la expresión $x \geq 2$ fue evaluada como falsa, así que (x / y) no se ejecutó debido a la regla del cortocircuito, y no se produjo ningún error.

Es posible construir las expresiones lógicas colocando estratégicamente una evaluación para validar justo antes de la evaluación que podría causar un error, como se muestra a continuación:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

En la primera expresión lógica, $x \geq 2$ es falsa, así que la evaluación se detiene en el and. En la segunda expresión lógica, $x \geq 2$ es verdadera, pero $y \neq 0$ es falsa, de modo que nunca se alcanza (x / y) .

En la tercera expresión lógica, $y \neq 0$ va *después* del cálculo de (x / y) de modo que la expresión falla con un error.

En la segunda expresión, se dice que $y \neq 0$ actúa como validación para garantizar que solo se ejecute (x / y) en el caso de que y no sea cero.



El valor None

Existe un valor especial en Python para representar aquellos casos en los que “no hay valor”; es el valor `None`. Este valor es el único de su tipo. Se interpreta como falso en las expresiones lógicas y es el devuelto por las funciones que no devuelven ningún valor explícitamente. Cuando el resultado de una expresión es `None`, el intérprete no lo muestra:

```
>>> None  
>>>
```

3.9.6 Ejercicios de Operadores Lógicos

- 1) Determinar el valor de las expresiones lógicas siguientes (y luego comprobar el resultado en el intérprete de Python):
 - a) `True and not False or False`
 - b) `False or False or True`
 - c) `not (False or False) and True`
 - d) `not False and not False and True`
- 2) Obtener las expresiones booleanas equivalentes de los enunciados siguientes:
 - a) `a` no está entre -1 y 1 ni entre 2 y 3 .
 - b) `a` y `b` no son mayores a 5 .
 - c) `a` es un múltiplo de 10 ubicado entre 3000 y 4000 .
 - d) `a` está entre $[5.0$ y $10.0]$ o entre $(15.0$ y $20.0]$.
 - e) `a` es un número impar múltiplo de 5 , o par acabado en 0 .
 - f) `a` no es el mayor ni el menor número entre `a`, `b`, y `c`.
 - g) `a`, `b`, y `c` son todas ciertas o todas falsas.
 - h) El cociente de `a` entre `b` es igual a su resto (residuo).
 - i) Los números `a`, `b`, y `c` están incluidos en un intervalo menor a la unidad.
 - j) El menor de `a` y `b` está entre 0 y 1 .
- 3) Dadas dos variables reales `x` e `y`, escribir una expresión booleana que valide que `x` está fuera del intervalo $[100, 200]$ e `y` está dentro del intervalo $(20,40]$.
- 4) Sean `A`, `B` y `C` tres variables enteras que representan las ventas de tres productos respectivamente, escribir expresiones lógicas (booleanas) que representen las siguientes afirmaciones:
 - a) Las ventas del producto `A` son las más elevadas.
 - b) Ningún producto tiene unas ventas inferiores a 300 .
 - c) Algún producto tiene unas ventas superiores a 1000 .
 - d) La media de ventas es superior a 700 .
 - e) El producto `C` no es el más vendido

- 5) Las edades de tres primos se guardan en las variables a , b , c . Escribir una expresión booleana que evalúe True si solamente 2 de los 3 primos tienen la misma edad y, además, la suma de las 3 edades sea múltiplo de 3.
- 6) Obtener las expresiones booleanas equivalentes a los enunciados siguientes:
- No se cumple z de tipo booleano o que x e y sean positivas y su suma múltiplo de 7.
 - La suma de a y b , de tipo entero, es múltiplo de 5 y al menos una de ellas es par.
 - x no es la más grande ni la más pequeña de entre x , y , z de tipo real.
 - La suma de a y b , de tipo entero, es múltiplo de 5 y ambas son impares.
 - Exactamente dos de entre x , y , z son positivas.
 - La más pequeña de las variables a y b de tipo real está entre 0 y 1.
 - Una variable a de tipo entero pertenece al conjunto de los números menores de 100 o mayores de 200 y además es un impar múltiplo de 3.
 - x , y , z de tipo entero son todas impares o todas pares.
 - ξ de tipo real es mayor que la distancia entre x e y de tipo real. $xi^{**2} > (x-y)^{**2}$ # Debe usarse un identificador válido (xi), no ξ
 - Alguno de los valores absolutos de a , b o c de tipo real es menor que 1.
 - Alguno de los valores de a , b o c de tipo real está en medio del intervalo formado por los otros dos.
 - x , y , z de tipo real pueden ser los lados de un triángulo. Los valores podrán ser un triángulo cuando el valor más grande sea menor que la suma de los otros dos.
 - No se cumple una persona trabaja (trabaja de tipo booleano) o sueldo de tipo entero está entre 1000 y 2000.

3.9.7 Prioridad de evaluación de expresiones con diferentes operadores

Cuando una expresión tiene operadores de diferentes grupos, se evaluarán de acuerdo a la Tabla 3-9 [4]:

Tabla 3-9: Características de los operadores Python. El nivel de precedencia 1 es el de mayor prioridad.

Operadores	Operación	Operador	Paridad	Asociatividad	Precedencia
Aritméticos	Exponenciación	$**$	Binario	Por la derecha	1
	Identidad	$+$	Unario	-	2
	Cambio de signo	$-$	Unario	-	2
	Multiplicación	$*$	Binario	Por la izquierda	3
	División	$/$	Binario	Por la izquierda	3

	Módulo (o resto)	%	Binario	Por la izquierda	3
	Suma	+	Binario	Por la izquierda	4
	Resta	-	Binario	Por la izquierda	4
Comparación	Igual que	==	Binario	-	5
	Distinto de	!=	Binario	-	5
	Menor que	<	Binario	-	5
	Menor o igual que	<=	Binario	-	5
	Mayor que	>	Binario	-	5
	Mayor o Igual que	>=	Binario	-	5
Lógicos	Negación	not	Unario	-	6
	Conjunción	and	Binario	Por la izquierda	7
	Disyunción	or	Binario	Por la izquierda	8

Esto significa, por ejemplo, que las multiplicaciones se evalúan antes que las sumas, y que las comparaciones se evalúan antes que las operaciones lógicas:

```
>>> 2 + 3 * 4
14
>>> 1 < 2 and 3 < 4
True
```

Operaciones dentro de un mismo nivel son evaluadas en el orden en que aparecen en la expresión, de izquierda a derecha:

```
>>> 15 * 12 % 7 # es igual a (15 * 12) % 7
5
```

La **única excepción** a la regla anterior son las potencias, que son evaluadas de **derecha** a izquierda:

```
>>> 2 ** 3 ** 2 # es igual a 2 ** (3 ** 2)
512
```

3.9.8 Uso de paréntesis para alterar la prioridad de operadores

Se Puede alterar el orden natural de prioridades entre operadores utilizando los paréntesis, encerrando entre ellos los elementos que se resuelven en primer lugar dentro de una expresión. De esta forma, se resolverán en primer lugar las operaciones que se encuentren en los paréntesis más interiores, finalizando por las de los paréntesis exteriores. Es importante tener en cuenta, que dentro de los paréntesis se seguirá manteniendo la prioridad explicada anteriormente.

3.9.9 Operadores abreviados de asignación

Estos operadores simplifican la escritura de expresiones, facilitando la creación del código. El resultado empleando operadores abreviados (Tabla 3-10) en una expresión, es el mismo que utilizando la sintaxis normal, pero con un pequeño ahorro en la escritura de código. Se utilizan en Python a partir de la versión 2.0, cuando la variable aparece a la derecha e izquierda de la instrucción de asignación.

Tabla 3-10. Operadores abreviados

Operación	Expresión Normal	Expresión Abreviada
Suma	<code>a = a + b</code>	<code>a += b</code>
Resta	<code>a = a - b</code>	<code>a -= b</code>
Multiplicación	<code>a = a * b</code>	<code>a *= b</code>
División	<code>a = a / b</code>	<code>a /= b</code>
División Entera	<code>a = a \ b</code>	<code>a //= b</code>
Potencia	<code>a= a ** b</code>	<code>a **= b</code>
Resto o residuo	<code>a= a % b</code>	<code>a %= b</code>

Por ejemplo:

```
>>> c = c + 1 # la variable c se incrementa en 1
>>> d = d - 1 # la variable d se decrementa en 1
>>> c += 1 # equivale a: c = c + 1
>>> x += 0.01 # equivale a: x = x + 0.01
>>> d -= 2 # equivale a: d = d - 2
>>> 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
```

3.9.10 Evaluación de expresiones.

Sin usar el computador, evalúe las siguientes expresiones, y para cada una de ellas indique el resultado y su tipo (si la expresión es válida) o qué error ocurre (si no lo es):

```
>>> 2 + 3 # Respuesta: tipo int, valor 5
>>> 4 / 0 # Respuesta: error de división por cero
>>> 5 + 3 * 2
>>> '5' + '3' * 2
>>> 2 ** 10 == 1000 or 2 ** 7 == 100
>>> int('cuarenta')
>>> 70/16 + 100/24
>>> 200 + 19 %
>>> 3 < (1024 % 10) < 6
>>> 'six' + 'eight'
```

```
>>> 'six' * 'eight'  
>>> float(-int('5') + int('10'))  
>>> abs(len('ocho') - len('cinco'))  
>>> bool(14) or bool(-20)  
>>> float(str(int('5' * 4) / 3)[2])
```

Compruebe sus respuestas en el computador.

3.9.11 Ejercicios Propuestos de expresiones aritméticas y lógicas.

1. Determinar el valor de las expresiones aritméticas siguientes (y luego comprobar el resultado en el intérprete de Python):

- a. $2 + 7 // 3 * 2 - 15$
- b. $32 \% 4 + 12 - 4 * 3$
- c. $42 // 8 - 3 * 14 + 6$
- d. $24/5**2 - 0.96$
- e. $3 * 5 \% 4 + 11 // 4 * 3$

2. Ejercicios sobre prioridad de operadores aritméticos

Determinar el valor de las expresiones aritméticas siguientes (y luego comprobar el resultado en el intérprete de Python):

- a) $2 + 7 // 3 * 2 - 15$
- b) $32 \% 4 + 12 - 4 * 3$
- c) $42 // 8 - 3 * 14 + 6$
- d) $24/5**2 - 0.96$
- e) $3 * 5 \% 4 + 11 // 4 * 3$

3. Ejercicios sobre prioridad de operadores lógicos

Determinar el valor de las expresiones lógicas siguientes (y luego comprobar el resultado en el intérprete de Python):

- a) True and not False or False
- b) False or False or True
- c) not (False or False) and True
- d) not False and not False and True

4. Ejercicios de revisión de expresiones

Determinar si las expresiones tienen sintaxis correcta. En caso de que sea válida la sintaxis, determinar su valor, y en el caso contrario justificar el error:

- a) $1 + 2 * 3 \% 2 - 1$
- b) $1 >= 7 \% 2$
- c) $1 \% 2 > 0.5$
- d) True or not and ($2 > 1$)
- e) $(7.5 - 3) // 3 == 1.5$
- f) $(3 // 2) > (3 \% 2)$ and not True
- g) $(3 // 3 > 3 \% 3)$ or not True
- h) $p == (4 \% 3 + 1 > 0)$ and not p

5. Ejercicios para crear expresiones lógicas

Obtener las expresiones lógicas equivalentes de los siguientes enunciados:

- a) a no está entre -1 y 1 ni entre 2 y 3 .
- b) a y b no son mayores a 5 .
- c) a es un múltiplo de 10 ubicado entre 3000 y 4000 .
- d) a está entre $[5.0$ y $10.0)$ o entre $(15.0$ y $20.0]$.
- e) a es un número impar múltiplo de 5 , o par acabado en 0 .
- f) a no es el mayor ni el menor número entre a, b, y c.
- g) a, b, y c son todas ciertas o todas falsas.
- h) El cociente de a entre b es igual a su resto (residuo).
- i) Los números a, b, y c están incluidos en un intervalo menor a la unidad.
- j) El menor de a y b está entre 0 y 1 .

6. De las expresiones aritméticas siguientes, escribir expresiones algorítmicas equivalentes en lenguaje Python. Intentar usar el mínimo de paréntesis posible

- a) $\sqrt{1 - \frac{1+x}{2a}}$
- b) $\frac{\frac{1+x}{y-1}}{\frac{z}{\sqrt[3]{(x+3)}}}$
- c) $\frac{\sqrt{a+1}}{2} + \frac{1}{2 \cdot a \cdot b}$
- d) $\frac{1}{\sqrt[3]{x}} - \frac{x^2}{y+1}$

e)
$$\frac{\frac{1}{\sqrt{x^3}} + \sqrt{y}}{-z}$$

f)
$$\frac{1}{\sqrt[3]{x}} - \frac{x^2}{y+1}$$

g)
$$\frac{\frac{4x}{2\pi f} + \frac{y^2}{2}}{2y}$$

h)
$$\frac{4x}{2\pi f} - \sqrt[3]{\frac{1}{x^5}}$$

i)
$$1 - \frac{1}{\sqrt{(x-y)^2 + (y-x)^3}}$$

j)
$$1 + \frac{1}{\sqrt[3]{x^2 + 1}}$$

7. Dadas dos variables reales x e y, escribir una expresión booleana que valide que x está fuera del intervalo [25, 35] y está dentro del intervalo (20,40].
8. Sean A, B y C tres variables enteras que representan las ventas de tres productos respectivamente, escribir expresiones lógicas (booleanas) que representen las siguientes afirmaciones:
 - a) Las ventas del producto A son las más elevadas.
 - b) Ningún producto tiene unas ventas inferiores a 300.
 - c) Algún producto tiene unas ventas superiores a 1000.
 - d) La media de ventas es superior a 700.
 - e) El producto C no es el más vendido
9. Las edades de tres primos se guardan en las variables a, b, c. Escribir una expresión booleana que evalúe True si solamente 2 de los 3 primos tienen la misma edad y, además, la suma de las 3 edades sea múltiplo de 3.
10. Obtener las expresiones booleanas equivalentes a los enunciados siguientes:
 - a) No se cumple z de tipo booleano o que x e y sean positivas y su suma múltiplo de 7.
 - b) La suma de a y b, de tipo entero, es múltiplo de 5 y al menos una de ellas es par.
 - c) x no es la más grande ni la más pequeña de entre x, y, z de tipo real.
 - d) La suma de a y b, de tipo entero, es múltiplo de 5 y ambas son impares.
 - e) Exactamente dos de entre x, y, z son positivas.
 - f) La más pequeña de las variables a y b de tipo real está entre 0 y 1.
 - g) a de tipo entero pertenece al conjunto de los números menores de 100 o mayores de 200 y además es un impar múltiplo de 3.

- h) x, y, z de tipo entero son todas impares o todas pares.
- i) ξ de tipo real es mayor que la distancia entre x e y de tipo real. $xi^{**2} > (x-y)^{**2}$
 # Debe usarse un identificador válido (xi), no ξ
- j) Alguno de los valores absolutos de a, b o c de tipo real es menor que 1.
- k) Alguno de los valores de a, b o c de tipo real está en medio del intervalo formado por los otros dos.
- l) x, y, z de tipo real pueden ser los lados de un triángulo. Los valores podrán ser un triángulo cuando el valor más grande sea menor que la suma de los otros dos.
- m) No se cumple una persona trabaja (trabaja de tipo booleano) o sueldo de tipo entero está entre 1000 y 2000.
11. Un valor entero x es positivo e impar
12. El peso se encuentra entre 50,5 y 75 Kg
13. El sexo es masculino
14. La edad es múltiplo de 10
15. Puede votar si el sexo es masculino y tiene edad para votar. Considere edad superior o igual a 21 años para votar.
16. Puede pagar si la tarjeta es de débito y el monto es superior a 20 Bs o si la tarjeta es de crédito y el monto es superior a 50 Bs. Considere Tipo de tarjeta 1: débito, 2: crédito.
17. Un punto (x, y) pertenece a una circunferencia de radio r con centro en (xc, yc) . Ecuación de una circunferencia $(x - xc)^2 + (y - yc)^2 = r^2$
18. Dada la fecha actual (da, ma, aa) y una fecha de vencimiento (dv, mv, av) el producto está vencido.
19. Dado un cuadrado de lado a su área es superior a 100. área = lado²

3.10 Funciones predefinidas

Python proporciona funciones (ver Tabla 3-11) para utilizar en las expresiones. Estas funciones se dice que están *predefinidas*.

Tabla 3-11. Simple Python Built-in Functions

Función	Descripción	Ejemplo
<code>abs(x)</code>	Devuelve el valor absoluto de x .	<code>abs(-2) es 2</code>
<code>max(x1, x2, ...)</code>	Devuelve el mayor valor de $x1, x2, ...$	<code>max(1, 5, 2) es 5</code>
<code>min(x1, x2, ...)</code>	Devuelve el menor valor de $x1, x2, ...$	<code>min(1, 5, 2) es 1</code>

<code>pow(a, b)</code>	Devuelve a^b . Same as <code>a ** b</code> .	<code>pow(2, 3) es 8</code>
<code>round(x)</code>	Devuelve un número entero más cercano a <code>x</code> . Si <code>x</code> es igualmente cercano a dos enteros, se devuelve el valor par.	<code>round(5.4) es 5</code> <code>round(5.5) es 6</code> <code>round(4.5) es 4</code>
<code>round(x, n)</code>	Devuelve el valor flotante redondeado a <code>n</code> dígitos después del punto decimal.	<code>round(5.466, 2) es 5.47</code> <code>round(5.463, 2) es 5.46</code>

El argumento de la función debe ir encerrado entre paréntesis, por ejemplo:

```
>>> abs(0)
0
>>> abs 0
SyntaxError: invalid syntax
```

A continuación, se presentan algunas funciones predefinidas:

float: conversión a flotante. Si recibe un número entero como argumento, devuelve el mismo número convertido en un flotante equivalente.

```
>>> float(3)
3.0
```

La función **float** también acepta argumentos de tipo cadena. Cuando se le pasa una cadena, **float** la convierte en el número flotante que esta representa:

```
>>> float('3.2')
3.2
>>> float('3.2e10')
32000000000.0
```

Pero si la cadena no representa un flotante, se produce un error de tipo **ValueError**, es decir, <<error de valor>>:

```
>>> float('un texto')
Traceback (innermost last):
File "<stdin>", line 1, in ?
ValueError: invalid literal for float(): un texto
```

Si **float** recibe un argumento flotante, devuelve el mismo valor que se suministra como argumento.

int: conversión a entero. Si recibe un número flotante como argumento, devuelve el entero que se obtiene eliminando la parte fraccionaria.

```
>>> int(2.1)
2
>>> int(-2.9)
-2
```

También la función **int** acepta como argumento una cadena:

```
>>> int('2')
2
```

Si **int** recibe un argumento entero, devuelve el argumento tal cual.

str: conversión a cadena. Recibe un número y devuelve una representación de este como cadena.

```
>>> str(2.1)
'2.1'
>>> str(234E47)
'2.34e+49'
```

La función **str** también puede recibir como argumento una cadena, pero en ese caso devuelve como resultado la misma cadena.

ord: acepta una cadena compuesta por un único carácter y devuelve su código ASCII (un entero).

chr: recibe un entero (entre 0 y 255) y devuelve una cadena con el carácter que tiene a dicho entero como código ASCII.

round: redondeo. Puede usarse con uno o dos argumentos. Si se usa con un sólo argumento, redondea el número al flotante más próximo cuya parte decimal sea nula.

```
>>> round(2.1)
2.0
>>> round(2.9)
3.0
>>> round(-2.9)
-3.0
>>> round(2)
2.0
```

(¡Observa que el resultado siempre es de tipo flotante!) Si **round** recibe dos argumentos, estos deben ir separados por una coma y el segundo indica el número de decimales que deseamos conservar tras el redondeo.

```
>>> round(2.1451, 2)
```

```
2.15
>>> round(2.1451, 3)
2.145
>>> round(2.1451, 0)
2.0
```

Las funciones pueden formar parte de expresiones y sus argumentos pueden, a su vez, ser expresiones. Observa los siguientes ejemplos:

```
>>> abs(-23) % int(7.3)
2
>>> abs(round(-34.2765,1))
34.3
>>> str(float(str(2) * 3 + '.123')) + '321'
'222.123321'
```

3.11 Módulos Estándar

En Python, un módulo es una pequeña parte de un gran programa. Este módulo es un archivo dentro de una unidad de almacenamiento.

Motivos por los que usar módulos:

- Se crean archivos pequeños, donde es más fácil de localizar elementos de tu código.
- Un módulo se puede usar en todos los programas que deseas. Sin necesidad de repetir código fuente.

Para un nuevo módulo se crea un archivo y se nombra como `mi_modulo.py` en este archivo incluimos el código que necesitemos, por ejemplo, funciones.

Para utilizar las funciones del módulo se escribe `import mi_modulo`, una vez incluido el módulo ya se dispone de todas las funciones que se utilizarán en el nuevo programa.

Al importar un módulo Python recorre todos los directorios indicados en la variable de entorno `PYTHONPATH` en busca de un archivo con el nombre adecuado. El valor de la variable `PYTHONPATH` se puede consultar desde Python mediante `sys.path` incluida en el módulo `sys`. A continuación, se presentan los módulos estándar de Python y las funciones más utilizadas dentro de cada módulo.

3.11.1 Obtener Ayuda sobre un Módulo

A continuación, practicarás con comandos que ofrecen información necesaria para hacer un buen uso de los módulos de Python.

El comando ***dir(nombre_modulo)*** imprime un listado con las funciones disponibles para ese módulo.

Para obtener información específica de una función incluida en un módulo, lanzaremos el comando ***help(nombre_modulo.nombre_funcion)***.

Prueba las siguientes órdenes en el intérprete de Python:

```
>>> import os  
>>> dir(os)  
>>> help(os)  
>>> help(os.getcwd)
```

3.11.2 Funciones definidas en módulos

Python también proporciona funciones trigonométricas, logaritmos, entre otras, pero no están directamente disponibles cuando iniciamos una sesión, se debe importar el módulo respectivo antes de utilizarlo.

3.11.2.1 El módulo *math*

Importar la función seno (*sin*, del inglés “*sinus*”) del módulo matemático (*math*):

```
>>> from math import sin
```

Ahora se puede utilizar la función en los cálculos:

```
>>> sin(0)  
0.0  
>>> sin(1)  
0.841470984808
```

Observa que el argumento de la función seno debe expresarse en radianes.

Inicialmente Python desconoce la función seno. Cuando se importa la función, Python carga su definición y permite utilizarla. Las definiciones de funciones residen en

módulos. Las funciones trigonométricas residen en el módulo matemático. Por ejemplo, la función coseno, en este momento, es desconocida para Python.

```
>>> cos(0)
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: cos
```

Antes de utilizarla, es necesario importarla del módulo matemático:

```
>>> from math import cos
>>> cos(0)
1.0
```

En una misma sentencia se puede importar más de una función. Basta con separar sus nombres con comas:

```
>>> from math import sin, cos
```

Puede resultar tedioso importar un gran número de funciones y variables de un módulo.

Python ofrece un atajo: si se utiliza un asterisco, se importan *todos* los elementos de un módulo. Para importar todas las funciones del módulo *math* se escribe:

```
>>> from math import *
```

Así de fácil. De todos modos, no resulta muy aconsejable por dos razones:

- 1) Al importar elemento a elemento, el programa gana en legibilidad, pues se sabe de dónde proviene cada identificador.
- 2) Si se ha definido una variable con un nombre determinado y dicho nombre coincide con el de una función definida en un módulo, la variable será sustituida por la función.

Si no conoce todos los elementos que define un módulo, es posible que esta coincidencia de nombre tenga lugar, pase inadvertido inicialmente y ocurra un error cuando se intente utilizar la variable. He aquí un ejemplo del segundo de los problemas indicados:

```
>>> pow = 1
>>> from math import *
>>> pow += 1
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +=:
'builtin_function_or_method' and 'int'
```

A continuación, se muestra algunas de las funciones del módulo *math*:

Conversión angular

<code>degrees (x)</code>	Transforma radianes a grados
<code>radians (x)</code>	Transforma g

Funciones trigonométricas

<code>math.acos(x)</code>	devuelve el coseno del arco 'x' (en radianes)
<code>math.asin(x)</code>	Seno del arco 'x'
<code>math.atan(x)</code>	arcotangente
<code>math.sin(x)</code>	devuelve en seno de 'x'. ('x' en radianes)
<code>math.cos(x)</code>	devuelve en coseno de 'x'. ('x' en radianes)
<code>math.tan(x)</code>	Tangente

Funciones hiperbólicas

<code>math.acosh(x)</code>	Devuelve el coseno hiperbólico del arco 'x' (en rad)
<code>math.asinh(x)</code>	Seno del arco hiperbólico 'x'
<code>math.atanh(x)</code>	Arcotangente hiperbólica
<code>math.cosh(x)</code>	Devuelve el coseno hiperbólico de x.
<code>math.sinh(x)</code>	Seno hiperbólico

Exponentiales y logarítmicas

<code>math.exp(x)</code>	exponencial en base e
<code>math.log(x[, base])</code>	si no le damos una base, toma e (neperiano)
<code>math.log1p(x)</code>	devuelve el logaritmo de 1+x (base e)
<code>math.log10(x)</code>	logaritmo decimal
<code>math.pow(x, y)</code>	eleva la base 'x' al exponente 'y'
<code>math.sqrt(x)</code>	raíz cuadrada de 'x'
<code>math.hypot(x, y)</code>	devuelve la norma euclídea (sqrt(x ** 2 + y ** 2))

Funciones de redondeo y caracterización

<code>math.fabs(x)</code>	valor absoluto de x
<code>math.factorial(x)</code>	factorial. Lanza ValueError si x no es entero o < 0
<code>math.floor()</code>	devuelve el mayor entero <= x
<code>math.ceil(x)</code>	devuelve el menor entero >= x
<code>math.fmod(x)</code>	resto de la división entera (de acuerdo a C)

En el módulo *math* se definen, además, algunas constantes de interés (observe que debe escribirse en minúscula):

```
>>> from math import pi, e
>>> pi
3.1415926535897931
>>> e
```

```
2.7182818284590451
```

Ejemplos del módulo math.

```
>>> import math  
>>> math.degrees(math.pi)  
180.0  
>>> math.radians(360)  
6.2831853071795862  
>>> math.exp(1)  
2.7182818284590451  
>>> math.log(4,2)  
2.0  
>>> math.log10(1000)  
3.0  
>>> math.pow(3,2.5)  
15.588457268119896  
>>> math.hypot(1,1)  
1.4142135623730951  
>>> math.fabs(3.5)  
3.5
```

```
>>> math.ceil(3.5)  
4.0  
>>> math.ceil(3.4)  
4.0  
>>> math.fmod(5,2)  
1.0  
>>> math.fmod(5.4,2)  
1.4000000000000004  
>>> math.ceil(3.5)  
4.0  
>>> math.ceil(3.4)  
4.0  
>>> math.fmod(5,2)  
1.0  
>>> math.fmod(5.4,2)  
1.4000000000000004
```

3.11.3 Evitando las coincidencias

Python ofrece un modo de evitar el problema de las coincidencias: importar sólo el módulo.

```
>>> import math
```

De esta forma, todas las funciones del módulo *math* están disponibles, pero usando el nombre del módulo y un punto como prefijo:

```
>>> import math  
>>> print (math.sin(0))  
0.0
```

Python muestra un error al intentar sumar un entero y una función. Efectivamente, hay una función *pow* en el módulo *math*. Al importar todo el contenido de *math*, la variable ha sido reemplazada por la función.

3.11.4 Evaluación de expresiones con funciones.

¿Qué resultados se obtendrán al evaluar las siguientes expresiones Python? Calcula primero a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

a) $\text{int}(\exp(2 * \log(3)))$

- b) $\text{round}(4 * \sin(3 * \pi / 2))$
- c) $\text{abs}(\log_{10}(0.01) * \sqrt{25})$
- d) $\text{round}(3.21123 * \log_{10}(1000), 3)$

3.12 Entornos de desarrollo

El código puede editarse con un simple editor de texto o utilizando el *IDLE* que hace más fácil la tarea de escribir programas. Python viene con su propio IDLE conformado por una consola y un editor de texto.

Además, hay otros buenos entornos de desarrollo más avanzados para Python: *Stanis Python Editor*, *Eric IDE*, *NinJa IDE*, [PyScripter](#), [WingIDE 101](#) o *Spyder*, entre otros.



3.12.1 Algunos IDE para Python

(<http://www.pythondiario.com/2016/11/los-5-mejores-ide-para-python.html>)

Para programar en Python solo basta con instalar Python y utilizar el IDLE que viene con la instalación o cualquier IDE que proporcionan diferentes características de apoyo para escribir y ejecutar el código.

Decir que un IDE en Python es mejor que otro es subjetivo, lo importante es que el IDE que utilice resulte cómodo y satisfaga sus necesidades. Este tema sobre **IDEs en Python** pretende mostrar entornos de trabajo completos a la hora de escribir código en Python.

3.12.1.1 Pycharm IDE

El **IDE Pycharm** es muy completo, creado por **JetBrains**. Este IDE es profesional y viene en dos modalidades: una **edición Free** y otra muy completa privada que apunta a empresas de desarrollo de software. La popularidad del **IDE Pycharm** se puede medir a

partir de que grandes empresas como **Twitter**, **Groupon**, **Spotify**, **Ebay** y **Telefónica**, han utilizado éste para su trabajo.

La mayoría de sus características están disponibles en la versión gratuita, se integra con **IPython**, soporta **Anaconda**, así como otros paquetes científicos como **matplotlib** y **NumPy**.

Características como **desarrollo remoto**, **soporte de bases de datos**, soporte de **frameworks de desarrollo web**, etc., están disponibles solo para la **edición profesional** de PyCharm.

Algo muy útil de Pycharm es su compatibilidad con múltiples marcos de desarrollo web de terceros como **Django**, **Pyramid**, **web2py**, motor de aplicaciones **Google** y **Flask**, lo que lo convierte en un competo **IDE de desarrollo** de aplicaciones rápidas.

Se encuentra disponible para **Windows** y **Linux**, puedes descargarlo desde la siguiente **URL**: <http://www.jetbrains.com/pycharm/>

Como instalar PyCharm en Windows

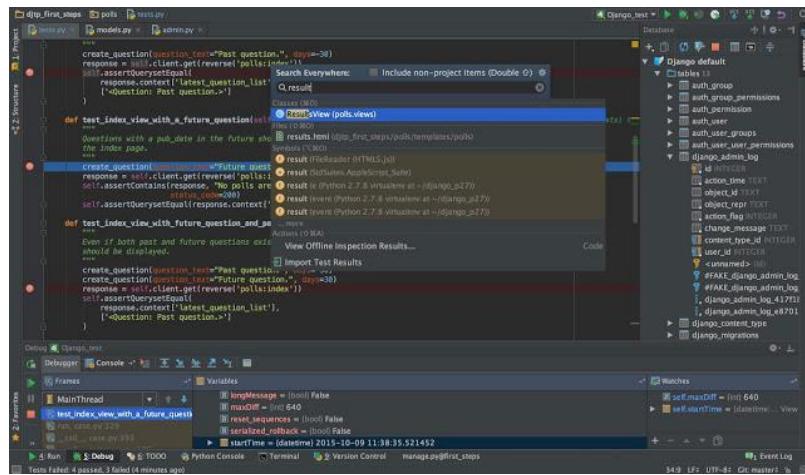
1 - Una vez finalizada la descarga, ejecute el archivo descargado, debe aparecer el asistente de instalación.

2 - Haga clic en siguiente y marque las 2 casillas.

Como instalar Pycharm en Linux

1 - Descomprima el archivo descargado en un directorio.

2 - Para ejecutar PyCharm, ejecute pycharm.sh desde el subdirectorio bin.

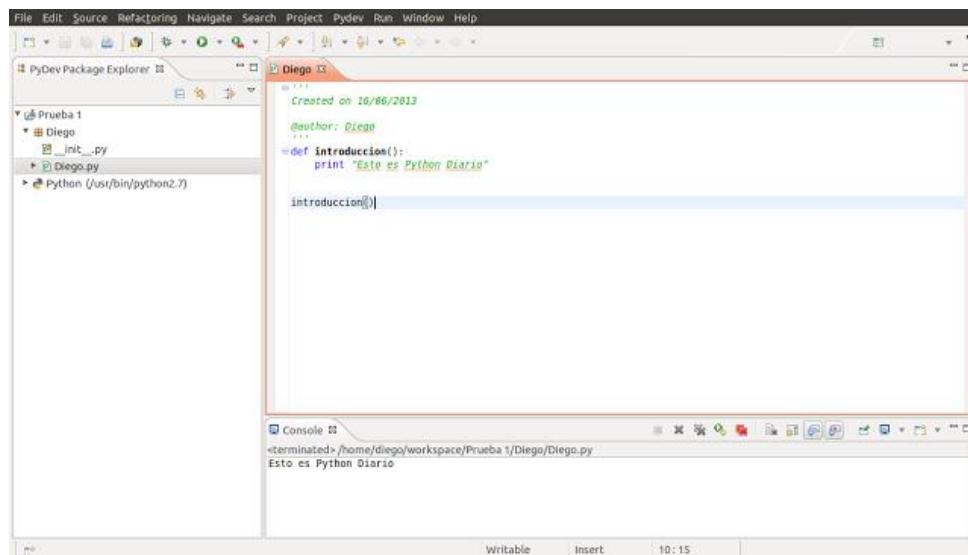


3.12.1.2 PyDev IDE

PyDev: es libre de costo y está lleno de características poderosas para programar de manera eficiente en Python. Es un plugin de código abierto y se ejecuta en **Eclipse**. Tiene integración con **Django**, completa el **código de manera automática**, soporte **multilenguaje**, plantillas de código, análisis de código, marcado de errores y mucho más.

Se mantiene siempre **actualizado** y contiene una gran comunidad de usuarios y empresas de patrocinio como **Liclipse**, **Squish**, **TraceTronic** y algunas más.

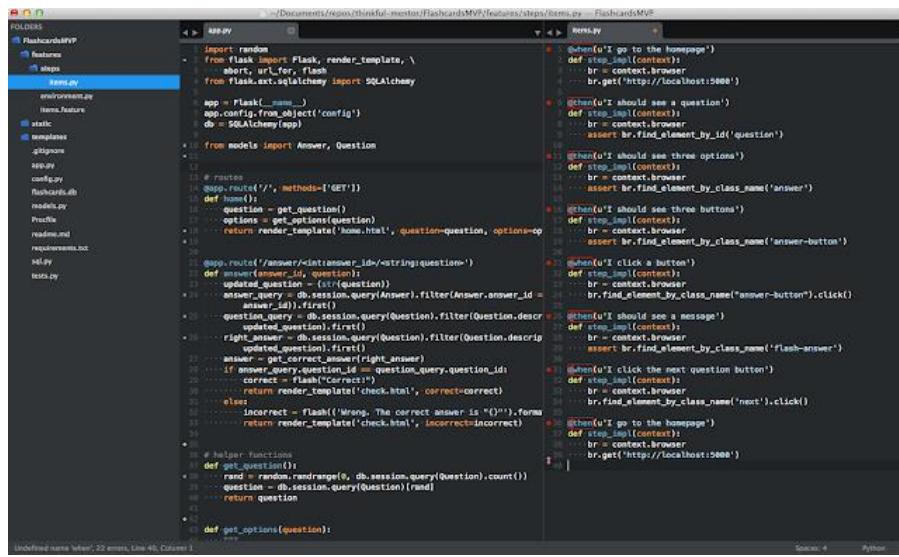
Aunque **PyDev** califica como uno de los mejores **IDE de Python de código abierto**, también viene empaquetado junto con otro producto llamado **Liclipse**, un producto comercial construido sobre Eclipse que proporciona mejoras en la usabilidad y temas adicionales.



3.12.1.3 Sublime Text 3 IDE

Este **IDE** es uno de los más **livianos y potentes**. Con el uso de *plugins*, **Sublime Text 3** puede utilizarse como un IDE completo. En un solo lugar se puede ver la elegancia del código y el poder de Python para hacer magia.

La interfaz de usuario es muy rápida y fácil de configurar. Contiene muchos paquetes para darle diferentes características. **Anaconda** es un complemento que convierte **Sublime Text 3** en un excelente **IDE Python**; aumenta su productividad y le ayuda a garantizar la calidad y estilo del código. Para más información en su página oficial: <https://www.sublimetext.com/>

A screenshot of the Sublime Text 3 interface. The left sidebar shows a project structure for a Flask application named 'FlashcardsMVP'. The main editor window displays two files: 'items.py' and 'steps.py'. 'items.py' contains Python code for a Flask application, including imports for random, flask, sqlalchemy, and models. It defines routes for '/questions' and '/answers'. 'steps.py' is a test script using the behave framework to define steps for interacting with the application. The right panel shows a browser preview of the application's homepage, displaying a question and three answer options. A status bar at the bottom indicates 'Spaces: 4' and 'Python'.

3.12.1.4 Wing IDE

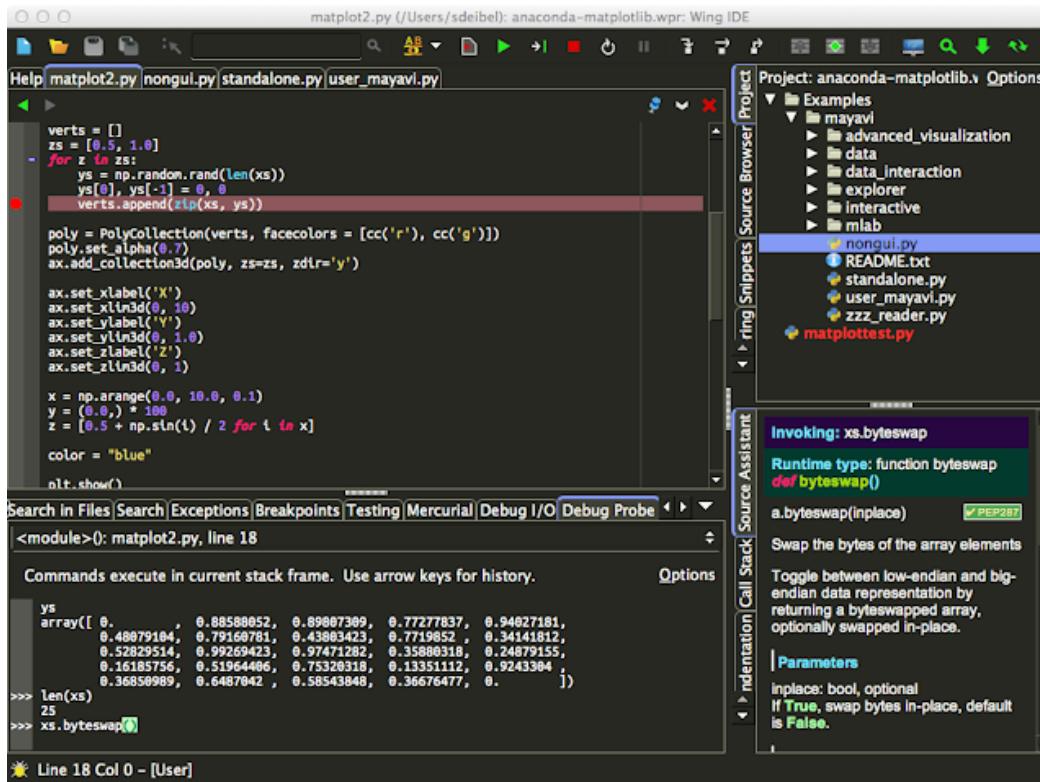
Wing IDE es comercial y apunta a **desarrolladores profesionales**. Fue lanzado hace 15 años atrás y es un producto muy maduro, con un montón de herramientas y características para **programar en Python**.

Wing IDE es soportado por Windows, OS X Linux. Como **Pycharm**, tiene una versión básica gratuita, una edición personal y una profesional muy potente.

En el **Debugging** es donde **Wing IDE** brilla más e incluye funciones como depuración de procesos múltiples, depuración de subprocessos, depuración automática de

procesos secundarios, puntos de interrupción, inspección de código, etc. También ofrece funciones para **depurar remotamente** el código que se ejecuta en **Raspberry PI**.

Wing IDE también soporta una gran cantidad de frameworks Python como: Maya, MotionBuilder, Zope, PyQt, PySide, pyGTK, Django, matplotlib entre otros. Más información en la página oficial: <https://wingware.com/>

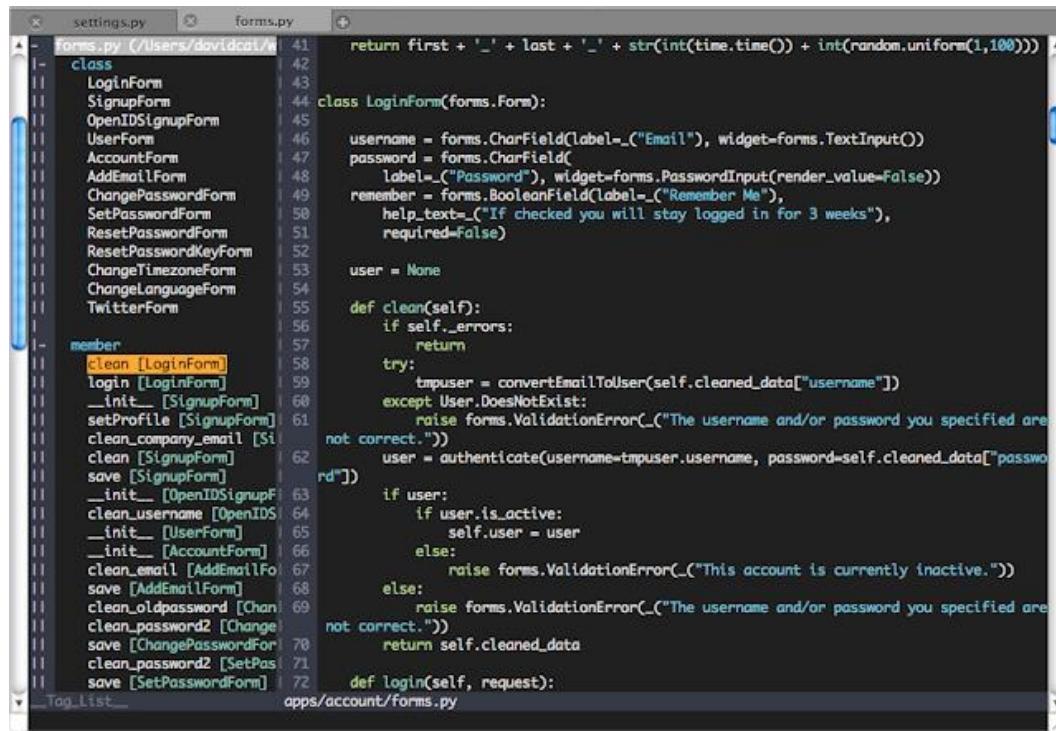


3.12.1.5 Vim IDE

Vim IDE es uno de los editores más avanzados y populares dentro de la comunidad de **desarrolladores Python**. Es de código abierto y se encuentra disponible gratuitamente bajo licencia GPL.

Sin embargo, **Vim** es más conocido como **editor**, aunque nos ofrece un entorno completo de **desarrollo para Python** cuando está configurado correctamente. **Vim** es ligero, modular y el más adecuado para los amantes del teclado, para los que no utilizan el mouse mientras se escribe código. La configuración inicial puede llevarnos un poco de tiempo ya que es necesario utilizar varios complementos **VIM** para que funcione de la manera que

queramos, pero lo que obtenemos al final vale la pena el esfuerzo. Si buscas un **IDE Python para Linux**, Vim puede ser tu mejor opción. Mas información: <http://www.vim.org/>



The screenshot shows the Vim editor with two tabs open: 'settings.py' and 'forms.py'. The 'forms.py' tab is active and displays Python code for a form class named 'LoginForm'. The code includes imports from 'forms' and defines fields for email, password, and remember me. It also includes a clean method to handle user authentication and a login function that takes a request parameter.

```
settings.py      forms.py
+- Forms.py (/Users/davidcar/... 41     return first + '_' + last + '_' + str(int(time.time()) + int(random.uniform(1,100)))
+- class
+- LoginForm
+- SignupForm
+- OpenIDSignupForm
+- UserForm
+- AccountForm
+- AddEmailForm
+- ChangePasswordForm
+- SetPasswordForm
+- ResetPasswordForm
+- ResetPasswordKeyForm
+- ChangeTimezoneForm
+- ChangeLanguageForm
+- TwitterForm
+- member
+- clean [LoginForm]
+- login [LoginForm]
+- __init__ [SignupForm]
+- setProfile [SignupForm]
+- clean_company_email [Si]
+- clean [SignupForm]
+- save [SignupForm]
+- __init__ [OpenIDSignupF]
+- clean_username [OpenIDS]
+- __init__ [UserForm]
+- __init__ [AccountForm]
+- clean_email [AddEmailFo]
+- save [AddEmailForm]
+- clean_oldpassword [Chan]
+- clean_password2 [Change
+- save [ChangePasswordFor
+- clean_password2 [SetPas
+- save [SetPasswordForm]
+- def login(self, request):
+- Tag List
+- apps/account/forms.py
```

3.12.1.6 Spyder – Anaconda (opción recomendada)

Es una distribución de código abierto de los lenguajes de programación Python y R. Se utiliza mucho para procesamiento de datos a gran escala, análisis predictivo y computación científica. Tiene como objetivo simplificar la administración y el despliegue de paquetes. Las versiones de paquetes son gestionadas por el sistema de gestión de paquetes *conda*.

Spyder “Scientific Python Development EnviRonment” en si es las IDE que usaremos con Anaconda y la recomendada por [Unipython](#). Las características útiles incluyen:

- Provisión de la consola IPython (Qt) como un mensaje interactivo, que puede mostrar gráficos en línea
- Capacidad de ejecutar fragmentos de código desde el editor de la consola
- Análisis continuo de archivos en el editor y provisión de advertencias visuales sobre posibles errores
- Ejecución paso a paso
- Explorador de variables

The screenshot shows the Spyder IDE interface. On the left, there is an 'Editor' tab with two files open: 'Interpolation.py' and 'montecarlo_pi.py'. The 'montecarlo_pi.py' file contains Python code for calculating pi using Monte Carlo integration. On the right, the 'Object inspector' panel is open, showing the documentation for the 'numpy.mean' function. The documentation includes a brief description, parameters, and examples. Below the object inspector is the 'Console' panel, which displays an IPython session with the command 'sin([1,2,3])' and its output. The bottom status bar shows permissions as 'Rw', end-of-lines as 'LF', encoding as 'UTF-8', and line/column counts.

3.12.1.7 Atom

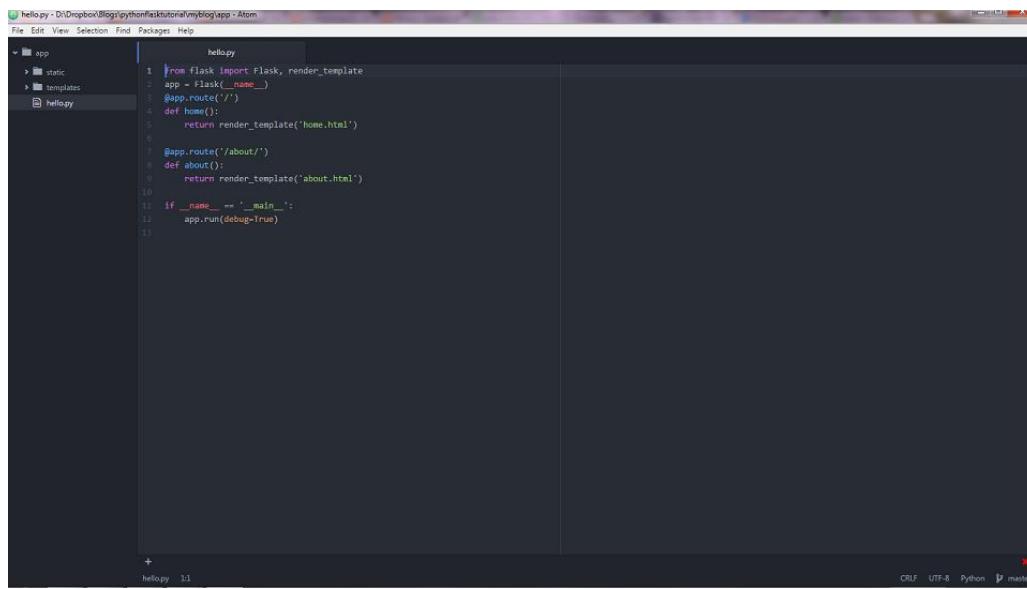
La forma más básica de crear y ejecutar un programa Python es crear un archivo vacío con extensión .py, y apuntar a ese archivo desde la línea de comandos con nombre de archivo.py. Alternativamente, puedes usar el IDLE de Python por defecto que viene instalado con Python. Puedes escribir y ejecutar tu código dentro de IDLE. Sin embargo, si desea ser productivo, las dos primeras opciones no serán las mejores. Querrías usar algo como el editor Atom.



Atom fue construido por GitHub con el lema “Un editor de texto hackeable para el siglo XXI”. Y es realmente flexible y tiene un gran soporte de paquetes externos que lo convierten en un poderoso Entorno de Desarrollo Interactivo (IDE).

Estos son los pasos que le permitirán empezar a trabajar con Atom:

Descargue e instale Atom desde <https://atom.io/> Una vez que instale *atom*, puede lanzarlo escribiendo *atom* en la línea de comandos. Si esto no funciona, asegúrese de que el átomo ha sido añadido a sus variables de trayectoria/entorno de búsqueda. El átomo debería verse así:

A screenshot of the Atom code editor interface. The title bar says "HelloPy - D:\Dropbox\Blog\pythonforiotutorial\myblog\app - Atom". The left sidebar shows a file tree with a folder "app" containing "hello.py", "static", and "templates". The main editor area displays the following Python code:

```
 1 from flask import Flask, render_template
 2 app = Flask(__name__)
 3 @app.route('/')
 4 def home():
 5     return render_template('home.html')
 6
 7 @app.route('/about/')
 8 def about():
 9     return render_template('about.html')
10
11 if __name__ == '__main__':
12     app.run(debug=True)
```

The status bar at the bottom shows "CRLF", "UTF-8", "Python", and "master".

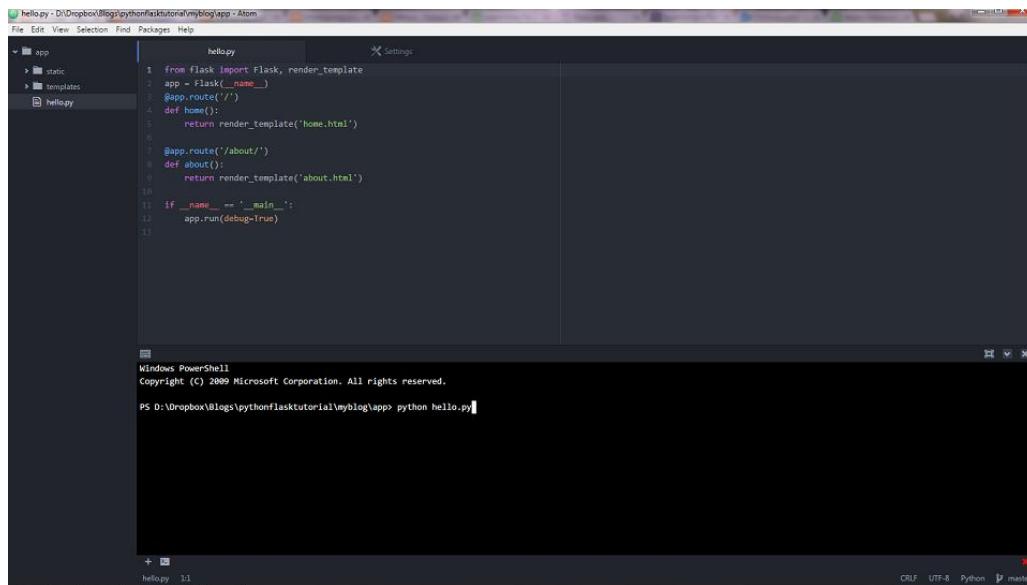
Una forma más conveniente de abrir Atom podría ser hacer clic con el botón derecho del ratón en la carpeta donde se encuentran los archivos y luego ir a Abrir con Atom. Esto añadirá todos los archivos de esa carpeta a la vista de árbol tal y como se ve en la captura de pantalla anterior. Esto es estupendo cuando el programa o la aplicación web que estás desarrollando consta de varios archivos. Puedes saltar de archivo en archivo y editarlos desde dentro de Atom.

Pero, ¿cómo ejecutar un archivo Python en Atom?

Bueno, puedes abrir la línea de comandos y apuntar a las rutas de los archivos o puedes usar un gran paquete Atom llamado plataforma-ide-terminal. que se integra con Atom para que puedas ejecutar los archivos en Atom.

Para instalar un paquete, seleccione *File -> Settings* y luego *Install and search for the name of the package* (i.e., platformio-ide-terminal). Una vez que haya instalado el paquete terminal-plus puede encontrar la herramienta y abrir una instancia de terminal seleccionado

Paquetes. Una forma más rápida de abrir un terminal es hacer clic en el signo “+” que se ha añadido en la parte inferior de la ventana de Atom. Eso debería abrir la terminal:



Como se puede observar, el terminal apunta al directorio principal de sus archivos. Ahora se pueden ejecutar scripts Python desde allí como se muestra en la captura de pantalla anterior.

Se puede dividir el editor en varias ventanas.

Cuando se tiene más de un archivo abierto dentro de Atom, puedes ir a *View -> Panes -> Split Right* para enviar el archivo actual a la mitad derecha de la ventana. Esto puede aumentar su productividad cuando trabaja con varios archivos.

3.13 Manejo de datos

3.13.1 Salida estándar

El cambio más notable y ampliamente conocido en Python 3 es como la función *print* es utilizada. El uso del paréntesis () es obligatorio en la versión 3, el mismo fue opcional en la versión 2.

```
print "Hello World" #is acceptable in Python 2
print ("Hello World") # in Python 3, print must be followed by ()
```

La función *print()* por defecto inserta al final un salto de línea. Este se puede eliminar en Python 2 escribiendo ',' al final de la función. En Python 3 "end=' '" agrega un espacio en vez de un salto de línea.

```
print x, # Trailing comma suppresses newline in Python 2  
print(x, end=" ") # Appends a space instead of a newline in Python 3
```

Los argumentos de la función *print* son los siguientes:

```
print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la sentencia *print*. En su forma más básica a la palabra clave *print* le sigue una cadena, que se mostrará en la salida estándar al ejecutarse el estamento.

```
>>> print ('Hola mundo')  
Hola mundo
```

La siguiente sentencia, por ejemplo, imprimiría la palabra “Hola”, seguida de una línea vacía (dos caracteres de nueva línea, ‘\n’), y a continuación la palabra “mundo” con sangría (un carácter tabulador, ‘\t’).

```
>>> print ('Hola\n\n\tmundo')  
Hola  
    mundo
```

Para que la siguiente impresión se realice en la misma línea se escribe una coma al final de la sentencia.

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

Esto se diferencia del uso del operador + para concatenar las cadenas en que al utilizar las comas *print* introduce automáticamente un espacio para separar cada una de las cadenas. Este no es el caso al utilizar el operador +, ya que lo que le llega a *print* es un solo argumento: una cadena ya concatenada.

```
>>> print ('Hola' + 'mundo')
```

Además, al utilizar el operador + tendríamos que convertir antes cada argumento en una cadena de no serlo ya, ya que no es posible concatenar cadenas y otros tipos, mientras que al usar el primer método no es necesaria la conversión.

```
>>> print ('Cuesta', 3, 'euros')
Cuesta 3 euros
>>> print ('Cuesta' + 3 + 'euros')
<type 'exceptions.TypeError': cannot concatenate 'str' and
'int' objects
```

3.13.1.1 Formato de salida

En el siguiente código:

```
>>> print ('%10s mundo' % 'Hola')
_____Hola mundo
>>> print ('%-10s mundo' % 'Hola')
Hola_____mundo
```

Lo que hace la primera línea es introducir los valores a la derecha del símbolo % (la cadena “mundo”) en las posiciones indicadas por los especificadores de conversión de la cadena a la izquierda del símbolo % (ver Tabla 3-12), tras convertirlos al tipo adecuado.

En la segunda línea, se pasa más de un valor a sustituir. En este ejemplo sólo se tiene un especificador de conversión: %s.

Los especificadores más sencillos están formados por el símbolo % seguido de una letra que indica el tipo con el que se formatea el valor. Por ejemplo:

Tabla 3-12. Códigos de % y su conversión

% Código	Conversión
%c	Convierte a un carácter
%s	Convierte a una cadena y aplica la función str() a la cadena antes de formatear
%i	Convierte a un entero decimal con signo
%d	Convierte a un entero decimal con signo
%u	Convierte a un entero decimal sin signo
%o	Convierte a un entero octal
%x	Convierte a un entero hexadecimal con letras minúsculas
%X	Convierte a un entero hexadecimal con letras mayúsculas
%e	Convierte a una notación exponencial con minúscula "e"
%E	Convierte a una notación exponencial con mayúscula "E"
%f	Convierte a un número real de coma flotante
%g	Convierte al valor más corto de %f y %e
%G	Convierte al valor más bajo de %f y %E

Se puede introducir un número entre el % y el carácter que indica el tipo al que formatear, indicando el número mínimo de caracteres que queremos que ocupe la cadena. Si el tamaño de la cadena resultante es menor que este número, se añadirán espacios a la

izquierda de la cadena. En el caso de que el número sea negativo, ocurrirá exactamente lo mismo, sólo que los espacios se añadirán a la derecha de la cadena.

En el caso de los números reales, es posible indicar la precisión a utilizar precediendo la letra “**f**” de un punto seguido del número de decimales a mostrar:

```
>>> from math import pi  
>>> print ('%.4f' % pi)  
3.1416
```

La misma sintaxis se puede utilizar para indicar el número de caracteres de la cadena que queremos mostrar.

```
>>> print ('%.4s' % 'hola mundo')  
hola
```

3.13.2 Caracteres especiales

Es posible que desee que las cadenas incluyan caracteres, como tabulaciones y comillas. Para hacer esto, utiliza una barra invertida “\” seguido de algún carácter único. Estos caracteres individuales junto con una barra invertida indican la presencia de un carácter especial.

La Tabla 3-13 enumera varios caracteres de escape de barra invertida junto con sus equivalentes octales, decimales y hexadecimales.

Tabla 3-13. Caracteres de escape para cadenas

ESCAPE CHARACTER	NAME	CHARACTER	DECIMAL	OCTAL	HEXADECIMAL
\n	Newline\\ Linefeed	LF	10	012	0x0A
\t	Horizontal Tab	HT	9	011	0x09
\b	Backspace	BS	8	010	0x08
\0	Null character	NUL	0	000	0x00
\a	Bell	BEL	7	007	0x07
\v	Vertical tab	VT	11	013	0x0B
\r	Carriage return	CR	13	015	0x0D
\e	Escape	ESC	27	033	0x1B
\"	Double quote	"	34	042	0x22

v	Single quote	'	39	047	0x27
\f	Form feed	FF	12	014	0x0C
\\"	Backslash	\	92	134	0x5C

Consideremos un ejemplo del carácter *NEWLINE* (\n).

```
>>> print ('Hello \nworld')
Hello
world!!!
```

Tenga en cuenta que \n no se imprime en la salida. También hay una nueva línea antes del comienzo de la segunda palabra. Además, cuando tiene una declaración larga que excede una sola línea, puede usar una barra diagonal inversa “\” para escapar de *NEWLINE* y continuar la instrucción. Por ejemplo,

```
>>> print ('The name for the registration number %s is %s' \
... % ('A002', 'Steve'))
The name for the registration number A002 is Steve
```

Para imprimir caracteres especiales, no solo usa los caracteres con una barra diagonal inversa, sino también sus valores decimales, octales y hexadecimales. Por ejemplo,

```
>>> print ("Hello \012world)
Hello
world!! !
```

3.13.3 Lectura de Datos por Teclado

Python 2 tiene dos versiones de función de entrada, *input()* y *raw_input()*. La función *input()* trata los datos recibidos como *string* si está encerrado entre comillas " o "", sino el dato es considerado como un número.

```
In Python 2
>>> x = input('something:')
something:10 #entered data is treated as number
>>> x
10
>>> x = input('something:')
something:'10' #entered data is treated as string
>>> x
'10'
>>> x = raw_input("something:")
something:10 #entered data is treated as string even without ''
>>> x
'10'
>>> x = raw_input("something:")
something:'10' #entered data treated as string including ''
>>> x
"'10'"
```

En Python 3 la función *raw_input()* es obsoleta. Además, que el dato recibido era siempre tratado como un *string*.

La función *input()* permite obtener texto escrito por teclado. Al llegar a la función, el programa se detiene esperando que se escriba algo y se pulse la tecla *Enter*, como muestra en los siguientes ejemplos:

```
In Python 3
>>> x = input("something:")
something:10
>>> x
'10'
>>> x = input("something:")
something;'10' #entered data treated as string with or
without '
>>> x
"'10'"
>>> x = raw_input("something:") # will result NameError
Traceback (most recent call last):
File "", line 1, in
X = raw_input("something:")
NameError: name 'raw_input' is not defined
```

Para incluir la pregunta en la solicitud de la respuesta, se utiliza la concatenación (operador +):

```
nombre = input('Introduzca el nombre: ')
apellido = input('Introduzca el apellido, ' + nombre + ': ')
print('Me alegro de conocerle,', nombre, apellido)
```

Otra solución es utilizar las cadenas "f" introducidas en Python 3.6:

```
nombre = input('Introduzca el nombre: ')
apellido = input(f'Introduzca el apellido, {nombre}: ')
print(f'Me alegro de conocerle, {nombre} {apellido}')
```

Otro ejemplo:

```
numero1 = int(input('Introduzca un número: '))
numero2 = int(input(f'Introduzca un número mayor que {numero1}: '))
print(f'La diferencia entre ellos es {numero2 - numero1}')
```

3.14 Glosario

Algoritmo: Un proceso general para resolver una clase completa de problemas.

Análisis sintáctico: La examinación de un programa y el análisis de su estructura sintáctica.

Asignación: sentencia que asigna un valor a una variable.

Código de objeto: La salida del compilador una vez que ha traducido el programa.

Código fuente: es un conjunto de instrucciones y órdenes lógicas, compuestos de algoritmos que se encuentran escritos en un determinado lenguaje de programación, las cuales deben ser interpretadas o compiladas, para permitir la ejecución del programa informático.

Código fuente: Un programa escrito en un lenguaje de alto nivel antes de ser compilado.

Coma flotante: un formato para representar números con decimales.

Comentario: un segmento de información en un programa, destinado a otros programadores (o cualquiera que lea el código fuente) y que no tiene efecto sobre la ejecución del programa.

Compilar: Traducir un programa escrito en un lenguaje de alto nivel a un lenguaje de bajo nivel todo al mismo tiempo, en preparación para la ejecución posterior.

Composición: la capacidad de combinar expresiones sencillas y sentencias hasta crear sentencias y expresiones compuestas, con el fin de representar cálculos complejos de forma concisa.

Concatenar: Unir dos operandos, extremo con extremo.

Depuración: El proceso de hallazgo y eliminación de los tres tipos de errores de programación.

Diagrama de estado: representación gráfica de un conjunto de variables y de los valores a los que se refiere.

División de enteros: es una operación que divide un entero entre otro y devuelve un entero. La división de enteros devuelve sólo el número entero de veces que el numerador es divisible por el denominador, y descarta el resto.

Error (bug): Un error en un programa.

Error en tiempo de ejecución: Un error que no ocurre hasta que el programa ha comenzado a ejecutarse e impide que el programa continúe.

Error semántico: Un error en un programa que hace que ejecute algo que no era lo deseado.

Error sintáctico: Un error en un programa que hace que el programa sea imposible de analizar sintácticamente (e imposible de interpretar).

Evaluar: simplificar una expresión ejecutando las operaciones para entregar un valor único.

Excepción: Otro nombre para un error en tiempo de ejecución.

Expresión: una combinación de variables, operadores y valores. Dicha combinación representa un único valor como resultado.

Guion: Un programa archivado (que va a ser interpretado).

Interpretar: Ejecutar un programa escrito en un lenguaje de alto nivel traduciéndolo línea por línea

Lenguaje de alto nivel: Un lenguaje como Python diseñado para ser fácil de leer y escribir para la gente.

Lenguaje de bajo nivel: Un lenguaje de programación diseñado para ser fácil de ejecutar para un computador; también se lo llama “lenguaje de máquina” o “lenguaje ensamblador”.

Lenguaje de programación: es un lenguaje informático, diseñado para expresar órdenes e instrucciones precisas, que deben ser llevadas a cabo por una computadora. El mismo puede utilizarse para crear programas que controlen el comportamiento físico o lógico de un ordenador. Está compuesto por una serie de símbolos, reglas sintácticas y semánticas que definen la estructura del lenguaje.

Lenguaje formal: Cualquier lenguaje diseñado por humanos que tiene un propósito específico, como la representación de ideas matemáticas o programas de computadores: todos los lenguajes de programación son lenguajes formales.

Lenguaje informático: Es un idioma artificial, utilizado por ordenadores, cuyo fin es transmitir información de algo a alguien. Los lenguajes informáticos, pueden clasificarse en: a) lenguajes de programación (Python, PHP, Pearl, C, etc.); b) lenguajes de especificación (UML); c) lenguajes de consulta (SQL); d) lenguajes de marcas (HTML, XML); e) lenguajes de transformación (XSLT); f) protocolos de comunicaciones (HTTP, FTP); entre otros.

Lenguaje natural: Cualquier lenguaje hablado que evolucionó de forma natural.

Lenguajes de alto nivel: son aquellos cuya característica principal, consiste en una estructura sintáctica y semántica legible, acorde a las capacidades cognitivas humanas. A diferencia de los lenguajes de bajo nivel, son independientes de la arquitectura del hardware, motivo por el cual, asumen mayor portabilidad.

Lenguajes interpretados: a diferencia de los compilados, no requieren de un compilador para ser ejecutados sino de un intérprete. Un intérprete, actúa de manera casi idéntica a un compilador, con la salvedad de que ejecuta el programa directamente, sin necesidad de generar previamente un ejecutable. Ejemplo de lenguajes de programación interpretado son Python, PHP, Ruby, Lisp, entre otros.

Multiparadigma: acepta diferentes paradigmas (técnicas) de programación, tales como la orientación a objetos, aspectos, la programación imperativa y funcional.

Multiplataforma: significa que puede ser interpretado en diversos Sistemas Operativos como GNU/Linux, Windows, Mac OS, Solaris, entre otros.

Operador: un símbolo especial que representa un cálculo sencillo, como la suma. La multiplicación o la concatenación de cadenas.

Operando: uno de los valores sobre los que actúa un operador.

Palabra reservada: es una palabra clave que usa el compilador para analizar sintácticamente los programas. No pueden usarse palabras reseñadas, por ejemplo, *if*, *def* y *while* como nombres de variables.

Portabilidad: La cualidad de un programa que le permite ser ejecutado en más de un tipo de computador.

Programa ejecutable: Otro nombre para el código de objeto que está listo para ejecutarse.

Programa: Un conjunto de instrucciones que especifica una computación.

Reglas de precedencia: la serie de reglas que especifican el orden en el que las expresiones con múltiples operadores han de evaluarse.

Semántica: El significado de un programa.

Sentencia print: Una instrucción que causa que el intérprete Python muestre un valor en el monitor.

Sentencia: es una porción de código que representa una orden o acción. Hasta ahora, las sentencias que hemos visto son las asignaciones y las sentencias *print*.

Sintaxis: La estructura de un programa.

Solución de problemas: El proceso de formular un problema, hallar la solución y expresar esa solución.

Tipado dinámico: un lenguaje de tipado dinámico es aquel cuyas variables, no requieren ser definidas asignando su tipo de datos, sino que éste, se auto-asigna en tiempo de ejecución, según el valor declarado.

Tipo: un conjunto de valores. El tipo de un valor determina cómo puede usarse en las expresiones. Hasta ahora, los tipos que hemos visto son enteros (tipo *int*), números de coma flotante (tipo *float*) y cadenas (tipo *string*).

Unidad: Uno de los elementos básicos de la estructura sintáctica de un programa, análogo a una palabra en un lenguaje natural.

Valor: un número o cadena (o cualquier otra cosa que se especifique posteriormente) que puede almacenarse en una variable o calcularse en una expresión.

Variable: nombre que hace referencia a un valor.

3.15 Ejercicios Resueltos de Estructura Secuencial

Ejercicio 3-1: Realiza un programa que lea 2 números por teclado y determine los siguientes aspectos (es suficiente con mostrar True o False).

Si los dos números son iguales

Si los dos números son diferentes

Si el primero es mayor que el segundo

Si el segundo es mayor o igual que el primero

Código 3-1. comp_num.py 

```

# Propósito: Comparar dos números
# Autor:      Alejandro Bolívar
# Fecha:     30/01/2020

# Entrada de datos
n1 = float(input("Introduce el primer número: "))
n2 = float(input("Introduce el segundo número: "))

print("¿Son iguales? ", n1 == n2)
print("¿Son diferentes?", n1 != n2)
print("¿El primero es mayor que el segundo?", n1 > n2)
print("¿El segundo es mayor o igual que el primero?", n1 <= n2)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Ejecución:

Introduce el primer número: 10
 Introduce el segundo número: 10
 ¿Son iguales? True
 ¿Son diferentes? False
 ¿El primero es mayor que el segundo? False
 ¿El segundo es mayor o igual que el primero? True

Ejercicio 3-2: Utilizando operadores lógicos, determina si una cadena de texto introducida por el usuario tiene una longitud mayor o igual que 3 y a su vez es menor que 10 (es suficiente con mostrar True o False).

Código 3-2. tam_cad.py

```

# Propósito: Determinar tamaño de una cadena
# Autor:      Alejandro Bolívar
# Fecha:     30/01/2020

# Entrada de datos
cadena = input("Escribe una cadena: ")

print("¿La longitud de la cadena es mayor o igual que 3 y menor que 10?", 
      len(cadena) >= 3 and len(cadena) < 10)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Ejecución:

Escribe una cadena: Hola
 ¿La longitud de la cadena es mayor o igual que 3 y menor que 10? True

Ejercicio 3-3: Realiza un programa que cumpla el siguiente algoritmo utilizando siempre que sea posible operadores en asignación:

- a) Lee por pantalla otro numero_usuario, especifica que sea entre 1 y 9
- b) Multiplica el numero_usuario por 9 en sí mismo
- c) Guarda en una variable numero_magico el valor 12345679 (sin el 8)
- d) Multiplica el numero_magico por el numero_usuario en sí mismo
- e) Finalmente muestra el valor final del numero_magico por pantalla

Código 3-3. num_mag.py 

```
# Propósito: Determinar el número mágico
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos
numero_usuario = int(input("Introduce un número del 1 al 9: "))

numero_usuario *= 9
numero_magico = 12345679
numero_magico *= numero_usuario
print("El número mágico es:", numero_magico)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecución:

```
Introduce un número del 1 al 9: 5
El número mágico es: 555555555
```

Ejercicio 3-4: En el siguiente ejemplo se convierte grados Celsius a Fahrenheit; al ejecutarlo e introducir por ejemplo el valor 27, se muestra el resultado en la consola:

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logró?
Valor de una temperatura en grados Centígrados	El valor de dicha temperatura en grados Fahrenheit	${}^{\circ}\text{F} = 9/5 * {}^{\circ}\text{C} + 32$

Código 3-4. conv_temp.py 

```
# Propósito: Conversión de temperatura
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020
```

```

# Programa que convierte °C a °F

# Entrada de datos
c = float(input("Entra temperatura en °C: "))

# Procesamiento de datos
f = 9 / 5 * c + 32

# Salida de datos
print(c, " °C equivale a ", f, " °F")

```

Ejecución:

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

Ejercicio 3-5: El programa siguiente calcula el área y el perímetro de un círculo a partir de su radio. Como se requiere el uso del valor π (pi) lo importaremos del módulo de funciones y constantes matemáticas de Python (math). Se introducirá del teclado el radio de valor 1.2.

Código 3-5. área_per_cir.py 

```

# Propósito: Determinar el área y el perímetro de un círculo
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Programa que calcula perímetro y área de un círculo
from math import pi

# Entrada de datos
r = float(input('Introduce el radio del círculo: '))

# Procesamiento de datos
per = 2 * pi * r
area = pi * r ** 2

#Salida de datos
print('Perímetro =', per)
print('Área = ', area)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Ejecución:

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 argumento 2 por el número de decimales que se requiera.

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

Ejecución:

Introduce el radio del círculo: 1.2

Perímetro = 7.54

Área = 4.52

Ejercicio 3-6: Ejercicio de conversión de segundos. Diseña un programa que lea una cantidad de segundos (valor entero) y calcule el número de días, horas, minutos y segundos equivalentes.

Código 3-6. conv_tiempo.py

```
# Propósito: Conversión de tiempo
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Programa para convertir segundos a días, horas, minutos, y segundos

# Entrada de datos
s = int(input("Entra segundos: "))
auxs = s
# Procesamiento de datos
m = s // 60
s = s % 60
h = m // 60
m = m % 60
d = h // 24
h = h % 24

#Salida de datos
print(auxs, 'segundos son:')
print(d, "días,", h, "horas,", m, "minutos y", s, "segundos")
```

Ejecución 1:

Entra segundos: 3676

3676 segundos son:

0 días, 1 horas, 1 minutos y 16 segundos

Ejecución 2:

Entra segundos: 400000
400000 segundos son:
4 días, 15 horas, 6 minutos y 40 segundos

Ejercicio 3-7: Ejercicios de cambio de monedas. Diseña un programa que reciba una cierta cantidad de céntimos (c) y retorne su equivalente en el mínimo número de monedas de curso legal (2 euros, 1 euro, 50 céntimos, 20 céntimos, 10 céntimos, 5 céntimos, 2 céntimos y 1 céntimo).

Código 3-7. convertirmonedas.py

```
# Propósito: Conversión de denominación de monedas
# Autor:      Alejandro Bolívar
# Fecha:     30/01/2020

# Programa céntimos a monedas de curso legal

# Entrada de datos
c = int(input("Entra céntimos: "))
auxc = c
# Procesamiento de datos
e2 = c // 200
c = c % 200
e = c // 100
c = c % 100
c50 = c // 50
c = c % 50
c20 = c // 20
c = c % 20
c10 = c // 10
c = c % 10
c5 = c // 5
c = c % 5
c2 = c // 2
c = c % 2

#Salida de datos
print(auxc, 'céntimos equivalen a:')
print(e2, 'monedas de 2 €,', e, 'de 1 €,', c50, 'de 50 cts,', c20, 'de 20 cts,',)
print(c10, 'monedas de 10 cts,', c5, 'de 5 cts,', c2, 'de 2 cts y', c, 'de 1'
céntimo')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecución 1:

Entra céntimos: 589
589 céntimos equivalen a:

2 monedas de 2 €, 1 de 1 €, 1 de 50 cts, 1 de 20 cts,
 1 monedas de 10 cts, 1 de 5 cts, 2 de 2 cts y 0 de 1 céntimo
 Pulse una tecla para finalizar...

Ejecución 2:

Entra céntimos: 399

399 céntimos equivalen a:

1 monedas de 2 €, 1 de 1 €, 1 de 50 cts, 2 de 20 cts,
 0 monedas de 10 cts, 1 de 5 cts, 2 de 2 cts y 0 de 1 céntimo
 Pulse una tecla para finalizar...

Ejercicio 3-8: Dada la longitud de los catetos de un triángulo rectángulo, utilizando el teorema de Pitágoras ($H^2=C1^2+C2^2$), desarrolle un programa que determine el valor de la longitud de la hipotenusa.

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logro?
Longitud Cateto 1 (C1) Longitud Cateto 2 (C2) El hecho que el triángulo es rectángulo	La longitud de la hipotenusa (H)	Usando el teorema de Pitágoras

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Longitud del Cateto 1	c1	<input type="radio"/> Entero <input type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena
Longitud del Cateto 2	c2	<input type="radio"/> Entero <input type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena
Longitud de la Hipotenusa	h	<input type="radio"/> Entero <input type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena

Código 3-8. hipotenusa.py

```
# Propósito: Determinar la hipotenusa.
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos
c1 = float(input("Longitud del primer cateto:"))
c2 = float(input("Longitud del segundo cateto:"))

# Procesamiento de datos
h = (c1 ** 2 + c2**2)**0.5

#Salida de datos
print("Longitud de la hipotenusa= ", h)
```

```
# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecución 1:

Longitud del primer cateto:3
 Longitud del segundo cateto:4
 Longitud de la hipotenusa= 5.0
 Pulse una tecla para finalizar...

Ejercicio 3-9: Dado el valor de un ángulo, expresado en radianes, desarrolle un programa que determine e imprima el valor de dicho ángulo en grados, sabiendo que $180^\circ = n$ radianes.

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logró?
Valor del ángulo La unidad de medida del ángulo es en radianes	El valor de ángulo medido en grado	Ángulo en ${}^\circ$ = ángulo en radianes * $180/n$

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Valor del ángulo en Grados	angulo1	O Entero ® Real O Lógico O Cadena
Valor del ángulo en Radianes	angulo2	O Entero ® Real O Lógico O Cadena

Código 3-9. convierteangulo.py

```
# Propósito: Determinar el ángulo
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos
angulo1 = float(input("Valor en Radianes:"))

# Procesamiento de datos
angulo2 = angulo1 * 180 / 3.141516

#Salida de datos
print("Valor en Grados es= ", angulo2)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecución 1:

Valor en Radianes:3.14

Valor en Grados es= 179.9131374788478

Pulse una tecla para finalizar...

Ejercicio 3-10: Dada la longitud de la base y la altura de triángulo cualesquiera, desarrolle un programa que determine el área del triángulo, sabiendo que el área=base*altura/2

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logro?
Longitud de la base Longitud de la altura Cualquier forma de triángulo	El valor del área que encierra el triángulo	base*altura área= 2

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Longitud de la Base	base	O Entero ® Real O Lógico O Cadena
Longitud de la altura	altura	O Entero ® Real O Lógico O Cadena
Área del Triángulo	area	O Entero ® Real O Lógico O Cadena

Código 3-10. área_triang.py

```
# Propósito: Determinar el área de un triángulo rectángulo
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos
base = float(input('Longitud de la Base:'))
altura = float(input('Longitud de la altura:'))

# Procesamiento de datos
area = base * altura / 2

# Salida de datos
print("Área del Triángulo= ", area)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecución:

Longitud de la Base:3

Longitud de la altura:4

Área del Triángulo= 6.0

Pulse una tecla para finalizar...

Ejercicio 3-11: Dado el largo y ancho de un rectángulo, desarrolle un programa que determine e imprima el área y el perímetro del rectángulo, sabiendo que el área= largo*ancho y el perímetro=2*Largo + 2*ancho.

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logro?
Longitud del largo del rectángulo Longitud del ancho del rectángulo	El valor del área que encierra el rectángulo El valor del perímetro del rectángulo	Área=largo*ancho Perímetro=2*Largo + 2*ancho

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Longitud del Largo	largo	O Entero ® Real O Lógico O Cadena
Longitud del Ancho	ancho	O Entero ® Real O Lógico O Cadena
Área del Rectángulo	area	O Entero ® Real O Lógico O Cadena
Perímetro del Rectángulo	perim	O Entero ® Real O Lógico O Cadena

Código 3-11. área_per_rect.py

```
# Propósito: Determinar el área y el perímetro de un rectángulo
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos
largo = float(input('Longitud del Largo del rectángulo:'))
ancho = float(input('Longitud del Ancho del rectángulo:'))

# Procesamiento de datos
area = largo * ancho
perim = 2 * largo + 2 * ancho

#Salida de datos
print(f'Área del Rectángulo= {area}')
print(f'Perímetro del rectángulo= {perim}')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejercicio 3-12: Dadas las longitudes de los lados de un triángulo (l_1 , l_2 y l_3), desarrolle un programa que determine el área que encierra el triángulo utilizando el método del semiperímetro, donde el semiperímetro se calcula como $s = (l_1+l_2+l_3) / 2$ y el área como $(s*(s-l_1)*(s-l_2)*(s-l_3))^{0.5}$**

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logro?
Longitud del Lado L1 Longitud del Lado L2 Longitud del Lado L3	1. El valor del área por el método del semiperímetro	$s=(l_1+l_2+l_3)/2$ $area=(s*(s-l_1)*(s-l_2)*(s-l_3))^{**0.5}$

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Longitud del Lado L1	l1	O Entero ® Real O Lógico O Cadena
Longitud del Lado L2	l2	O Entero ® Real O Lógico O Cadena
Longitud del Lado L3	l3	O Entero ® Real O Lógico O Cadena
Área del Triángulo	area	O Entero ® Real O Lógico O Cadena
Valor del Semiperímetro	s	O Entero ® Real O Lógico O Cadena

Código 3-12. área_triang.py

```
# Propósito: Determinar el área de un triángulo
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos
l1 = float(input('Longitud del Lado 1:'))
l2 = float(input('Longitud del Lado 2:'))
l3 = float(input('Longitud del Lado 3:'))

# Procesamiento de datos
s = (l1 + l2 + l3) / 2
area = ((s - l1) * (s - l2) * (s - l3)) ** 0.5

# Salida de datos
print(f'Área del Rectángulo: {área}')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejercicio 3-13: Dadas las longitudes del radio de la base y la altura de un cilindro, determine el área y el volumen del cilindro, sabiendo que el área = $2\pi r^2 h$ y el volumen = $\pi r^2 h$.

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logró?
Longitud del radio de la base	El valor del área del cilindro	Área = pi * Radio ²
Longitud de la altura	El valor del volumen del cilindro	Volumen = (2 * pi * Radio) * altura

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Radio de la base	radio	O Entero ® Real O Lógico O Cadena
Altura del Cilindro	altura	O Entero ® Real O Lógico O Cadena
Área del Cilindro	area	O Entero ® Real O Lógico O Cadena
Volumen del Cilindro	volumen	O Entero ® Real O Lógico O Cadena

Código 3-13. area_vol_cil.py

```
# Propósito: Determinar el área y el volumen de un cilindro
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Módulos Importados
from math import pi

# Entrada de datos
radio = float(input('Radio de la base [cm]: '))
altura = float(input('Altura del cilindro [cm]: '))

# Procesamiento de datos
area = (2 * pi * radio) * altura
volumen= pi * radio * radio * altura

# Salida de datos
print('Área del Cilindro= %.2f' % area, ' [cm2]')
print('Volumen del Cilindro= %.2f' % volumen, ' [cm3]')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecución:

Radio de la base [cm]: 2
 Altura del cilindro [cm]: 10

Área del Cilindro= 125.66 [cm²]

Volumen del Cilindro= 125.66 [cm³]

Pulse una tecla para finalizar...

Ejercicio 3-14: Dadas las longitudes de los catetos de un triángulo, desarrolle un programa que determine la hipotenusa y para el ángulo que forma uno de los catetos con la hipotenusa, el valor del seno, coseno y tangente de dicho ángulo.

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logro?
Longitud de un cateto. Longitud del otro cateto.	Longitud de la hipotenusa. Seno del ángulo de la hipotenusa con uno de los catetos. Coseno del ángulo de la hipotenusa con el mismo cateto anterior Tangente del ángulo de los catetos anteriores	hip = (co ² + ca ²) ^{0.5} seno = co / hip cos = ca / hip tan = co / ca

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Longitud del cateto adyacente	ca	O Entero ® Real O Lógico O Cadena
Longitud del cateto opuesto	co	O Entero ® Real O Lógico O Cadena
Longitud de la hipotenusa	h	O Entero ® Real O Lógico O Cadena
Seno del ángulo	seno	O Entero ® Real O Lógico O Cadena
Coseno del ángulo	Cos	O Entero ® Real O Lógico O Cadena
Tangente del ángulo	tan	O Entero ® Real O Lógico O Cadena

Código 3-14. func_trigon.py

```
# Propósito: Determinar las funciones trigonométricas
# Autor: Alejandro Bolívar
# Fecha: 30/01/2020

# Entrada de datos

ca = float(input('Longitud del cateto adyacente:'))
co = float(input('Longitud del cateto opuesto:'))

# Procesamiento de datos
hip = (co**2 + ca**2) ** 0.5
seno = co / hip
coseno = ca / hip
tangente = co / ca
```

```

#Salida de datos
print("Longitud de la Hipotenusa=", hip)
print("Seno del ángulo= ", seno)
print("Coseno del ángulo= ", coseno)
print("Tangente del ángulo= ", tangente)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Ejecución:

Longitud del cateto adyacente:3
 Longitud del cateto opuesto:4
 Longitud de la Hipotenusa= 5.0
 Seno del ángulo= 0.8
 Coseno del ángulo= 0.6
 Tangente del ángulo= 1.333333333333333
 Pulse una tecla para finalizar...

Ejercicio 3-15: Debido a que aún en Norteamérica no se utilizan el sistema métrico decimal en la medición de longitudes, desarrolle un programa que, dada la longitud en pies de una pieza cualquiera, determine e imprima su equivalente en a) yardas; b) pulgadas; c) centímetros, y d) metros, sabiendo que 1 yarda=3 pies, 1 pie=12 pulgadas, 1 pulgada=2.54 centímetros, 1 metro=100 centímetros.

¿Qué información está presente?	¿Qué resultados exige?	¿Cómo lo logro?
Longitud en pies	Longitud equivalente en yardas Longitud equivalente en pulgadas Longitud equivalente en centímetros Longitud equivalente en metros	yarda = longitud / 3 pulgada = longitud * 12 cm = pulgada * 2.54 mts = cm / 100

Tabla de Variables:

Descripción del dato	Nombre de la Variable	Tipo de Dato
Longitud en pies	longitud	<input type="radio"/> Entero <input checked="" type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena
Longitud en Yardas	yarda	<input type="radio"/> Entero <input checked="" type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena
Longitud en Pulgadas	pulgada	<input type="radio"/> Entero <input checked="" type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena
Longitud en Centímetros	cm	<input type="radio"/> Entero <input checked="" type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena
Longitud en Metros	mts	<input type="radio"/> Entero <input checked="" type="radio"/> Real <input type="radio"/> Lógico <input type="radio"/> Cadena

Código 3-15. conv_pie.py 

```
# Propósito: Convertir de pie a otras unidades de longitud
# Autor:      Alejandro Bolívar
# Fecha:      30/01/2020

# Entrada de datos
longitud= float(input('Longitud de la pieza [pie]:'))

# Procesamiento de datos
yarda = longitud / 3
pulgada = longitud * 12
cm = pulgada * 2.54
mts = cm / 100

#Salida de datos
print("Longitud de la pieza en Metro:", mts)
print("Longitud de la pieza en Pie:", longitud)
print("Longitud de la pieza en Yardas:", yarda)
print("Longitud de la pieza en Pulgada:", pulgada)
print("Longitud de la pieza en Centimetro:", cm)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Unidad 4

Estructuras de Control

Estructuras Selectivas

Las estructuras selectivas permiten seleccionar o decidir la ejecución de un conjunto de instrucciones, según un valor o expresión lógica.

Unidad IV. Estructuras de Control

Objetivos de la unidad:

- En esta unidad aprenderá a utilizar las estructuras condicionales, con el fin de dirigir el flujo del programa de acuerdo a la necesidad de ejecutar las instrucciones necesarias para resolver un problema.
- Se presenta una recopilación y adaptación de ejercicios relacionados con el desarrollo de series numéricas.

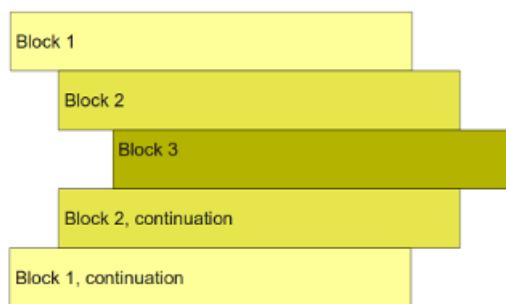
4.1 Control de flujo de datos

Todos los lenguajes de programación disponen de instrucciones de control de flujo. Estas instrucciones permiten al programador decidir el orden de ejecución del código con el fin de permitir ejecutar diferentes órdenes en función de una serie de condiciones sobre el estado. Python ofrece los dos tipos básicos de sentencias de control de flujo: las sentencias condicionales y los ciclos (o repeticiones).

4.2 Sangría

La **sangría** es la sangría que se asigna a las instrucciones que se ejecutarán dentro de una estructura de control.

No todos los lenguajes de programación, necesitan de una sangría, aunque sí, se estila implementarla, a fin de otorgar mayor legibilidad al código fuente. Pero en el caso de Python, la sangría es obligatoria, ya que, de ella, dependerá su estructura.



PEP 8: sangría

Una sangría de **4 (cuatro) espacios en blanco**, indicará que las instrucciones sangradas, forman parte de una misma estructura de control.

Una estructura de control, entonces, se define de la siguiente forma:

inicio de la estructura de control:
expresiones

4.2.1 Una excepción a la regla de sangría

Cada vez que una sentencia acaba con dos puntos (:), Python espera que la sentencia o sentencias que le siguen aparezcan con una mayor sangría. Es la forma de marcar el inicio y el fin de una serie de sentencias que <<dependen>> de otra. Hay una excepción: si solo hay una sentencia que <<depende>> de otra, puedes escribir ambas en la misma línea como se aprecia en el código *parimpar1.py*,

```
a = int(input('Dame un entero positivo: '))

if a % 2 == 0:
    print ('El número es par ')
else:
    print ('El número es impar ')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

y este otro, *parimpar2.py*,

```
a = int(input('Dame un entero positivo: '))

if a % 2 == 0: print ('El número es par ')
else: print ('El número es impar ')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

son equivalentes.



Atención

Todas las instrucciones que describen el cuerpo de la estructura deben tener una sangría mayor que el encabezado de la estructura. Esta sangría puede ingresarse mediante 4 espacios preferiblemente o un tabulador, pero es importante que sea la misma para todas las instrucciones de la estructura.

4.3 Estructuras Selectivas

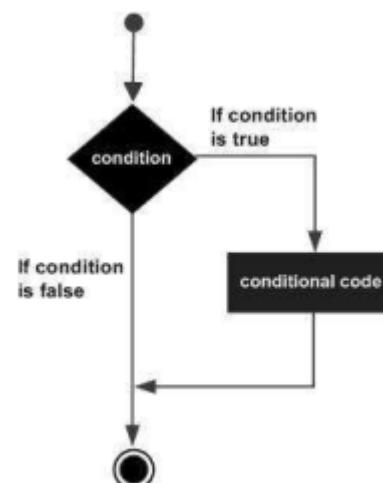
Las estructuras de selección o decisión permiten ejecutar un bloque de código entre varios disponibles, según el resultado de la evaluación de una expresión lógica situada en la cabecera de la estructura.

4.3.1 Estructura Condicional Simple

```
if expresión_lógica_a_evaluar:  
    ejecutar_si_cierto
```

Esta forma del condicional *if* se interpreta como *Si if condition : (expresión_lógica) es verdadera, el flujo del programa continúa ejecutando las instrucciones que estén dentro del bloque if, las cuales se especifican y se reconocen por estar debidamente sangradas respecto al inicio de la palabra if*. Recuerde que la sangría se establece dejando espacios en blanco, pasando la instrucción (palabra) *if*. *Si la (expresión_lógica) es falsa, el flujo del programa se continúa ignorando las instrucciones del bloque if*.

Por ejemplo, el siguiente código *mayoredad.py* verifica si una persona es mayor de edad.



```
edad = int(input('Ingrese su edad: '))  
if edad >= 18:  
    print ('Es mayor de edad')  
# Fin del programa  
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejecute este programa, probando varias veces con valores diferentes.

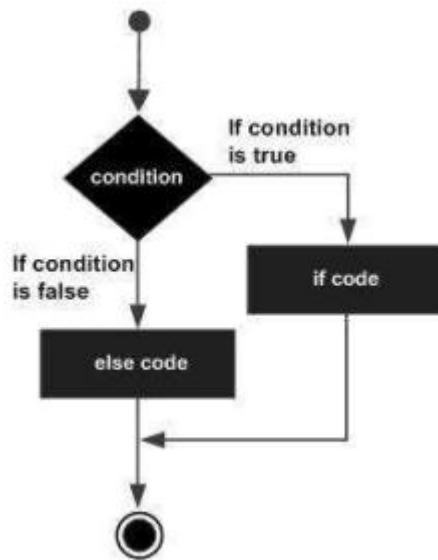
4.3.2 Estructura Condicional Doble

La forma del *if* estudiada en la sección anterior presenta la posibilidad de hacer un desvío en el flujo de ejecución de un programa cuando la condición o expresión lógica que determina al *if* es verdadera (True). Si la expresión lógica no se cumple, es falsa (False), el flujo del programa continúa su secuencia como si la instrucción *if* no estuviese en el programa. La construcción *if-else* ofrece la alternativa de ejecutar otras instrucciones en caso que la expresión lógica que determina la instrucción *if* sea falsa. Su forma general es como sigue:

```
if expresion_a_evaluar:  
    ejecutar_si_cierto  
else:  
    ejecutar_en_caso_contrario
```

En otras palabras, esta forma del condicional *if* la interpretaremos como Si la (expresión lógica) es verdadera, el flujo del programa se continúa ejecutando las instrucciones que estén dentro del bloque *if* (las cuales se especifican y se reconocen por estar debidamente sangradas respecto al inicio de la palabra *if*). Si la (expresión lógica) es falsa, el flujo del programa continúa ejecutando las instrucciones que estén dentro del bloque *else* (las cuales se especifican y se reconocen por estar debidamente sangradas respecto al inicio de la palabra *else*). En este caso, se garantiza que una de las dos bifurcaciones del flujo del programa se ejecuta. Debe notarse que las instrucciones *if:* y *else:* NO están sangradas una de la otra. Es decir, ambas instrucciones comienzan al mismo nivel en el programa (solo se sangra el bloque de instrucciones que debe ejecutarse en caso que alguna de las condiciones se cumpla). Cabe mencionar que mientras la instrucción *if* puede tener significado por sí sola, la instrucción *else* NO puede aparecer de forma aislada. Siempre debe ir precedida por un *if*. Por ejemplo, el siguiente código (parimpar3.py) realiza acciones distintas dependiendo de si el número de entrada es par o impar:

if condition :
 if-block
else:
 else-block



```

edad = int(input('Ingrese su edad: '))
if edad >= 18:
    print ('Es mayor de edad')
else:
    print ('Es menor de edad')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

La última sentencia no está sangrada, por lo que no es parte de la estructura condicional, y será ejecutada siempre.

4.3.3 Estructura Condicional Múltiple

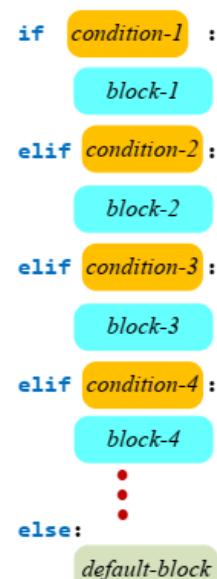
Como el lector puede haber anticipado, se puede tener situaciones en las que se tenga que seleccionar entre más de dos condiciones. Para estos casos Python ofrece la estructura condicionada *if–elif–else*, cuya forma general es como sigue:

```

if expresion_lógica_1:
    ejecutar_si_verdadero_1
elif expresion_lógica_2:
    ejecutar_si_verdadero_2
elif expresion_lógica_3:
    ejecutar_si_verdadero_3
else:
    ejecutar_si_ninguna

```

En palabras, esta forma del condicional *if* la interpretaremos como *Si la (expresión_lógica_1) es verdadera, el flujo del programa se continúa ejecutando las instrucciones que estén dentro del bloque if (las cuales se especifican y se reconocen por estar debidamente sangradas respecto al inicio de la palabra if). Si la (expresión_lógica_1) es falsa, entonces el flujo del programa pasa a verificar la primera instrucción elif cuyo bloque de instrucciones se ejecutan Si la (expresión_lógica_2) es verdadera. En caso contrario, el flujo del programa continúa con la verificación de la siguiente instrucción elif cuyo bloque de instrucciones se ejecutan Si la (expresión_lógica_3) es verdadera. En caso que ninguna de las condiciones previas a la instrucción else: sea ejecutada, el flujo del programa hace que se ejecuten las instrucciones de ese bloque. A continuación, se presentan ejemplos de la aplicación de una estructura condicional.*



Ejercicio 4-1: Dado un valor real x, determine si es positivo, negativo o cero.

Código 4-1. posneg.py 

```
# Lectura de datos
x = float(input('Ingrese un valor x: '))

# Procesamiento de datos
if x < 0.0:
    msg = 'negativo'
elif x > 0.0:
    msg = 'positivo'
else:
    msg = 'cero'

# Salida de datos
print ('x es ', msg)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Ejercicio 4-2: La tasa de impuesto a pagar por una persona según su sueldo es la siguiente.

Tabla 4-1. Tasa de impuesto en función del sueldo recibido

Sueldo	Tasa de impuesto
menos de 1000	0%
$1000 \leq \text{sueldo} < 2000$	5%
$2000 \leq \text{sueldo} < 4000$	10%
4000 o más	12%

Entonces, el programa que calcula el impuesto a pagar es el siguiente:

Código 4-2. impuesto.py 

```
# Lectura de datos
sueldo = float(input('Ingrese su sueldo: '))

# Procesamiento de datos
if sueldo < 1000:
    tasa = 0.00
elif sueldo < 2000:
    tasa = 0.05
elif sueldo < 4000:
    tasa = 0.10
else:
    tasa = 0.12

# Salida de datos
```

```

print ('Usted debe pagar ', tasa * sueldo, ' de impuesto')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Siempre sólo una de las alternativas será ejecutada. Tan pronto alguna de las condiciones es verdadera, el resto de ellas no se evalúan.

4.4 Ejercicios Resueltos con estructuras condicionales

- 1) Un sistema de ecuaciones lineales $\begin{cases} a * x + b * y = c \\ d * x + e * y = f \end{cases}$

Se puede resolver con las siguientes fórmulas: $x = (c * e - b * f) / (a * e - b * d)$; $y = (a * f - c * d) / (a * e - b * d)$. Desarrolle un programa (*sistema.py*) que lea los coeficientes de ambas ecuaciones (a, b, c, d, e y f) y determine los valores de x e y. Nota: evite la división por cero e imprima el mensaje adecuado.

Código 4-3. sistema.py

```

# Lectura de datos
print('Introduzca los coeficientes: ')
a = float(input('a:'))
b = float(input('b:'))
c = float(input('c:'))
d = float(input('d:'))
e = float(input('e:'))
f = float(input('f:'))

# Procesamiento de datos
deno = a * e - b * d
if deno == 0:
    # Si el denominador es cero, no existe solución
    print('No tiene solución el sistema de ecuaciones')
else:
    x = (c * e - b * f) / deno
    y = (a * f - c * d) / deno

# Salida de datos
print('La solución del sistema de ecuaciones planteado es')
print('X=', x)
print('Y=', y)
input('Pulse una tecla para finalizar... ') # Pausa

```

- 2) Dado un valor numérico entero, desarrolle un programa (*paroimpar.py*) que determine si es par o impar.

Código 4-4. paroimpar.py

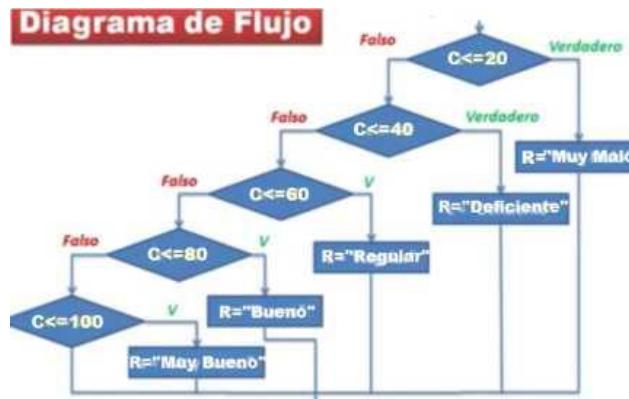
```
# Entrada del dato
n = int(input('De un valor entero: '))
# Procesamiento y salida de datos
if n == 0:
    print(n, ' No es par ni impar')
elif n % 2 != 0:
    print(n, ' Es Impar')
else:
    print(n, ' Es Par')

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

- 3) Dada la el valor numérico con que una persona indica el agrado por un determinado producto, un valor entre 0 y 100, elabore un programa (calificación.py) que determine la calificación del producto en base a la siguiente tabla y empleando *if* anidados.

Rango	Calificación
0-20	Muy Malo
21-40	Deficiente
41-60	Regular
61-80	Bueno
81-100	Muy Bueno

Para plantear la solución empezamos con una decisión considerando el límite superior del rango, si es menor o igual a ese valor está en esa calificación sino en alguna de las otras, así vamos descartando una a una las posibilidades, quedando en diagrama de flujo algo como lo siguiente:



Código 4-5. calificación.py

```
# Lectura de datos
agrado = float(input('Ingrese el valor de agrado: '))

# Procesamiento de datos
if agrado <= 20:
    msg = 'Muy Malo'
elif agrado <= 40:
    msg = 'Deficiente'
elif agrado <= 60:
    msg = 'Regular'
elif agrado <= 80:
    msg = 'Bueno'

else: # Observe que no es necesario hacer la última pregunta
    msg = 'Muy Bueno'

# Salida de datos
print ('La Calificación es ' + msg)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa
```

Unidad 4

Estructuras de Control

Estructuras Repetitivas

Las estructuras repetitivas, permiten ejecutar un conjunto de instrucciones un número determinado de veces o mientras una condición se cumpla. Las estructuras repetitivas son muy utilizadas para la resolución de problemas, prácticamente todo programa utiliza estas estructuras.

Objetivos de la Unidad:

- En esta unidad se aprenderá a utilizar las estructuras repetitivas for y while, así como también las herramientas de programación para resolver la diversidad de problemas que se pueden presentar en programación.
- Realizar el desglose y composición de números enteros

4.5 Estructuras Repetitivas

Estas estructuras, también denominadas ciclos o bucles, ejecutan un bloque de código de forma repetitiva mientras se cumpla una condición asociada a la estructura. A cada una de las ocasiones en que se ejecuta el código contenido en estas estructuras se le denomina **iteración**.

Instrucción	Python
Ciclo Repetitivo con condición al inicio	<code>while condición: Instrucción(es)</code>
Ciclo de Repetición Automática	<code>for elemento in secuencia: Instrucción(es)</code>

4.5.1 Ciclo while

La estructura repetitiva **mientras** (en inglés **while**) es aquella en que el cuerpo del bucle o ciclo (en inglés **block**) se repite mientras se cumple una determinada condición. Cuando se ejecuta la instrucción **mientras**, la primera cosa que sucede es que se evalúa la condición (una expresión *booleana*). Si se evalúa *falsa*, no se toma ninguna acción y el programa prosigue en la siguiente instrucción del bucle. Si la expresión *booleana* es *verdadera*, entonces se ejecuta el cuerpo del bucle, después de lo cual se evalúa de nuevo la expresión booleana. Este proceso se repite una y otra vez **mientras** la expresión *booleana* (condición) sea verdadera [1].

while *condition* :
block

```
while <Condicion y/o variable booleana >:  
    # Instrucciones sangradas que serán ejecutadas
```

- La palabra reservada ***while*** encabeza la estructura repetitiva.
- La *condición* determina sí el cuerpo del ciclo se ejecuta (o se continua) ejecutando. Seguidamente se escriben dos puntos después de la condición.
- El *bloque* o cuerpo de la estructura repetitiva son una o varias instrucciones a ejecutar mientras la condición sea verdadera. Todas las instrucciones que conforman el bloque, se deben colocar con una sangría a partir de la posición que ocupa el inicio de la palabra ***while***.

Es importante notar que se puede escribir un ***while*** dentro de otro ***while*** (ciclo anidado) respetando el sangrado y también es posible escribir dentro del ciclo las entidades condicionales *if*, *elif*, *else*. Al momento de usar un ciclo *while* en un programa se debe ser cuidadoso, puesto que si la expresión booleana nunca llega a ser falsa el programa puede quedar en un *ciclo infinito* y no terminar el programa.

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 (*True*). Hay que recordar que en Python la secuencia de instrucciones dentro de la estructura algorítmica debe estar sangrada.

A continuación, se presenta un ejercicio que muestra la tabla de multiplicar (Código 4-6) del número introducido por el usuario. Esto es un esquema típico de recorrido:

Código 4-6. Tabla_de_multiplicar.py

```
# Tabla de multiplicar (del 1 al 10) del número introducido
n = int(input('Tabla: '))
k = 1
while k <= 10:
    print(n, ' * ', k, ' = ', n * k)
    k = k + 1
print('Tabla de multiplicar del', n)
```

El programa pide al usuario un número *n*, se inicializa la variable *k* en 1 y luego se ejecuta la iteración *while* 10 veces, mientras la condición *k <= 10* es *True*. En la décima

iteración la variable *k* toma el valor 11 y entonces la expresión booleana *k* ≤ 10 es *False* y sale del *while*, pasando el programa a la instrucción siguiente *print()*.

```
Tabla: 7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
Tabla de multiplicar del 7
```

El siguiente código (Código 4-7) muestra los primeros *n* números naturales en forma decreciente:

Código 4-7. decreciente.py

```
# Mostrar los N primeros números naturales en forma descendente
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 (Código 4-8) que calcula la suma de los *n* primeros números naturales.

Código 4-8. suma_n_primeros.py

```
# Cálculo 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 (Código 4-9) sigue un esquema típico de búsqueda. Se requiere saber si un número natural es primo o no. Un número primo solo es divisible por 1 y por sí

mismo, por lo que se prueba con los números desde 2 hasta el anterior a él mismo. Si es divisible por alguno entonces se tiene un divisor y se detiene la búsqueda, el número no será primo. Si llega al final y no se ha encontrado divisores, entonces es primo.

Código 4-9. `número_primo.py`.

```
# Programa que determina si un número es primo o no
n = int(input('Entra un número natural: '))
d = 2
un_divisor = True # Si hay un divisor el número NO es primo
while d < n and un_divisor:
    if n % d == 0:
        un_divisor = False # Se encontró un divisor
    d = d + 1
print('¿Es primo? ', 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 se desea saber si es primo o no. La condición $d < n$ del ciclo *while* puede cambiarse para detener la búsqueda antes y obtener el mismo resultado.

El algoritmo de Euclides para calcular el máximo común divisor de dos números n y m , según los siguientes pasos.

1. Teniendo n y m , se obtiene r , el resto de la división entera de m / n .
2. Si r es cero, n es el MCD de los valores iniciales.
3. Se reemplaza $m = n$, $n = r$, y se vuelve al primer paso.

El algoritmo de Euclides (Código 4-10) para hallar el MCD de dos números naturales se puede programar en Python como:

Código 4-10. `Algoritmo_de_Euclides.py`.

```
# Algoritmo de Euclides
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 se utilizó 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.

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

Durante el proceso de entrada de datos en programas informáticos se puede validar los valores de una variable. En este caso se debe cerciorar que el número introducido por el usuario sea positivo. El siguiente código, 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!!: '))

```

4.5.2 Ciclo *for*

Un *for* es una entidad que recorre de manera determinada los elementos de una lista y ejecuta las instrucciones que tenga sangradas [5]. Posee la siguiente sintaxis:

<code>for <variable local > in <lista >: # Grupo de instrucciones a ejecutar # Pueden usar o no a la variable local</code>
--

En muchos cálculos se incluye el proceso de una cadena carácter a carácter. A menudo empiezan por el principio, seleccionan cada carácter por turno, hacen algo con él y

siguen hasta el final. Este patrón de proceso se llama recorrido. Una forma de codificar un recorrido es una sentencia *while*:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print (letra)
    indice = indice + 1
```

Este ciclo recorre la cadena y muestra cada letra en una línea distinta. La condición del ciclo es *indice < len(fruta)*, de modo que cuando *indice* es igual a la longitud de la cadena, la condición es falsa y no se ejecuta el cuerpo del ciclo. El último carácter al que se accede es el que tiene el índice *len(fruta)-1*, que es el último carácter de la cadena.

Es tan habitual usar un índice para recorrer un conjunto de valores, que Python facilita una sintaxis alternativa más simple: el ciclo *for*:

```
for car in fruta:
    print (car)
```

Cada vez que se recorre el ciclo, se le asigna a la variable *car* el siguiente carácter de la cadena. El ciclo continúa hasta que no quedan caracteres.

4.5.3 La Función `range()`

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función incorporada `range()` la cual genera una lista que contiene progresiones aritméticas:

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

El valor final dado **nunca** es parte de la lista; `range(10)` genera una lista de 10 valores, los índices correspondientes para los elementos de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se le llama 'paso'), en la Tabla 4-2 se muestra diferentes usos de la función `range`:

Tabla 4-2. Ejemplos de Usos de la función `range`

Uso	Descripción	Ejemplo
<code>range(vf)</code>	Retorna la lista desde 0 hasta vf-1	<code>>>> range(3)</code> <code>[0,1,2]</code>
<code>range(vi, vf)</code>	Retorna la lista desde vi hasta vf-1	<code>>>> range(1,4)</code>

`range(vi, vf, inc)`

Retorna la lista desde vi hasta vf-1 pero sumando
inc valores

[1,2,3]
>>> `range(1,5,2)`
[1,3]

```
for i in range(1, 6):  
    print (i)
```

Al ejecutar el programa, se visualiza lo siguiente por pantalla:

Ejecución:

1
2
3
4
5

O también,

```
for i in range(4):  
    print(i,end=" ")
```

Ejecución:

0 1 2 3



Obi Wan

Puede resultar sorprendente que `range(a, b)` incluya todos los números enteros comprendidos entre a y b , pero sin incluir b . En realidad, la forma “natural” o más frecuente de usar `range` es con un sólo parámetro: `range(n)` que devuelve una lista con los n primeros números enteros incluyendo al cero (hay razones para que esto sea lo conveniente, ya llegaremos). Como incluye al cero y hay n números, no puede incluir al propio número n . Al extenderse el uso de `range` a dos argumentos, se ha mantenido la “compatibilidad” eliminando el último elemento. Una primera ventaja es que resulta fácil calcular cuántas iteraciones realizará un ciclo `range(a, b)`: exactamente $b - a$. (Si el valor b estuviera incluido, el número de elementos sería $b - a + 1$.)

Hay que ir con cuidado, pues es fácil equivocarse “*por uno*”. De hecho, equivocarse “*por uno*” es tan frecuente al programar (y no sólo con `range`) que hay una expresión para este tipo de error: un error Obi Wan (Kenobi), que es más o menos como suena en inglés “*off by one*” (pasarse o quedarse corto por uno).



Índice de ciclo for-in: ¡prohibido asignar!

Hemos aprendido que el ciclo **for-in** utiliza una variable índice a la que se van asignando los diferentes valores del rango. En muchos ejemplos se utiliza la variable *i*, pero sólo porque también en matemáticas las sumatorias y productorias suelen utilizar la letra *i* para indicar el nombre de su variable índice. Puedes usar cualquier nombre de variable válido.

Pero que el índice sea una variable cualquiera no te da libertad absoluta para hacer con ella lo que quieras. En un ciclo, las variables de índice sólo deben usarse para consultar su valor, nunca para asignarles uno nuevo. Por ejemplo, este fragmento de programa es incorrecto:

```
for i in range(0, 5):
    i += 2 # Error
```

Y ahora qué sabes que los ciclos pueden anidarse, también has de tener mucho cuidado con sus índices. Un error frecuente entre primerizos de la programación es utilizar el mismo índice para dos ciclos anidados. Por ejemplo, estos ciclos anidados están mal:

```
for i in range(0, 5):
    for i in range(0, 3):
        print (i) # Error
```

En el fondo, este problema es una variante del anterior, pues de algún modo se está asignando nuevos valores a la variable *i* en el ciclo interior, pero *i* es la variable del ciclo exterior y asignarle cualquier valor está prohibido.

Recuerda: *nunca debes asignar un valor a un índice de ciclo ni usar la misma variable índice en ciclos anidados.*

4.5.4 Ejercicios de Estructuras Repetitivas

Ejercicio 4-3: Elabore un programa (promvalores.py**) que lea n valores de tipo entero y devuelva el promedio de los n valores.**

Código 4-11. **promvalores.py**

```
# Determinar el promedio de n números de un conjunto

# Lectura de datos
n = int(input('Indique la cantidad de elementos: '))

# Inicialización de variables
suma = 0

# Procesamiento de datos
for k in range(n):
```

```

# Lectura de datos
num = int(input('Indique un valor: '))
suma = suma + num
prom = suma / n

# Salida de datos
print ('El promedio es: ', prom)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Ejecución:

===== RESTART: G:/Ejercicios Python/promvalores.py =====

Indique la cantidad de elementos: 5

Indique un valor: 1

Indique un valor: 2

Indique un valor: 3

Indique un valor: 4

Indique un valor: 5

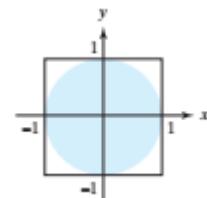
El promedio es: 3.0

Pulse una tecla para finalizar...

Ejercicio 4-4: Una simulación de Monte Carlo usa números aleatorios y probabilidad para resolver problemas. Tiene una amplia gama de aplicaciones en matemática computacional, física, química y finanzas. Veamos un ejemplo de usar una simulación Monte Carlo para estimar el valor de π .

Primero, dibuja un círculo con su cuadrado delimitador.

Suponga que el radio del círculo es 1. Entonces, el área del círculo es π y el área cuadrada es 4. Genere aleatoriamente un punto en el cuadrado. La probabilidad de que el punto esté en el círculo es $\text{circleArea} / \text{squareArea} = \pi / 4$.



Escribe un programa (MonteCarloSimulation.py) que genere al azar 1000000 puntos que caen en el cuadrado y deja que `numberOfHits` denote la cantidad de puntos que caen en el círculo. Entonces, `numberOfHits` es aproximadamente $1000000 * (\pi / 4)$. El valor de π puede aproximarse a $4 * \text{numberOfHits} / 1000000$. El programa completo se muestra en el Código 4-12.

Código 4-12. MonteCarloSimulation.py

```
import random

NUMBER_OF_TRIALS = 1000000 # Constante
numberOfHits = 0

for i in range(NUMBER_OF_TRIALS):
    x = random.random() * 2 - 1
    y = random.random() * 2 - 1

    if x * x + y * y <= 1:
        numberOfHits += 1

pi = 4 * numberOfHits / NUMBER_OF_TRIALS

print("PI es", pi)
```

El resultado es PI es 3.14124.

Ejercicio 4-5: Mostrar un menú (menú.py) con tres opciones: 1- comenzar programa, 2- imprimir listado, 3-finalizar programa. A continuación, el usuario debe poder seleccionar una opción (1, 2 ó 3). Si elige una opción incorrecta, informarle del error. El menú se debe volver a mostrar luego de ejecutada cada opción, permitiendo volver a elegir. Si elige las opciones 1 ó 2 se imprimirá un texto. Si elige la opción 3, se interrumpirá la impresión del menú y el programa finalizará.

Código 4-13. menú.py.

```
while True:
    print("Opción 1 - comenzar programa")
    print("Opción 2 - imprimir listado")
    print("Opción 3 - finalizar programa")
    opcion = int(input("Opción elegida: "))
    if opcion == 1:
        print("¡Comenzamos!")
    elif opcion == 2:
        print("Listado:")
        print("Nadia, Esteban, Mariela, Fernanda")
    elif opcion == 3:
        print("Hasta la próxima")
        break
    else:
        print("Opción incorrecta. Debe ingresar 1, 2 o 3")
```

4.6 Herramientas de programación

Como hemos visto hasta ahora, los programas son una combinación de asignaciones, condicionales y ciclos, organizados de tal manera que describan el algoritmo a ejecutar.

Existen algunas tareas comunes y que casi siempre se resuelven de la misma manera. Por lo tanto, es conveniente conocerlas.

En programación, se llama patrón a una solución que es aplicable a un problema que ocurre a menudo. A continuación, veremos algunos patrones comunes que ocurren en programación.

4.6.1 Sumar y multiplicar valores

La suma y la multiplicación son operaciones binarias: operan sobre dos valores.

Para sumar y multiplicar más valores, generalmente dentro de un ciclo que los vaya generando, hay que usar una variable para ir guardando el resultado parcial de la operación. Esta variable se llama acumulador.

En el caso de la suma, el acumulador debe partir con el valor cero. Para la multiplicación, con el valor uno. En general, el acumulador debe ser inicializado con el elemento neutro de la operación que será aplicada.

Por ejemplo, el siguiente programa entrega el producto de los mil primeros números naturales:

```
producto = 1
for i in range(1, 1001):
    producto = producto * i
print (producto)
```

El siguiente programa entrega la suma de los cubos de los números naturales cuyo cuadrado es menor que 1000:

```
i = 1
suma = 0
while i ** 2 < 1000:
    valor = i ** 3
    i = i + 1
    suma = suma + valor
print (suma)
```

En todos los casos, el patrón a seguir es algo como esto:

```
acumulador = valor_inicial
ciclo:
    valor = ...
    ...
    acumulador = acumulador operación valor
```

El cómo adaptar esta plantilla a cada situación de modo que entregue el resultado correcto es responsabilidad del programador.

4.6.2 Contar sucesos

Para contar cuántas veces ocurre algo, hay que usar un acumulador, al que se le suele llamar contador.

Tal como en el caso de la suma, debe ser inicializado en cero, y cada vez que aparezca lo que queremos contar, hay que incrementarlo en uno.

Por ejemplo, el siguiente programa cuenta cuántos de los números naturales menores que mil tienen un cubo terminado en siete:

```
c = 0
for i in range(1000):
    ultimo_dígito = (i ** 3) % 10
    if ultimo_dígito == 7:
        c = c + 1
print (c)
```

4.6.3 Encontrar el mínimo y el máximo

Para encontrar el máximo de una secuencia de valores, hay que usar una variable auxiliar para recordar cuál es el mayor valor visto hasta el momento. En cada iteración, hay que examinar cuál es el valor actual, y si es mayor que el máximo, actualizar la variable auxiliar.

La variable auxiliar debe ser inicializado con un valor que sea menor a todos los valores que vayan a ser examinados.

Por ejemplo, el siguiente programa pide al usuario que ingrese diez números enteros positivos, e indica cuál es el mayor de ellos:

```
print ('Ingrese diez números positivos')
```

```
mayor = -1
for i in range(10):
    n = int(input())
    if n > mayor:
        mayor = n

print ('El mayor es ', mayor)
```

Otra manera de hacerlo es reemplazando esta parte:

```
if n > mayor:
    mayor = n
```

por ésta:

```
mayor = max(mayor, n)
```

En este caso, como todos los números ingresados son positivos, se inicializa la variable auxiliar en -1, que es menor que todos los valores posibles, por lo que el que sea el mayor eventualmente lo reemplazará.

¿Qué hacer cuando no exista un valor inicial que sea menor a todas las entradas posibles? Una solución es poner un número «pequeño negativo», de tal manera que el usuario no ingrese uno menor que él. Esta no es la mejor solución, ya que no cubre todos los casos posibles:

```
mayor = -999999999
for i in range(10):
    n = int(input())
    mayor = max(mayor, n)
```

Una opción más robusta es usar el primero de los valores por examinar:

```
mayor = int(input()) # preguntar el primer valor

for i in range(9): # preguntar los nueve siguientes
    n = int(input())
    mayor = max(mayor, n)
```

La otra buena solución es usar explícitamente el valor $-\infty$, que en Python puede representarse usando el tipo *float* de la siguiente manera:

```
mayor = -float('inf') # Así se asigna "infinito" en Python
for i in range(10):
    n = int(input())
    mayor = max(mayor, n)
```

Si se sabe de antemano que todos los números por revisar son positivos, podemos simplemente inicializar la variable mayor en 1.

Por supuesto, para obtener el menor valor se hace de la misma manera, pero inicializando el acumulador con un número muy grande, y actualizándolo al encontrar un valor menor.

4.6.4 Generar pares

Para generar pares de elementos en un programa, es necesario usar dos ciclos anidados (es decir, uno dentro del otro).

Ambos ciclos, el exterior y el interior, van asignando valores a sus variables de control, y ambas son accesibles desde dentro del doble ciclo.

Por ejemplo, todas las casillas de un tablero de ajedrez pueden ser identificadas mediante un par (fila, columna). Para recorrer todas las casillas del tablero, se puede hacer de la siguiente manera:

```
for i in range(1, 9):
    for j in range(1, 9):
        print ('Casilla', i, j)
```

Cuando los pares son desordenados (es decir, el par **(a,b)** es el mismo que el par **(b,a)**), el ciclo interior no debe partir desde cero, sino desde el valor que tiene la variable de control del ciclo interior.

Por ejemplo, el siguiente programa muestra todas las piezas de un juego de dominó:

```
for i in range(7):
    for j in range(i, 7):
        print (i, j)
```

Además, otros tipos de restricciones pueden ser necesarias. Por ejemplo, en un campeonato de fútbol, todos los equipos deben jugar entre ellos dos veces, una como local y una como visita. Por supuesto, no pueden jugar consigo mismos, por lo que es necesario

excluir los pares compuestos por dos valores iguales. El siguiente programa muestra todos los partidos que se deben jugar en un campeonato con 6 equipos, suponiendo que los equipos están numerados del 0 al 5:

```
for i in range(6):
    for j in range(6):
        if i != j:
            print (i, j)
```

Otra manera de escribir el mismo código es:

```
for i in range(6):
    for j in range(6):
        if i == j:
            continue
        print (i, j)
```

4.7 Secuencias y Series

Para los estudiantes de programación, es interesante la aplicación de las técnicas de resolución de ejercicios relacionados al cálculo de los términos de una serie o secuencia, porque en ellas se utilizan todas las técnicas elementales de programación, tales como acumuladores, ciclos de diferentes tipos, banderas y por supuesto expresiones de diferentes tipos.

SUCESIÓN Y SECUENCIA: Una sucesión matemática es una función definida sobre los enteros naturales. Una secuencia es una concatenación de símbolos obtenidos a partir de una sucesión. Son semejantes a las sucesiones y se pueden derivar fácilmente de éstas. Por ejemplo, la definición matemática de la secuencia de Fibonacci es:

$$fib(n) \begin{cases} fib(1) = 0 \text{ Para } n = 1 \\ fib(2) = 1 \text{ Para } n = 2 \\ fib(n) = fib(n - 2) + fib(n - 1) \forall n > 2 \end{cases}$$

Los primeros 10 términos de esta secuencia son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . . Así, si se toma como base la secuencia de Fibonacci, es sencillo definir una nueva secuencia para el alfabeto $A = \{0,1\}$ según el siguiente método:

$$S(n) \begin{cases} 1 & \text{Si } n \text{ está en la sucesión de Fibonacci} \\ 0 & \text{En otro caso} \end{cases}$$

que obtendría como resultado un 1 para el numero natural 1, ya que éste está en la secuencia de Fibonacci; 1 para los números naturales 2 y 3; pero 0 para el numero natural 4, ya que él no pertenece a la secuencia de Fibonacci. Obteniéndose, entonces, la siguiente secuencia de dígitos binarios: 1 1 1 1 0 1 0 0 1 0 0 0 0 1 0 ...

Observe la relación entre la secuencia de los dígitos binarios generada, la secuencia de Fibonacci: y los números naturales:

N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S(n)	1	1	1	1	0	1	0	0	1	0	0	0	0	1	0

- Los ejercicios típicos para estos casos corresponden a: [1] Determinar los primeros N términos de la secuencia, [2] Determinar los términos de la secuencia menores que N, [3] Determinar los términos comprendidos en el rango abierto de N y M. El problema podría referirse a sumar los términos calculados, o determinar promedios/porcentajes, etc.
- Para calcular, desde el segundo término en adelante, la secuencia de Fibonacci, se requiere recordar los dos últimos términos calculados. El concepto de *recordar* valores obtenidos de cálculos anteriores, se conoce como *un historial* y es interesante desde el punto de vista de programación, porque requiere el uso de variables auxiliares, asignaciones o cambios de valores adecuados.
- **SERIES:** Una serie es la suma de los términos de una sucesión. Se representa una serie con términos a_n como se muestra a continuación:

$$\sum_{i=1}^N a_n$$

donde N es el índice final de la serie.

A continuación, se presenta, como ejemplo, la serie para el cálculo de: **seno de x** y **e^x**

$$\sin(x) = x + \frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} \dots, \quad -\infty < x < \infty$$

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, \quad -\infty < x < \infty$$

En el desarrollo de programas para el cálculo de series es muy importante hacer un análisis previo, detallado y preciso de [1] cómo el programa generara el término en sí y [2] cómo se conforman los términos de la serie.

Para la generación de los términos, el programador puede diseñar el programa de tal manera que se trabaje:

- a. CON EL TÉRMINO COMO UN TODO, en cuyo caso la transición de los elementos se realiza sobre una sola variable. Esta variable se comporta como una productoria del término anterior por la transición para generar el nuevo término. Esta técnica no es apropiada para aquellos casos en que el término de la serie conlleva una sumatoria.
- b. CON LOS ELEMENTOS QUE CONFORMAN EL TÉRMINO, en este caso se descompone el término en elementos básicos como sumatorias, productorias, signos alternos, cálculos previos o internos.

Entre los elementos de programación más usados en los programas de series, se encuentran:

- **CONTADORES:** Se usan típicamente para contar el número de términos a calcular, para calcular pares/impares y definir paridad
- **SUMATORIAS:** Variables que almacenan el valor de la serie, o de porciones de los términos
- **PRODUCTORIAS:** Variables apropiadas para el cálculo de factoriales y potencias
- **BANDERAS:** Variables necesarias para detectar cuando debe cambiarse el signo o invertir los términos, o para cálculos especiales que requieran la determinación de un evento
- **CÁLCULOS PREVIOS:** Se refiere a los cálculos necesarios para obtener el primer término de la serie a partir del cual se generarán los siguientes. Tal como el caso de un factorial decreciente o potencias decrecientes

- **CALCULOS INTERNOS:** Caso que se presenta cuando no es posible determinar el término a generar a partir del anterior. Como por ejemplo el caso de una potencia donde la base cambia entre términos.

Ejercicio 4-6: Dados N y X, desarrollar el Diagrama de Flujo de un programa que calcule y escriba cada término de la serie y la suma de los N primeros términos.

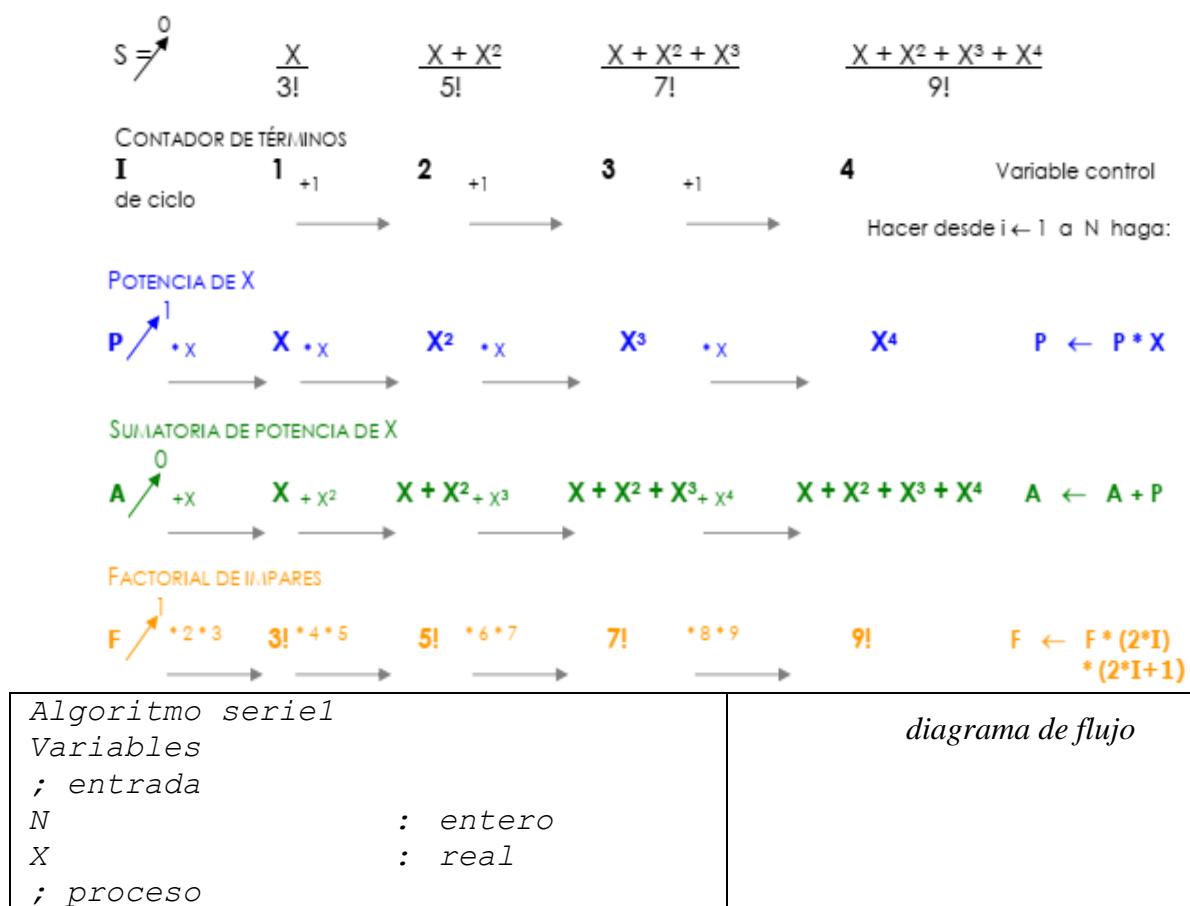
$$S = \frac{X}{3!} + \frac{X + X^2}{5!} + \frac{X + X^2 + X^3}{7!} + \frac{X + X^2 + X^3 + X^4}{9!} + \dots$$

Análisis del ejercicio:

¿QUE SE TIENE? Valores N y X leídos. Se asume que $N > 0$

¿QUE SE PIDE? Imprimir los primeros N términos y sumarlos, al final imprimir la suma

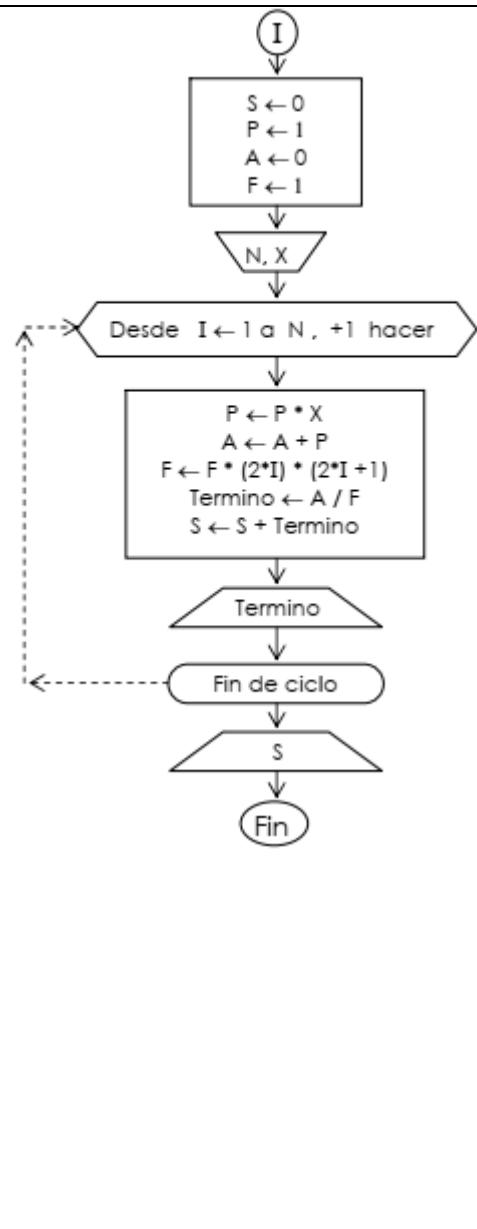
¿CÓMO LOGRARLO?



```

A : real
P : real
F : entero
; salida
S : real
Inicio
Imprimir "# de Términos:"
Leer N
Imprimir "Valor de X:"
Leer x
; inicio contadores y acumuladores
S<-0
P<-1
A<-0
F<-1
; Ciclo para generar los términos
Para I<- 1 hasta N hacer
    ; elementos del término
    P <- p * x
    A <- A + P
    F <- F * (2 * i) * (2 * i -1)
    ; TÉRMINO
    Termino <- A/F
    ; Uso EL TÉRMINO
    S <- S + termino
Finpara
Imprimirln" La suma de los primeros
", N, "términos de la
serie es ", S
Fin

```



Ejercicio 4-7: Dados los valores de X y R, elabore el Diagrama de Flujo de un programa que determine el valor aproximado de $e^{-\left(\frac{x^2}{2}\right)}$ según el desarrollo de los primeros R términos, empleando el método de Taylor, el cual se muestra a continuación,

$$e^{-\left(\frac{x^2}{2}\right)} = 1 - \frac{x^2}{2 * 1!} + \frac{x^4}{4 * 2!} - \frac{x^6}{6 * 3!} + \frac{x^8}{8 * 4!} - \dots$$

Análisis del Problema:

ELEMENTO:	FORMULA	INICIO
i : 1 2 3 4 5		

signo:	+	-	+	-	+	signo <- signo	signo <-1
potX:	1	x²	x⁴	x⁶	x⁸	potX <- potX * x * x	potX <- 1
fact:	1	1!	2!	3!	4!	fact <- fact * i i>1	fact <- 1
par:	1	2	4	6	8	par 2 * i i>1	par 1

Algoritmo serie2

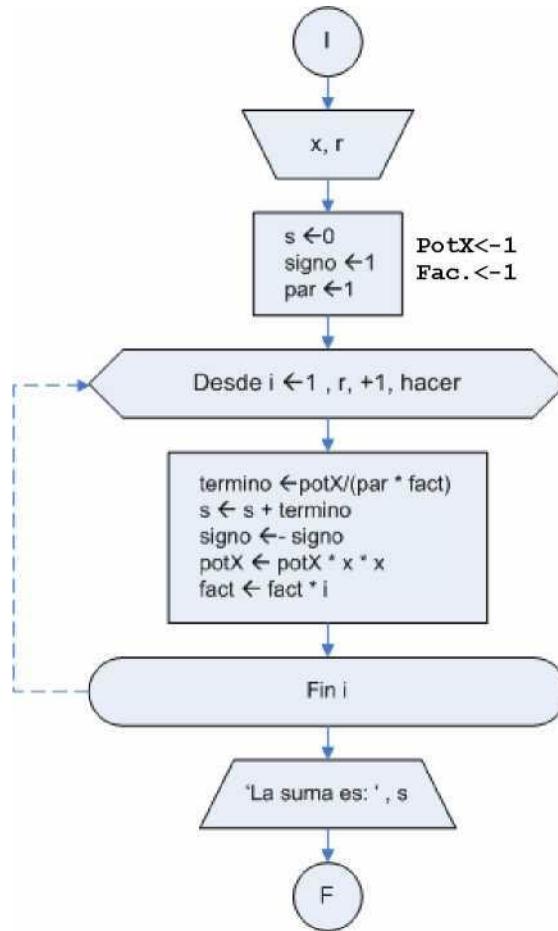
Variables

; entrada

```

N :      entero
R :      real
; proceso
Signo : real
Par : real
Potx : real
; salida
S : real
Inicio
    Imprimir "# de Términos:"
    Leer N
    Imprimir "Valor de X:"
    Leer x
    S<-0
    signo <-1
    par <-0
    potx <-1
    fact <- 1
    Para i <- 1 hasta N hacer
        termino <- potx
        / (par * fact)
        S <- s + termino
        signo <- - signo
        potx <- potx * x * x
        fact <- fact * i
    Finpara
    Imprimirln " La suma de
los primeros ", N, "terminos
de la serie es ", S
Fin

```



Código 4-14. serie2.py

```

...
Declarar variables de entrada
n
x
Declarar variables de salida
suma
Declarar variables de proceso

```

```

termino
signo
numerador
denominador
fact
par
i
...
# lectura de la cantidad de términos y de la variable de la serie

n = int(input("n: "))
x = float(input("x: "))

# Inicializar las variables
suma = 0
signo = 1
fact = 1
par = 1

# Proceso
for i in range(n):
    # Calcular cada elemento del término
    if i == 0:
        termino = 1
    else:
        signo = (-1) ** i
        numerador = x ** (2 * i)
        fact *= i
        par = 2 * i
        denominador = par * fact
        # Calcular el término
        termino = signo * numerador / denominador

    # Sumar el término
    suma += termino
    print("Término (", i, ") = ", termino)

# Impresión de resultado
print("Suma=", suma)

```

Ejecución del programa `serie1.py`:

```

TERMINAL PROBLEMAS SALIDA CONSO

PS C:\Users\bolivar> & C:/WPy64-3910/
AMACIÓN/PARCIAL 1/serie 1.py"
n: 3
x: 2
Termino ( 0 ) =  1
Termino ( 1 ) = -2.0
Termino ( 2 ) =  2.0
Suma= 1.0

```

Ejercicio 4-8: Dados H y M, desarrolle un programa que imprima y sume los M primeros términos de la siguiente serie y el resultado de la sumatoria:

$$S = -\frac{H * (1)^M}{2!} + \frac{H^2 * (1+3)^M}{2+4} + \frac{H^3 * (1+3+5)^M}{(2+4+6)!} + \frac{H^4 * (1+3+5+7)^M}{2+4+6+8} + \dots$$

Elemento:	Términos			Fórmula	Inicializar
i:	0	1	2		
signo:	-	+	+		Signo = -1
suma_impar	1	1+3	1+3+5	suma_impar += (2 * i + 1)	suma_impar = 0
suma_pares	2	2+4	2+4+6	suma_pares += (2 * i + 2)	suma_pares = 0
numerador	$-H * (1)^M$	$+H^2 * (1+3)^M$	$+H^3 * (1+3+5)^M$	numerador = signo * H ^ (i + 1) * suma_impar ** M	
denominador	$2!$	$2+4$	$(2+4+6)!$	denominador_par = suma_pares! denominador_impar = suma_pares	
término	$\frac{-H * (1)^M}{2!}$	$\frac{H^2 * (1+3)^M}{2+4}$	$\frac{H^3 * (1+3+5)^M}{(2+4+6)!}$	término = numerador/denominador	

Código 4-15. serie2.py

```

'''
Declaración de Variables:
Variables de entrada (Qué tengo?)
h: float, es el Número de variable de la serie
m: int, es la cantidad de Términos
Variables de salida (Qué quiero?)
Suma: float, es la suma de los términos
Termino: float, es el valor de cada término
Variables de proceso (Cómo lo logro?)
I: int, es el contador del ciclo
Signo: int, es el signo del termino
suma_impar: int, es el números del numerador
suma_pares: int, es el número del denominador
numerador: float, es el valor de todo el numerador
denominador: float, es el valor de todo el denominador
'''
```

```

# Lecturas correspondientes
# Se leen los valores de número de variable de la serie (h) y términos (m)

h = float(input("Introduzca el valor de la variable de la serie (h): "))
m = int(input("Introduzca el valor del número de términos (m): "))

# Inicializaciones correspondientes
suma = 0
signo = -1
suma_impares = 0
suma_pares = 0

# Determinación de los términos de la serie
for i in range(m): # Para los primeros m términos
    # Cálculo de cada elemento del término
    suma_impares += (2 * i + 1)
    suma_pares += (2 * i + 2)
    numerador = signo * h ** (i + 1) * suma_impares ** m
    if i % 2 != 0: # Los términos pares no tienen factorial
        denominador = suma_pares
    else: # Los términos impares tienen factorial
        factorial = 1
        for j in range(suma_pares):
            factorial *= j + 1

        denominador = factorial

    termino = numerador / denominador # Determinación del término e impresión
    print("Término: ", i + 1, " = ", termino)
    signo = 1 # El resto de los términos son positivos
    suma += termino # Se acumula la suma

# Se imprime la suma
print("La suma de los ", m, " primeros términos es: ", suma)

```

Ejecución del programa serie1.py:

```

Introduzca el valor de la variable de la serie (h): 3
Introduzca el valor del número de términos (m): 3
Término: 1 = -1.5
Término: 2 = 96.0
Término: 3 = 4.109172077922078e-05
La suma de los 3 primeros términos es: 94.50004109172077

```

4.8 Composición y descomposición de un número.

Un sistema de numeración es un conjunto de símbolos y reglas que se utilizan para representar y operar con cantidades. En los sistemas de numeración existe un elemento

característico que define el sistema y se denomina base, siendo ésta el número de símbolos que se utilizan para la representación. Así, un sistema de base diez utiliza 10 símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), un sistema de base dos utiliza 2 símbolos (0, 1), un sistema de base cuatro utiliza 4 símbolos (0, 1, 2, 3). Desde hace años, el hombre utiliza como sistema de numeración el sistema decimal, derivado del sistema numérico indo-árabigo y muy posiblemente fundamentado en los diez dedos de las manos.

Los sistemas de numeración actuales son sistemas posicionales, en los que el valor que representa cada símbolo o cifra depende de su valor absoluto y de la posición relativa que ocupa la cifra con respecto al resto. Siendo entonces el dígito menos significativo aquel ubicado en la posición relativa de menor valor; y el dígito más significativo aquel ubicado en la posición relativa de mayor valor. Para un número entero de cinco dígitos: el dígito menos significativo es el dígito ubicado en la posición relativa: unidad y el dígito más significativo es el dígito en la posición relativa: decena de miles. Por ejemplo: para el numero: 1 2 3 4 5 el dígito menos significativo es: 5 y el dígito más significativo es: 1

Las tareas más comunes para la manipulación de dígitos que conforman un número son: (1) Componer un número a partir de sus dígitos; (2) Determinar la cantidad de dígitos que conforman un número; (3) Desglosar un número en sus dígitos; entre otras.

4.8.1 Algoritmos para Determinar la Cantidad de dígitos de un número Entero

Para determinar la cantidad de dígitos de un número hay que ir comparando el número con la Unidad seguida de ceros hasta encontrar el valor de la unidad seguida de tantos ceros que supere a nuestro número, la cantidad de veces que comparamos es la cantidad de dígitos del número. Ejemplo:

NUMERO (n)	UNIDAD SEGUIDA DE CEROS (u)	n >= u	cd	CÓDIGO	
45897	1	Si	1	u = 1 cd = 0 mientras n >= u u = u * 10 cd = cd + 1	Observe que al finalizar el proceso u contiene un dígito más que el número original
	1 * 10	Si	2		
	10 * 10	Si	3		
	100 * 10	Si	4		
	1000 * 10	Si	5		
	10000 * 10	No			

Otro Forma es partir de la unidad, pero considerando que el número tiene por lo menos un dígito, se compara con el valor de la Unidad *10:

NUMERO (n)	UNIDAD SEGUIDA DE CEROS (u)	$n >= u * 10$	cd	CÓDIGO	Observe que al finalizar el proceso u contiene la misma cantidad de dígitos que el número original
58975	1	Si	2	$u = 1$ $cd = 1$ mientras $n >= u * 10$ $u = u * 10$ $cd = cd + 1$	
	$1 * 10$	Si	3		
	$10 * 10$	Si	4		
	$100 * 10$	Si	5		
	$1000 * 10$	No			

4.8.2 Descomposición de un número en sus dígitos, del menos significativo al más significativo o de derecha a izquierda

El proceso consiste en determinar el dígito como el residuo de dividir el número por 10 y repetir el proceso con un nuevo número formado por el residuo de la división por 10, mientras existan dígitos.

Por ejemplo: Considere el número: 145897

NUMERO	DÍGITO	NUEVO NUMERO	PROCESO
45897	$45897 \% 10 =$	7	$45897 // 10 = 4589$
4589	$4589 \% 10 =$	9	$4589 // 10 = 458$
458	$458 \% 10 =$	8	$458 // 10 = 45$
45	$45 \% 10 =$	5	$45 // 10 = 4$
4	$4 \% 10 =$	4	$4 // 10 = 0$

Note que este algoritmo descompone el número desde el dígito menos significativo al dígito más significativo:
7 9 8 5 4

4.8.3 Descomposición de un número en sus dígitos, del más significativo al menos significativo o de izquierda a derecha

El proceso consiste en determinar el dígito como el cociente de dividir el número por la unidad seguida de tantos ceros como la posición del dígito dentro del número original y repetir el proceso con un nuevo número formado de quitar el dígito obtenido, mientras existan dígitos.

Por ejemplo: Considere el número: 145897

NÚMERO	DÍGITO	NUEVO NÚMERO	PROCESO
--------	--------	--------------	---------

45897	$45897 \text{ // } 10000 = 4$	$45897 \% 10000 = 5897$	1. Determinar la Potencia para obtener el primer dígito (P)
5897	$5897 \text{ // } 1000 = 5$	$5897 \% 1000 = 897$	
897	$897 \text{ // } 100 = 8$	$897 \% 100 = 97$	
97	$97 \text{ // } 10 = 9$	$97 \% 10 = 7$	
7	$7 \text{ // } 1 = 7$	$7 \% 1 = 0$	2. Determinar cada dígito mientras $P > 0$ $\text{dig} = n \text{ // } p$ $n = n \% p$ $p = p \text{ // } 10$

Note que este algoritmo descompone el número desde el dígito más significativo al dígito menos significativo:

4 5 8 9 7

Determinar la Potencia para obtener el primer dígito (P)

Usando el algoritmo para determinar la cantidad de dígitos.

$p = u \text{ // } 10$ {Para quitarle el dígito de más}

Usando el algoritmo 1, para determinar la cantidad de dígitos.

$p = u$

Los procesos de descomposición de números son destructivos, lo que implica que los valores originales de la variable se pierden en el proceso de descomposición, si requiere de estos valores una vez efectuado el proceso de descomposición, debe guardar una copia antes de iniciar el proceso de descomposición.

4.8.4 Composición de números a partir de sus dígitos, del más al menos significativo o de izquierda a derecha

El proceso consiste en acumular cada dígito, en el número multiplicado por 10.

Por ejemplo: Consideré los dígitos: 7 9 8 5 4 [siendo 7 el dígito más significativo y 4 el dígito menos significativo]

DIGITO	AGREGANDO EL DIGITO	NUEVO NUMERO	CÓDIGO
7	$0 * 10 + 7 = 7$	$0 + 7 = 7$	$cent = 0$
9	$7 * 10 + 9 = 79$	$70 + 9 = 79$	$n = 0$
8	$79 * 10 + 8 = 798$	$790 + 8 = 798$	$mientras cent == 0$
5	$798 * 10 + 5 = 7985$	$7980 + 5 = 7985$	$leer(dig, cent)$
4	$7985 * 10 + 4 = 79854$	$79850 + 4 = 79854$	$n = n * 10 + dig$

Note que este algoritmo compone el número a partir del dígito menos significativo al dígito más significativo: **79854**.

Nota: Para el ejemplo se leen los dígitos, pero pueden provenir de otro proceso, en cuyo caso no se leerían.

4.8.5 Composición de números a partir de sus dígitos, del menos al más significativo o de derecha a izquierda

El proceso consiste en acumular el producto de cada dígito por una potencia de 10, que indicará la posición del dígito dentro del número, es decir, si ocupa la decena se multiplica por 10, si ocupa la centena por 100 y así sucesivamente.

Por ejemplo: Consideré los dígitos: **7 9 8 5 4** [siendo 7 el dígito menos significativo y 4 el dígito más significativo]

DIGITO	AGREGANDO EL DIGITO	NUEVO NUMERO	CÓDIGO
7	$7 * 1 = 7$	$0 + 7 = 7$	$cent = 0$
9	$9 * 10 = 90$	$7 + 90 = 97$	$p = 1$
8	$8 * 100 = 800$	$97 + 800 = 897$	$n = 0$
5	$5 * 1000 = 5000$	$897 + 5000 = 5897$	$mientras cent = 0$
4	$4 * 10000 = 40000$	$5897 + 40000 = 45897$	$leer(dig, cent)$ $n = n + dig * p$ $p = p * 10$

Note que este algoritmo compone el número a partir del dígito menos significativo al dígito más significativo: **45897**.

Nota: Para el ejemplo se leen los dígitos, pero pueden provenir de otro proceso, en cuyo caso no se leerían.

Ejercicio 4-9: Escribir un programa que solicite números positivos al usuario. Mostrar el número cuya sumatoria de dígitos fue mayor y la cantidad de números cuya sumatoria de dígitos fue menor que 10.

Código 4-16. dígitos_suma_cant.py

```

cantidad = 0
mayor = -1
numero = int(input("Número positivo: "))
while numero >= 0:
    suma = 0
    while numero != 0:
        digito = numero % 10
        suma += digito
        numero //= 10
    if suma > mayor:
        mayor = suma
        cantidad = 1
    else:
        cantidad += 1
print("El número con mayor sumatoria de dígitos es:", mayor)
print("La cantidad de números con sumatoria de dígitos menor que 10 es:", cantidad)

```

```

if suma > mayor:
    mayor = suma
n_mayorsuma = numero

if suma < 10:
    cantidad += 1

numero=int(input("Número positivo: "))
print("Sumatoria de dígitos de ",n_mayorsuma,":",mayor)
print("Cantidad con sumatoria menor a 10: ",cantidad)

```

4.9 Glosario

anidamiento: Una estructura de programa dentro de otra; por ejemplo, una sentencia condicional dentro de una o ambas ramas de otra sentencia condicional.

bloque: Grupo de sentencias consecutivas con el mismo sangrado.

condición: La expresión booleana de una sentencia condicional que determina qué rama se ejecutará.

cuerpo: En una sentencia compuesta, el bloque de sentencias que sigue a la cabecera de la sentencia.

expresión booleana: Una expresión que es cierta o falsa.

operador de comparación: Uno de los operadores que comparan dos valores: ==, !=, >, <, >= y <=.

operador lógico: Uno de los operadores que combinan expresiones booleanas: *and*, *or* y *not*.

operador módulo: Operador, señalado con un signo de tanto por ciento (que trabaja sobre enteros y devuelve el resto cuando un número se divide entre otro).

sentencia compuesta: Estructura de Python que está formado por una cabecera y un cuerpo. La cabecera termina en dos puntos (:). El cuerpo tiene una sangría con respecto a la cabecera.

sentencia condicional: Sentencia que controla el flujo de ejecución de un programa dependiendo de cierta condición.

4.10 Resumen

Para poder tomar decisiones en los programas y ejecutar una acción u otra, es necesario contar con una **estructura condicional**.

Las **condiciones** son expresiones *booleanas*, es decir, cuyos valores pueden ser *verdadero* o *falso*, y se escriben mediante operadores entre distintos valores.

Mediante **expresiones lógicas** es posible modificar o combinar expresiones booleanas.

La estructura condicional puede contener, opcionalmente, un bloque de código que se ejecuta si no se cumplió la condición.

Es posible *anidar* estructuras condicionales, colocando una dentro de otra.

También es posible *encadenar* las condiciones, es decir, colocar una lista de posibles condiciones, de las cuales se ejecuta la primera que sea verdadera.

En esta unidad también hemos usado las dos estructuras de control que permiten repetir instrucciones.

Además de los *ciclos definidos*, en los que se sabe cuáles son los posibles valores que tomará una determinada variable, existen los *ciclos indefinidos*, que se terminan cuando no se cumple una determinada condición.

Se utiliza la función *range()* para generar una lista de valores automática que nos facilitó el uso de *for*, aunque su uso no es exclusivo de este contexto.

La condición que termina el ciclo puede estar relacionada con una entrada de usuario o depender del procesamiento de los datos.

Se utiliza el método del *centinela* cuando se quiere que un ciclo se repita hasta que el usuario indique que no quiere continuar.

Además de la condición que hace que el ciclo se termine, es posible interrumpir su ejecución con código específico dentro del ciclo.

4.11 Recomendaciones para la escritura de código limpio.



Las presentes recomendaciones son el resultado de la observación directa en ejercicios de diferentes evaluaciones, y que el estudiante debe procurar evitar para tener un código limpio y con un mejor estilo. También, se agregan algunas normas básicas de escritura de código (Adaptado del Manual básico, iniciación a Python 3 por José Miguel Ruiz Torres [6]). Estas buenas prácticas de programación tienen el fin de mejorar las tareas de revisión, intercambio y escalabilidad del código.

1. Los nombres de los archivos deben escribirse en minúscula. Ejemplo: hola.py, calculadora.py
2. La llamada a una función se escribe en minúscula. Ejemplo: print(), input()
3. Dejar un espacio en blanco después de cada coma. Ejemplo:

```
print('Tienes', num_zapatos, 'zapatos y', num_blusas, 'blusas.')
```

4. Dejar un espacio antes y después de cada operador. Ejemplo:

velocidad = distancia / tiempo

total += 6

5. Dejar la sangría con 4 espacios; evita usar la tecla tabuladora. Ejemplo:

```
if nombre == 'Ángel':  
    print('Hola', nombre)  
    input()
```

6. Escriba la expresión lógica de forma clara, sencilla y de forma positiva, es decir, que se debe cumplir para ejecutar el bloque de instrucciones.

7. No escriba condiciones innecesarias, en algunos casos puede hacer analogía con inecuaciones o conjuntos, evalúe la posibilidad de utilizar el descarte como parte de la condición.

8. Si la expresión lógica es muy compleja, puede utilizar una variable para almacenar dicha expresión.
9. Las variables se escriben en minúscula y en caso de estar formadas por varias palabras, éstas van unidas por guiones bajos. Ejemplo: velocidad, altura_inicial
10. Declarar las variables con el tipo de dato adecuado, escribiendo en minúscula dicho tipo de dato. Ejemplo:

nombre: str

contador_alumnos: int

11. Declarar las variables mediante identificadores adecuados. No utilizar identificadores que carezca de significado descriptivo. Con un **código legible** y nombres significativos, el código **se va auto documentado**.
12. Ser consistente al momento de utilizar un estándar para nombres largos, por ejemplo, para una variable que almacena cantidad de alumnos 'contador_alumnos'
13. Declarar variables en líneas separadas, posibilitando agregar una descripción de cada variable mediante comentarios.
14. Declare las variables en orden, recomendablemente: primero las variables de entrada, luego las variables de salida y finalmente las variables de proceso. Para realizar esto:
 - a. Identifique las variables de entrada en el enunciado o planteamiento del problema.
 - b. Identifique las variables de salida, las cuales corresponden a las preguntas en el enunciado.
 - c. Para identificar las variables de proceso, lógicamente tendrá que analizar cómo obtener las variables de salida, entonces debe ir pregunta por pregunta analizando como resolver, por ejemplo: si se le solicita determinar el promedio de estudiantes aprobados, como es el cálculo de un promedio, necesitará un acumulador y un contador, estas son dos variables que debe declarar en la sección de variables de proceso. Otro ejemplo: para determinar el alumno con la mayor nota, requerirá de una bandera, una variable para comparar la mayor nota y al menos una variable que guarde el nombre del estudiante, esto

dependerá de la pregunta del enunciado que indicará cuáles datos se requerirán de ese estudiante.

15. Inicializar las variables que así lo requieren, en especial: variables de tipo cadena (str), contadores, acumuladores, productorias, entre otras.

Opcionalmente puede iniciar la variable al declararla, por ejemplo:

nombre: str = "

16. Escribir los programas lo más simple posible.
17. Escribe los comentarios necesarios en tu código, describiendo cada detalle, para hacer que sea lo más claro y legible posible.
18. Comentar los programas, explicando el propósito, funcionamiento completo y el resultado esperado.
19. Evitar la incorporación de más de una instrucción por línea. Esto reduce notoriamente la legibilidad del código, ya que el programador habitualmente está acostumbrado a leer una instrucción por línea.
20. Realizar la sangría necesaria, si una instrucción abarca más de una línea. Por ejemplo:

```
a = (1 + 2  
     + 3 + 4)
```

21. Utilice los operadores aritméticos adecuadamente, especialmente para la potencia, el producto y para la división, debe escribirla tal cual lo haría de forma digital respetando la sintaxis del lenguaje de programación, por ejemplo:

```
t = i * x**(i + 1) / (2 * i + 1)
```

22. Los operadores lógicos tienen su jerarquía, realice la corrida en frío del siguiente ejercicio, después escriba y ejecute este código en Python para que compare sus resultados:

```
a:bool  
b:bool  
c:bool
```

```

d:bool

a = True
b = False
c = False
d = False

print(a or b and c or d) # True primero se evalua and y luego or

```

23. Utilizar la estructura condicional adecuadamente, por ejemplo:

En vez de escribir:

```

if condición1:
    ...
else
    if condición2:
        ...

```

Escriba:

```

if condición1:
    ...
elif condición2:
    ...

```

24. **No Repita Instrucciones** Innecesariamente dentro de los bloques de la estructura condicional, sí esto sucede evalúe la posibilidad de que dicha instrucción se escriba fuera de la estructura, por ejemplo:

En vez de escribir está estructura:

```

if condición:
    ...
    num = a + b
    den = a * b
    t = num / den
else:
    ...
    num = c + d
    den = c * d
    t = num / den

```

Escríbalo de esta manera:

```

if condición:
    ...
    num = a + b
    den = a * b
else:
    ...
    num = c + d
    den = c * d

```

```
t = num / den    # Esta instrucción estaba repetida en la estructura  
                 # condicional anterior, por lo tanto,  
                 # se escribe fuera de la estructura.
```

25. Utilice los operadores lógicos adecuadamente; por ejemplo, si se requiere la condición para los valores de x entre 0 y 5:

```
cond = 0 < x < 5
```

26. Con el fin de evitar errores de ejecución, realice la validación de las fórmulas o ecuaciones en los casos de que exista la posibilidad de una indeterminación, como es el caso de promedio, porcentaje, o en series numéricas. Por ejemplo, si se calcula el promedio de notas de aprobados como: `prom_aprob = suma_notas_aprob / cant_prob`, resultaría infinito si `cant_aprob` es cero, es una posibilidad remota; pero puede suceder, para ello se debe programar de la siguiente manera:

```
if cant_prob == 0:  
    print("No hubo aprobados")  
else:  
    prom_aprob = suma_notas_aprob / cant_prob  
    print(f"El Promedio de Aprobados es = {prom_aprob:.2f}")
```

De esta manera, se captura un posible error de ejecución en caso de que no haya estudiantes aprobados.

27. Utilice la estructura repetitiva adecuada, sí conoce la cantidad de veces a ejecutar el ciclo, utilice `for`, si las iteraciones dependen de una condición determinada, utilice `while`.

Unidad 5

Datos Estructurados

Objetivos de la unidad:

En esta unidad aprenderá a utilizar los tipos de datos estructurados, algunos métodos y como recorrer los elementos que estos datos contienen.

Datos Estructurados

5.1 Manejo Básico de Cadenas

Una *cadena* es una sucesión de caracteres encerrada entre comillas (simples o dobles). Python ofrece una serie de operadores y funciones predefinidos que manejan cadenas o devuelven cadenas como resultado [4].

Para seleccionar un carácter según la posición en la cadena, se escribe entre corchetes “[]” un valor numérico entero que representa el índice del elemento. Por ejemplo:

```
>>> fruta = 'banana'  
>>> letra = fruta[1]  
>>> print (letra)
```

La expresión fruta[1] selecciona el carácter en la posición 1 de la variable fruta. La variable letra apunta al valor siguiente:

```
a
```

La primera letra de "banana" no es *a*, en computación siempre se empieza a contar desde cero. El contenido almacenado en la posición cero de "banana" es *b*. El valor contenido en la posición 1 es *a*, y el de la posición 2 es la letra *n*.

Sí se requiere la primera letra de una cadena, simplemente escribe 0, o cualquier expresión de valor 0, entre los corchetes:

```
>>> letra = fruta[0]  
>>> print (letra)  
b
```

Operador + (concatenación de cadenas): acepta dos cadenas como operandos y devuelve la cadena que resulta de unir la segunda a la primera.

```
>>> 'Asignatura= ' + 'Computación I'  
'Asignatura= Computación I'
```

Concatenar con datos que no sean cadenas de caracteres: Puedes concatenar con variables que no son cadena simplemente convirtiendo el valor en una cadena usando la función **str()** como se muestra a continuación:

```
str = "This is test number " + str(15)  
print (str)
```

Operador * (repetición de cadena): acepta una cadena y un entero y devuelve la concatenación de la cadena consigo misma tantas veces como indica el entero.

```
>>> 'Computación I '*3  
'Computación I Computación I Computación I '
```

Operador % (sustitución de marcas de formato): acepta una cadena y una o más expresiones (entre paréntesis y separadas por comas) y devuelve una cadena en la que las marcas de formato (secuencias como %d, %f, etc.) se sustituyen por el resultado de evaluar las expresiones.

```
>>> velocidad = 25.123456
```

Impresión con dos decimales

```
>>> print ('velocidad= %.2f' % velocidad)  
velocidad= 25.12
```

Impresión sin decimales

```
>>> print ('velocidad= %d' % velocidad)  
velocidad= 25
```

O para trabajar con cadenas, imprimiendo un nombre

```
>>> nombre = 'Juan'  
>>> print ('Hola, %s! ' % nombre)  
Hola, Juan!
```

En la Tabla 5-1 se muestra los caracteres de formato.

Tabla 5-1. Caracteres de formato de cadena

Character	Description
d or I	Decimal (base 10) integer
f	Floating point number
s	String or any object

c	Single character
u	Unsigned decimal integer
X or x	Hexadecimal integer (upper or lower case)
0	Octal integer
e or E	Floating point number in exponential form
g or G	Like Xf unless exponent < -4 or greater than the precision. If so, acts like Xe or XE
r	repr() version of the object'
X	Use XX to print the percentage character.

Para imprimir dos o más especificadores de argumento, se utiliza una tupla (datos entre paréntesis):

```
>>> nombre = 'Juan'
>>> edad = 23
>>> print ('%s tiene %d años. ' % (nombre, edad))
Juan tiene 23 años.
```

O una tupla para accesar según su posición.

```
>>> tupla = ('Juan',23)
>>> print ('%s tiene %d años. ' % (tupla [0], tupla [1]))
Juan tiene 23 años.
```

También funciona para una lista.

```
>>> lista = ['Juan',23]
>>> print ('%s tiene %d años. ' % (lista[0], lista[1]))
Juan tiene 23 años.
```

En la Tabla 5-2 se muestra un resumen de los formatos de cadena.

Tabla 5-2. Secuencia de Escape

Sequence	Description
\n	Newline (ASCII LF)
\'	Single quote
\"	Double quote
\	Backslash
\t	Tab (ASCII TAB)
\b	Backspace (ASCII BS)
\r	Carnage return (ASCII CR)
\xhh	Character with ASCII value hh in hex
\ooo	Character with ASCII value ooo in octal
\f	Form feed (ASCII FF)*
\a	Bell (ASCII BEL)

\V

Vertical tab (ASCII VT)

5.1.1 Longitud de una cadena

La función **len** devuelve el número de caracteres de una cadena:

```
>>> fruta = 'banana'  
>>> len(fruta)  
6
```

Para obtener la última letra de una cadena puede intentar realizar la siguiente instrucción:

```
longitud = len(fruta)  
ultima = fruta[longitud] # ERROR!
```

Eso no funcionaría. Provoca un error en tiempo de ejecución *IndexError: string index out of range*. La razón es que no hay una sexta letra en "banana". Como **se empieza a contar por cero**, las seis letras están numeradas del 0 al 5. Para obtener el último carácter se resta 1 a la variable longitud.

```
longitud = len(fruta)  
ultima = fruta[longitud-1]
```

De forma alternativa, se puede utilizar índices negativos, que cuentan hacia atrás desde el final de la cadena. La expresión `fruta[-1]` devuelve la última letra. `fruta[-2]` nos da la penúltima, y así sucesivamente.

5.1.2 Porciones de cadenas

Llamamos **porción** a un segmento de una cadena. La selección de una porción es similar a la selección de un carácter:

```
>>> s = 'Pedro, Pablo, y María'  
>>> print(s[0:5])  
Pedro  
>>> print(s[7:12])  
Pablo  
>>> print(s[15:20])  
María
```

El operador [n:m] devuelve la parte de la cadena desde el n-ésimo carácter hasta el m-enésimo", incluyendo el primero, pero excluyendo el último (posición m). Este

comportamiento contradice a nuestra intuición; tiene más sentido si imagina los índices señalando entre los caracteres, como en el siguiente diagrama:

Fruta → “	b	a	n	a	n	a	“
Índice	0	1	2	3	4	5	

Si omite el primer índice (antes de los dos puntos), la porción comienza al principio de la cadena. Si omite el segundo índice, la porción llega al final de la cadena. Así:

```
>>> fruta = 'banana'  
>>> fruta[:3]  
'ban'  
>>> fruta[3:]  
'ana'
```

¿Qué valor devuelve `fruta[:]`?

Si utilizas un número negativo, el recuento comenzará hacia atrás, como lo ejemplo `fruta[-1]` que imprime el último carácter de la cadena.

5.1.3 Las cadenas son inmutables

Es tentador usar el operador `[]` en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena. Por ejemplo:

```
saludo = 'Hola, mundo'  
saludo[0] = 'M' # ERROR!  
print (saludo)
```

En lugar de presentar la salida Mola, mundo, este código presenta el siguiente error en tiempo de ejecución *TypeError: object doesn't support item assignment*.

Las cadenas son **inmutables**, lo que significa que no puede cambiar una cadena existente. Lo más que puede hacer es crear una nueva cadena que sea una variación de la original:

```
saludo = 'Hola, mundo'  
nuevosaludo = 'M' + saludo[1:]  
print (nuevosaludo)
```

Aquí la solución es concatenar una nueva primera letra a una porción de saludo. Esta operación no tiene efectos sobre la cadena original.

Se puede manejar cadenas, además, mediante métodos que les son propios (ver Anexo A).

5.1.4 Buscar una subcadena

Puedes buscar una subcadena utilizando el método `find`:

```
str = "welcome to likegeeks website"
print(str.find("likegeeks"))
```

El método `find` imprime la posición de la primera aparición de la cadena *likegeeks* si se encuentra.

Si no encuentra nada, devolverá -1 como resultado.

La función de `find` comienza desde el primer carácter, sin embargo, puede comenzar desde el enésimo carácter de esta manera:

```
str = "welcome to likegeeks website"
print(str.find("likegeeks",12))
```

Como comenzamos desde el carácter 12, no hay una palabra llamada *likegeeks* desde esa posición, por lo que devolverá -1.

5.1.5 Reemplazar cadenas

Puedes reemplazar una cadena utilizando el método `replace` de la siguiente forma:

```
str = "This website is about programming"
str2 = str.replace("This", "That")
print(str2)
```

Si tiene muchas ocurrencias y deseas reemplazar solo la primera, puede especificar eso:

```
str = "This website is about programming I like this website"
str2 = str.replace("website", "page",1)
print(str2)
```

Solo la primera palabra fue reemplazada.

5.1.6 Strip de cadenas

Puedes recortar los espacios en blanco de una cadena usando el método `strip` la siguiente forma:

```
str = " This website is about programming "
print(str.strip())
```

Puedes quitar solo los de la derecha o izquierda utilizando los métodos ***rstrip()*** o ***lstrip()*** respectivamente.

5.1.7 Cambiar entre Caracteres en mayúscula y minúsculas

Puedes cambiar entre mayúsculas y minúsculas los caracteres si necesitas compararlos.

```
str="welcome to likegeeks"
print(str.upper())
print(str.lower())
```

5.1.8 Convertir cadenas en números

Tenemos la función ***str()*** que convierte el valor a una cadena, pero esta no es la única función de conversión en la programación en Python.

Existen ***int()***, ***float()***, ***long()*** y otras funciones de conversión que puede usar.

La función ***int()*** convierte la entrada a un entero, mientras que la función ***float()*** convierte la entrada en ***float***.

```
str="10"
str2="20"
print(str+str2)
print(int(str)+int(str2))
```

La primera línea de ***print*** simplemente concatena los dos números sin sumar, mientras que la segunda línea de ***print*** suma los dos valores e imprime el total.

5.1.9 Contar Cadenas

Puede utilizar las funciones ***min()***, ***max()*** y ***len()*** para calcular el valor mínimo o máximo de los caracteres o la longitud total de estos.

```
str="welcome to likegeeks website"
print(min(str))
print(max(str))
print(len(str))
```

5.1.10 Iteraciones sobre cadenas

Puedes iterar sobre las cadenas y manipular cada carácter individualmente de la siguiente forma, este tema se estudiará en estructuras repetitivas:

```
str="welcome to likegeeks website"
for i in range(len(str)):
    print(str[i])
```

La función **len ()** cuenta la longitud de los objetos.

5.1.11 Codificar cadenas

En Python 3, todas las cadenas se almacenan como cadenas Unicode de forma predeterminada, por otro lado, en Python 2 se necesita codificar las cadenas de esta manera:

```
str = "welcome to likegeeks website"
str.encode('utf-8')
```

5.2 Listas

Al igual que una cadena, una lista es una secuencia de valores. En una cadena, los valores son caracteres; en una lista, pueden ser de cualquier tipo. Los valores en las listas reciben el nombre de **elementos**, o a veces **artículos**.

Hay varios modos de crear una lista nueva; el más simple consiste en encerrar los elementos entre corchetes ([y]):

Como es lógico, puedes asignar listas de valores a variables:

```
>>> quesos = ['Cheddar', 'Edam', 'Gouda']
>>> numeros = [17, 123]
>>> lista_anidada = ['spam', 2.0, 5, [10, 20]]
>>> vacia = []
```

5.2.1 Las listas son mutables

La sintaxis para acceder a los elementos de una lista es la misma que para acceder a los caracteres de una cadena, con el operador corchete. La expresión dentro de los corchetes especifica el índice. Recuerda que los índices comienzan por 0:

```
>>> print(quesos[0])  
Cheddar
```

A diferencia de las cadenas, las listas son mutables (pueden mutar), **porque puedes cambiar el orden de los elementos o reasignar un elemento dentro de la lista**. Cuando el operador corchete aparece en el lado izquierdo de una asignación, este identifica el elemento de la lista que será asignado.

```
>>> numeros = [17, 123]  
>>> numeros[1] = 5  
>>> print(numeros)  
[17, 5]
```

El elemento de números cuyo índice es uno, que antes era 123, es ahora 5.

Puedes pensar en una lista como una relación entre índices y elementos. Esta relación recibe el nombre de mapeo o direccionamiento; cada índice “dirige a” uno de los elementos.

Los índices de una lista funcionan del mismo modo que los índices de una cadena:

- ✓ Cualquier expresión entera puede ser utilizada como índice.
- ✓ Si se intenta leer o escribir un elemento que no existe, se obtiene un *IndexError*.
- ✓ Si un índice tiene un valor negativo, cuenta hacia atrás desde el final de la lista.

5.2.2 Recorrer una lista

El modo más habitual de recorrer los elementos de una lista es con un ciclo *for*. La sintaxis es la misma que para las cadenas:

```
for queso in quesos:  
    print(queso)
```

Esto funciona correctamente si sólo se necesita leer los elementos de la lista. Pero si quieres escribir o modificar los elementos, necesitarás los índices. Un modo habitual de hacerlo consiste en combinar las funciones *range* y *len*:

```
for i in range(len(numeros)):  
    numeros[i] = numeros[i] * 2
```

Este ciclo recorre la lista y actualiza cada elemento. *len* devuelve el número de elementos de la lista. *range* devuelve una lista de índices desde 0 hasta n - 1, donde n es la longitud de la lista. Cada vez que se recorre el ciclo, i obtiene el índice del elemento siguiente. La sentencia de asignación en el cuerpo utiliza i para leer el valor antiguo del elemento y asignarle el valor nuevo.

Un ciclo *for* aplicado a una lista vacía no ejecuta nunca el código contenido en su cuerpo:

```
for x in vacia:  
    print('Esto nunca ocurrirá.')
```

5.3 Manejo de Tuplas

5.3.1 Las tuplas son inmutables

Una tupla es una secuencia de valores muy parecida a una lista. Los valores almacenados en una tupla pueden ser de cualquier tipo, y están indexados por valores numéricos enteros que inicia en cero. La diferencia más importante es que las tuplas son inmutables. Las tuplas además son comparables y dispersables (hashables), de modo que las listas de tuplas se pueden ordenar y también es posible usar tuplas como valores para las claves en los diccionarios de Python.

Sintácticamente, una tupla es una lista de valores separados por comas:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

A pesar de que no es necesario, resulta corriente encerrar las tuplas entre paréntesis, lo que ayuda a identificarlas rápidamente dentro del código en Python.

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un único elemento, es necesario incluir una coma al final:

```
>>> t1 = ('a',)  
>>> type(t1)  
<type 'tuple'>
```

Sin la coma, Python trata ('a') como una expresión con una cadena dentro de un paréntesis, que evalúa como de tipo “string”:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Otro modo de construir una tupla es usar la función interna *tuple*. Sin argumentos, crea una tupla vacía:

```
>>> t = tuple()
>>> print(t)
()
```

Si el argumento es una secuencia (cadena, lista o tupla), el resultado de la llamada a *tuple* es una tupla con los elementos de la secuencia:

```
>>> t = tuple('altramuces')
>>> print(t)
('a', 'l', 't', 'r', 'a', 'm', 'u', 'c', 'e', 's')
```

Dado que *tuple* es una palabra reservada (es el nombre de un constructor), debe evitar utilizarlo como nombre de variable. La mayoría de los operadores de listas funcionan también con tuplas. El operador corchete indexa un elemento:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Y el operador de sector selecciona un rango de elementos.

```
>>> print(t[1:3])
('b', 'c')
```

Pero si se intenta modificar uno de los elementos de la tupla, se obtiene un error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

No se pueden modificar los elementos de una tupla, pero se puede reemplazar una tupla con otra:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

5.3.2 Asignación de tuplas

Una de las características sintácticas del lenguaje Python que resulta única es la capacidad de tener una tupla en el lado izquierdo de una sentencia de asignación. Esto permite asignar varias variables al mismo tiempo cuando se tiene una secuencia en el lado izquierdo.

En este ejemplo se tiene una lista de dos elementos (por lo que se trata de una secuencia), y se asigna los elementos primero y segundo de la secuencia a las variables x e y en una única sentencia.

```
>>> m = ['pásalo', 'bien']
>>> x, y = m
>>> x
'pásalo'
>>> y
'bien'
>>>
```

Python traduce aproximadamente la sintaxis de asignación de la tupla de este modo:

```
>>> m = ['pásalo', 'bien']
>>> x = m[0]
>>> y = m[1]
>>> x
'pásalo'
>>> y
'bien'
>>>
```

Cuando se utiliza una tupla en el lado izquierdo de la sentencia de asignación, se omite los paréntesis. Pero lo que se muestra a continuación es una sintaxis igualmente válida:

```
>>> m = ['pásalo', 'bien']
>>> (x, y) = m
>>> x
'pásalo'
>>> y
'bien'
>>>
```

Una aplicación especialmente ingeniosa de asignación usando una tupla nos permite intercambiar los valores de dos variables en una única sentencia:

```
>>> a, b = b, a
```

Ambos lados de esta sentencia son tuplas, pero el lado izquierdo es una tupla de variables; el lado derecho es una tupla de expresiones. Cada valor en el lado derecho es

asignado a su respectiva variable en el lado izquierdo. Todas las expresiones en el lado derecho son evaluadas antes de realizar ninguna asignación.

La cantidad de variables en el lado izquierdo y la cantidad de valores en el derecho debe ser la misma:

```
>>> a, b = 1, 2, 3  
ValueError: too many values to unpack
```

Generalizando más, el lado derecho puede ser cualquier tipo de secuencia (cadena, lista o tupla). Por ejemplo, para dividir una dirección de e-mail en nombre de usuario y dominio, podrías escribir:

```
>>> dir = 'monty@python.org'  
>>> nombreus, dominio = dir.split('@')
```

El valor de retorno de *split* es una lista con dos elementos; el primer elemento es asignado a *nombreus*, el segundo a *dominio*.

```
>>> print (nombreus)  
monty  
>>> print (dominio)  
python.org
```

5.4 Glosario

contador: Una variable usada para contar algo, normalmente inicializado a cero e incrementado posteriormente.

incrementar: Aumentar el valor de una variable en una unidad, **decrementar:** Disminuir el valor de una variable en una unidad, **espacio en blanco:** Cualquiera de los caracteres que mueven el cursor sin imprimir caracteres visibles. La constante *string.whitespace* contiene todos los caracteres de espacio en blanco.

índice: Una variable o valor usado para seleccionar un miembro de un conjunto ordenado, como puede ser un carácter de una cadena.

mutable: Un tipo de datos compuesto a cuyos elementos se les puede asignar nuevos valores.

porción: Una parte de una cadena especificada por un intervalo de índices.

recorrer: Realizar de forma iterativa una operación similar sobre cada uno de los elementos de un conjunto.

tipo de datos compuesto: Un tipo de datos en el que los valores están hechos de componentes o elementos que son a su vez valores.

Unidad 6

Archivos de datos secuenciales

En este tema se abarcará las actividades básicas de lectura y escritura de archivos de datos, con estos datos y mediante las herramientas de programación adecuadas se busca resolver los diferentes problemas que se puedan plantear con esta forma sencilla de manejo de archivos.

Objetivos de la Unidad:

En esta unidad aprenderá a manejar los archivos de texto en particular en el lenguaje Python. Esto incluye abrir, cerrar, leer desde y escribir en archivos .txt, .csv, .dat

Archivos de Texto

6.1 Creación y Uso de Archivos de Texto

Un *archivo* o *fichero* es un conjunto de datos estructurados en una colección de entidades elementales o básicas denominadas *registros* o *artículos*, que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas *campos* [1].

El archivo es un medio de almacenamiento separado del programa, que guarda en el mismo una gran cantidad de información. El uso de un archivo para guardar datos de forma externa es recomendable en procesos que no necesiten una organización muy compleja de los datos a gestionar ya que en tal caso lo mejor sería utilizar un programa gestor de base de datos, que ya incorpora, de modo mucho más optimizado, los algoritmos y procesos específicos para el tratamiento de la información.

6.2 Campos

Un campo (ver Figura 6-1) es un ítem o elemento de datos elementales, tales como un nombre, número de empleados, ciudad, número de identificación, etc. Un campo está caracterizado por su tamaño o longitud y su tipo de datos (cadena de caracteres, entero, lógico, etcétera.). Los campos pueden incluso variar en longitud. En la mayoría de los lenguajes de programación los campos de longitud variable no están soportados y se suponen de longitud fija [1].

Figura 6-1. Campos de un registro.

Nombre	Dirección	Fecha de nacimiento	Estudios	Salario	Trienios
--------	-----------	---------------------	----------	---------	----------

Un campo es la unidad mínima de información de un registro.

Los datos contenidos en un campo se dividen con frecuencia en *subcampos*; por ejemplo, el campo fecha se divide en los subcampos día, mes, año.

Campo	0	7	0	7	1	9	9	5
Subcampo	Día	Mes			Año			

Los rangos numéricos de variación de los subcampos anteriores son:

$$1 \leq \text{día} \leq 31$$

$$1 \leq \text{mes} \leq 12$$

$$\leq \text{año} \leq 1987$$

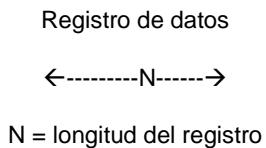
6.3 Registros

Un registro es una colección de información, normalmente relativa a una entidad particular [1]. Un registro es una colección de campos lógicamente relacionados (ver Figura 6-2), que pueden ser tratados como una unidad por algún programa. Un ejemplo de un registro puede ser la información de un determinado empleado que contiene los campos de nombre, dirección, fecha de nacimiento, estudios, salario, trienios, etc.

Los registros pueden ser todos de longitud fija; por ejemplo, los registros de empleados pueden contener el mismo número de campos, cada uno de la misma longitud para nombre, dirección, fecha, etc. También pueden ser de longitud variables.

Los registros organizados en campos se denominan registros lógicos.

Figura 6-2. Registro.



6.4 Creación de archivos de Texto

Para crear un archivo de texto se puede utilizar el editor de texto *Block de Notas* o *Note Pad*, en el mismo se escribirá la lista de valores o datos que se desean procesar

separándolos entre sí por una coma, el orden de los datos debe corresponder con la estructura definida para los mismos, sin líneas vacías, ni títulos.

Suponga que se dice que se tienen los datos de un conjunto de personas donde para cada persona se tiene *Nombre*, *Cédula* y *Nota*, el archivo correcto sería el mostrado a la derecha, el cual se guardará como *datos.dat*.

6.5 Uso de Archivos de Texto en un Programa

Para utilizar un archivo de texto en un programa se debe:

- a. Definir el archivo que se usará y de qué modo se usará.
- b. Usar el archivo según lo definido (Leer (*read*, *readlines*, *readline*) o Guardar (*write*)).
- c. Cerrar el archivo.

6.6 Definir el archivo que se usará y de qué modo se usará

Los archivos en Python son objetos de tipo *file* creados mediante la función *open* (abrir). Esta función toma como parámetros una cadena con la ruta del archivo a abrir, que puede ser relativa o absoluta; una cadena opcional indicando el modo de acceso (si no se especifica se accede en modo lectura) y, por último, un entero opcional para especificar un tamaño de *buffer* distinto del utilizado por **defecto**.

El modo de acceso que es un parámetro opcional, puede ser cualquier combinación lógica de los siguientes modos:

Luis Pérez, 10856576, 17
María Torres, 12474322, 16
Pedro Castro, 1111112, 12

1. ‘r’: read, lectura. Abre el archivo en **modo lectura**, es el modo por defecto. El archivo tiene que existir previamente, en caso contrario se lanzará una excepción de tipo IOError.
2. ‘w’: write, escritura. Abre el archivo en modo escritura. Si el archivo **no existe se crea**. **Si existe**, sobrescribe el contenido.
3. ‘a’: append, añadir. Abre el archivo en modo escritura. Se diferencia del modo ‘w’ **en que en este** caso no se sobrescribe el contenido del archivo, sino que se comienza a **escribir al final del archivo**.
4. ‘b’: **binary, binario**.
5. ‘+’: permite lectura **y escritura simultáneas**.
6. ‘U’: universal newline, saltos de línea **universales**. **Permite trabajar** con archivos que tengan un formato para los saltos de línea que no coincide con el de la plataforma actual (en Windows se utiliza el carácter CR LF, en Unix LF y en Mac OS CR).

```
arch = open(nombre_del_archivo, modo_de_acceso)
```

Por ejemplo, para abrir el archivo *agenda.txt* en modo de lectura, se escribe la siguiente instrucción.

```
arch = open("agenda.txt", 'r')
```

Tras crear la variable (objeto) que representa el **archivo mediante la función *open*** se puede realizar las operaciones de lectura/escritura pertinentes utilizando los métodos **del objeto que veremos en las siguientes secciones**.

Una vez se termine de trabajar con el archivo, se debe cerrarlo utilizando el método *close*.

```
arch.close()
```

6.7 Leer ó tomar información en archivos (texto, csv)

La función *open* crea una referencia a un archivo (usualmente llamado *file handle*) que se usa para leer los datos:

```
open(filename[, mode[, bufsize]])
```

Ejemplo de generación de *file handler* asociado al archivo *mi_archivo.txt*.

```
arch = open('mi_archivo.txt', 'r')
```

La 'r' indica "modo de lectura" y es el modo por defecto (por lo que se podría obviar la 'r').

Sobre el *file handle* (manejador de archivo **arch**) se puede:

- a. `read(n)`: Lee n bytes, por defecto lee el archivo entero.
- b. `readline()`: Devuelve *str* con una sola línea.
- c. `readlines()`: Devuelve una lista con una cadena como elemento por cada línea del archivo.

El Código 6-1 muestra diferentes códigos para realizar la lectura de un archivo de texto.

Código 6-1. Distintas maneras de leer el contenido de un archivo.

```
# script 1
arch = open('archivo.txt')
contenido = arch.read()
print (contenido)

# script 2
arch = open('archivo.txt')
contenido = arch.readlines()
print (contenido)

# script 3
contenido = ''
arch = open('archivo.txt')
while True:
    line = arch.readline()
    contenido += line
    if line == '':
        break
print(contenido)

# Para todos los casos:
arch.close()
```

El resultado de la lectura es el siguiente:

```
Luis Pérez, 10856576, 17
María Torres, 12474322, 16
Pedro Castro, 1111112, 12
['Luis Pérez, 10856576, 17\n', 'María Torres, 12474322, 16\n', 'Pedro Castro, 1111112, 12']
Luis Pérez, 10856576, 17
María Torres, 12474322, 16
```

Pedro Castro, 1111112, 12

Acceso secuencial al contenido de un archivo:

```
arch = open('archivo.txt')
contenido =
for linea in arch:
    contenido += linea
fh.close()
```

Leyendo con 'with' (Python 2.6 en adelante)

Manera genérica:

```
with <Expresión> as <variable>:
    <Código>
```

Ejemplo:

```
with open('archivo.txt') as arch:
    for line in arch:
        print(line)
```

La ventaja de *with* es que, al salir del bloque, se ejecuta un método especial que cierra el archivo automáticamente.

En el caso de leer un archivo con un *encoding* en particular:

```
import codecs
f = codecs.open('arch.txt', encoding='utf-8')
```

6.8 Escribir en un archivo

De la misma forma que para la lectura, existen dos formas distintas de escribir a un archivo.

Mediante cadenas:

```
archivo.write(cadena)
```

O mediante listas de cadenas:

```
archivo.writelines(lista_de_cadenas)
```

Así como la función *read* devuelve las líneas con los caracteres de fin de línea “\n”, será necesario agregar los caracteres de fin de línea a las cadenas que se vayan a escribir en el archivo.

Código 6-2. genera_saludo.py: Genera el archivo saludo.py

```
saludo = open("saludo.py", "w")
saludo.write("""
print "Hola Mundo"
""")
saludo.close()
```

El ejemplo que se muestra en el Código 6-1 contiene un programa Python que a su vez genera el código de otro programa Python.

Modos de escritura:

w: Write, graba un archivo nuevo, si existe, lo borrar.

a: Append (agregar), agrega información al final de un archivo preexistente. Si no existe, crea uno nuevo.

Ejemplo:

```
arch = open('/home/yo/archivo.txt','w')
arch.write('1\n2\n3\n4\n5\n')
arch.close()
```

6.9 Cerrar el Archivo

Al terminar de trabajar con un archivo, es recomendable cerrarlo, por diversos motivos: en algunos sistemas los archivos sólo pueden ser abiertos un programa a la vez; en otros, lo que se haya escrito no se guardará realmente hasta no cerrar el archivo; o el límite de cantidad de archivos que puede manejar un programa puede ser bajo, etc.

Para cerrar un archivo simplemente se debe ejecutar a:

```
arch.close()
```

ADVERTENCIA: Es importante tener en cuenta que cuando se utilizan funciones como *archivo.readlines()*, se está cargando en memoria el archivo completo. Siempre que una instrucción cargue un archivo completo en memoria debe tenerse cuidado de utilizarla sólo con archivos pequeños, ya que de otro modo podría agotarse la memoria de la computadora.

6.10 Ejemplo de procesamiento de archivos

Por ejemplo, para mostrar todas las líneas de un archivo, precedidas por el número de línea, podemos hacerlo como en el Código 6-3.

Código 6-3. numera_lineas.py: Imprime las líneas de un archivo con su número

```
arch = open("archivo.txt")
i = 1
for linea in arch:
    linea = linea.rstrip("\n")
    print (" %4d: %s" % (i, linea))
    i+=1
arch.close()
```

La llamada a `rstrip` es necesaria ya que cada línea que se lee del archivo contiene un fin de línea y con la llamada a `rstrip("\n")` se remueve.



Precauciones al trabajar con archivos

Debes cerrar todos los archivos tan pronto termines de trabajar con ellos. Si la aplicación finaliza normalmente, el sistema operativo cierra todos los archivos abiertos, así que no hay pérdida de información. Esto es bueno y malo a la vez. Bueno porque si olvidas cerrar un archivo y tu programa está, por lo demás, correctamente escrito, al salir todo quedará correctamente almacenado; y malo porque es fácil que te relajes al programar y olvides la importancia que tiene el correcto cierre de los archivos. Esta falta de disciplina hará que acabes por no cerrar los archivos cuando hayas finalizado de trabajar con ellos, pues “ellos solos ya se cierran al final”.

El riesgo de pérdida de información inherente al trabajo con archivos hace que debas ser especialmente cuidadoso al trabajar con ellos. Es deseable que los archivos permanezcan abiertos el menor intervalo de tiempo posible. Si una función o procedimiento actúa sobre un archivo, esa subrutina debería abrir el archivo, efectuar las operaciones de lectura/escritura pertinentes y cerrar el archivo. Sólo cuando la eficiencia del programa se vea seriamente comprometida, deberás considerar otras posibilidades.

Es más, deberías tener una política de copias de seguridad para los archivos de modo que, si alguna vez se corrompe uno, puedas volver a una versión anterior tan reciente como sea posible.

6.11 Lectura hasta el Fin de Archivo

A modo de ejemplo, el siguiente programa Python abre un archivo y lo copia en otro

```
f = open('origen.txt')
g = open('destino.txt', 'w')
for linea in f:
    g.write(linea)
```

```
g.close()  
f.close()
```

Lo interesante aquí es el ciclo "*for linea in f*". Esta es una forma de recorrer un archivo de texto, obteniendo una línea cada vez.

El archivo se puede leer con *f.readLine()* que nos da una línea cada vez, incluyendo el retorno de carro \n al final. Cuando se llega al final de archivo devolverá una línea vacía. Una línea en blanco en medio del archivo nos sería devuelta como un "\n", no como una línea vacía "". El siguiente ejemplo hace la copia del archivo leyendo con *readline()* y en un ciclo hasta fin de archivo.

```
f = open('origen.txt')  
g = open('destino.txt', 'w')  
linea = f.readline()  
while linea != '':  
    g.write(linea)  
    linea = f.readline()  
g.close()  
f.close()
```

Por ejemplo, para mostrar todas las líneas de un archivo, precedidas por el número de línea, podemos hacerlo como en el Código 6-4.

Código 6-4 numera_lineas.py: Imprime las líneas de un archivo con su número

```
archivo = open("archivo.txt")  
i = 1  
for linea in archivo:  
    linea = linea.rstrip("\n")  
    print(" %4d: %s" % (i, linea))  
    i+=1  
archivo.close()
```

El script anterior muestra:

```
1: Luis Pérez, 10856576, 17  
2: María Torres, 12474322, 16  
3: Pedro Castro, 1111112, 12
```

La llamada a *rstrip* es necesaria ya que cada línea que se lee del archivo contiene un fin de línea y con la llamada a *rstrip("\n")* se remueve.

Otra opción para hacer exactamente lo mismo sería utilizar la función de Python *enumerate(secuencia)*. Esta función devuelve un contador por cada uno de los elementos que

se recorren, puede usarse con cualquier tipo de secuencia, incluyendo archivos. La versión equivalente se muestra en el Código 6-5.

Código 6-5 numera_lineas2.py: Imprime las líneas de un archivo con su número

```
arch = open("archivo.txt")
for i, linea in enumerate(arch):
    linea = linea.rstrip("\n")
    print (" %4d: %s" % (i, linea))
arch.close()
```

6.12 Ejercicios de Archivo de Texto

Ejercicio 10. Sondeo de opinión

Durante un sondeo de opinión, se aplicó una encuesta anónima a un conjunto de N personas, donde por cada persona encuestada, se registró en un archivo de nombre *encuesta.txt* lo siguiente para cada encuestado: *Nombre, Género (M=Masculino, F=Femenino), Edad y el número del canal de televisión que más le gusta*. Se conoce cuantas personas fueron encuestadas, esta información se registró en la primera línea del archivo *encuesta.txt*. Desarrolle un programa (ver Código 6-6) que indique el género y el canal favorito de televisión de la persona con la mayor edad registrada.

Código 6-6. Sondeo.py

```
arch = open('encuesta.txt', 'r')

#Lectura de datos
#se lee de la primera línea el número de encuestados
N = int(arch.readline())

#Borrar el contenido inicial de la bandera, asignarle el valor que
# significa no se tiene el primer mayor
asignado = False

for registro in arch:
    # Devuelve una list con los campos del registro
    linea = registro.split(',')
    print (linea)

    # procesar elementos:
    # usar archivo para leer la lista
    nombre = linea[0]      # nombre de la persona
    print(nombre)
    genero = linea[1]       # género M=Masculino, F=Femenino
    print(genero)
    edad = int(linea[2])    # Edad de la Persona
    print(edad)
    canal = int(linea[3])   #Canal favorito
    print(canal)
```

```

# Proceso para determinar el mayor y guardar los datos
# requeridos
if not(asignado):
    mayor = edad
    mgenero = genero
    mcanal = canal
    asignado = True # bandera indica que ya hay primer mayor
elif edad > mayor: # si el dato a procesar excede al mayor
    mayor = edad
    mgenero = genero
    mcanal = canal

# cerrar archivos
arch.close()

#Escritura de resultados
print('Género de la persona con Mayor Edad: ', mgenero)
print('Canal favorito de la persona con Mayor Edad: ', mcanal)

# Fin del programa
input('Pulse una tecla para finalizar... ') # Pausa

```

Ejecución del programa

```

===== RESTART: G:\Ejercicios Python\04 Archivos de
Texto\01Archivos.py =====
5
['Luisa Lane', 'F', ' 18', '36\n']
Luisa Lane
F
18
36
['Alan Brito', 'M', ' 23', '54\n']
Alan Brito
M
23
54
['Asu Lado', 'M', ' 45', '36\n']
Asu Lado
M
45
36
['Alma Naque', 'F', ' 20', '7\n']
Alma Naque
F
20
7
['Armando Torres', 'M', '45', '33']
Armando Torres
M
45
33
Género de la persona con Mayor Edad: M
Canal favorito de la persona con Mayor Edad: 36
Pulse una tecla para finalizar...

```

Ejercicio 11. Definitiva de alumnos.

Dado el archivo *datos.dat* el cual contiene la información de los estudiantes de una sección *Nombre, Cédula y Nota definitiva*, desarrolle un programa (ver Código 6-7) que

genere dos archivos uno con la información de los aprobados y otra con los reprobados de nombres *aprobados.dat* y *reprobados.dat* respectivamente.

Nota: No se conoce el número de estudiantes almacenados.

Código 6-7 DefinitivaAlumnos.py

```
# Apertura de los archivos

# Archivo de lectura
arch1 = open('datos.dat', 'r')
# Archivos a generar
arch2 = open('aprobados.dat', 'w')
arch3 = open('reprobados.dat', 'w')

# Recorrido del archivo
for registro in arch1:
    # Devuelve una lista con los campos del registro
    linea = registro.split(',')
    print(linea)

    #Procesar elementos:
    nombre = linea[0]      # Nombre del estudiante
    print(nombre)
    cedula = linea[1]      # Cédula del estudiante
    print(cedula)
    nota = int(linea[2])   # Nota del estudiante
    print(nota)

    if nota >= 10: # Aprobado
        arch2.write(nombre + ', ' + cedula + '\n')
    else: # Reprobado
        arch3.write(nombre + ', ' + cedula + '\n')

# Cerrar archivos
arch1.close()
arch2.close()
arch3.close()

# Escritura de resultados
```

```
print('Los resultados están en los archivos aprobados y reprobados.dat en el directorio de trabajo')
```

Datos.dat

Luis Perez , 10856576 , 17
Lola Mento , 12474322 , 16
Mario Moronta , 1111112 , 12
Luisana Rojas, 1123423,08
Leo Pereira, 12345678, 07
Esteban Quito, 12321231, 12

Aprobados.dat

Reprobados.dat

Luis Perez , 10856576
Lola Mento , 12474322
Mario Moronta , 1111112
Esteban Quito, 12321231

Luisana Rojas, 1123423
Leo Pereira, 12345678

6.13 Recorrido de un archivo según su estructura o contenido.

6.13.1 Lista sencilla

El recorrido del archivo será en forma secuencial, los archivos presentan una estructura similar en todas sus líneas o registros y se desea recorrer todo el archivo, se puede utilizar las instrucciones del **!Error! No se encuentra el origen de la referencia..**

Considere el siguiente archivo para realizar su lectura e impresión por pantalla.

datos_transito.txt

Maria,1,15,25
Pedro,2,9,50
Juan,2,14,20
Pablo,2,14,30
Moises,2,1,80

Código 6-8. Lectura e impresión de las líneas del archivo datos_transito.txt:

```
# Manejo de archivos
with open("datos_transito.txt", 'r') as arch1:
```

```

# Proceso
for registro in arch1:
    # lectura de datos
    lista = registro.split(',')

    nombre = lista[0]
    genero = int(lista [1])
    hora = int(lista [2])
    velocidad = float(lista [3])

    print('%s, %d, %d, %.2f' % (nombre, genero, hora, velocidad))

```

observe que la estructura ***for registro in arch1:***, realiza el recorrido sobre cada línea del archivo de manera automática, la variable **registro** devuelve el contenido de la línea en la que se encuentra el apuntador de registro como tipo **str**, la función **Split** convierte en lista el contenido de la variable **registro**.

6.13.2 Lectura parcial o total del archivo

Ocasionalmente se puede presentar el caso de la lectura parcial del archivo (ver Código 6-9), en este caso se indica en la primera línea del archivo la cantidad de líneas a procesar. En el archivo **redes.txt** se tiene la información de *NOMBRE, GÉNERO, EDAD Y RED SOCIAL MÁS UTILIZADA[1-4]* y la cantidad de registros a procesar en la primera línea del archivo.

redes.txt

```

9
sergio,M,25,1
rosa,F,30,2
isabella,F,20,4
michelle,F,18,1
alfonzo,M,23,3
pedro,M,19,4
josefina,F,35,3
aurelio,M,40,2
yolanda,F,15,1

```

La instrucción: ***contenido = arch1.readlines()***, realiza la lectura de todo el archivo **redes.txt**, la variable **contenido** es una lista cuyos elementos son cadenas que contienen la información de cada línea del archivo, al final de cada elemento incluye un salto de línea **\n**, en caso de requerir eliminar el carácter de salto de línea, se emplea la función **strip('\n')**. Esto implica que siempre se conoce tanto la cantidad de registros como la cantidad de datos por registro del archivo.

La variable **w** en la instrucción: **w = int(contenido[linea])**, recibe el valor correspondiente a la cantidad de líneas a leer del archivo, dicho valor se debe validar que sea menor a la cantidad total de líneas del archivo; como se encuentra en la primera línea del archivo, el valor de la variable **Linea** es cero.

Código 6-9. Lectura de las n primeras líneas de un archivo

```
with open('redes.txt', 'r') as arch1:  
    contenido = arch1.readlines()  
    linea = 0  
    w = int(contenido[linea])  
    linea += 1  
  
    for _ in range(1, w + 1):  
        lista = contenido[linea].split(',')  
        linea += 1  
  
        nombre = lista[0]  
        genero = lista[1]  
        edad = int(lista[2])  
        red = int(lista[3])  
  
        print('%s, %s, %d, %d' % (nombre, genero, edad, red))
```

La estructura de control **for _ in range(1, w + 1):**, ejecuta **w** iteraciones para leer las líneas del archivo, se utiliza el guion de subrayado “_” en lugar del índice la estructura de control, debido a que dicho índice no será utilizado en el bloque de la estructura.

La variable **Linea** se incrementará su valor cada vez que se pase por una línea del archivo de datos, la instrucción: **lista = contenido[linea].split(',')** devuelve una lista con elementos de tipo **str** del contenido ubicado en la posición **Linea** del archivo.

6.13.3 Lista sencilla con una cantidad conocida de datos a leer por registro.

Otra estructura de archivo ofrece en la primera línea del archivo la cantidad de grupos de elementos a leer por cada línea del archivo (ver Código 6-10), por ejemplo, en el archivo **partidos.txt** por cada país se tiene un registro de los goles a favor y goles en contra, esto significa que cada par de datos ofrece la información correspondiente a cada partido:

partidos.txt

```
3  
Italia, 1, 1, 1, 1, 2, 0  
España, 1, 1, 5, 0, 2, 0  
Portugal, 0 ,1, 3, 2 ,2, 1
```

Francia, 1, 1, 2, 0, 0, 2
Alemania, 2, 0, 2, 1, 2, 1

La estructura general es la siguiente:

Nombre del País, GFpartido1, GCpartido1, GFpartido2, GCpartido2, ... GFpartidoN, GCpartidoN

Para este caso en particular se está tomando como cantidad de partidos 3 (dato de la primera línea), es decir se tienen 6 datos correspondientes a los goles a favor y los goles en contra para cada partido disputado.

El siguiente código realiza la lectura del archivo *partidos.txt* en el cual se tiene en la primera línea la cantidad de partidos realizados por país, esto significa que por cada partido se debe leer dos datos correspondientes a la cantidad de goles a favor y goles en contra respectivamente.

Código 6-10. Leer un archivo con n grupos de datos en cada línea del archivo.

```
# Manejo de Archivos
with open("partidos.txt", "r") as arch1:
    contenido = arch1.readlines()
    linea = 0
    cp = int(contenido[linea]) # cantidad de partidos
    linea += 1 # Posición del siguiente elemento

    # Ciclo de lectura
    for _ in range(1, len(contenido)): # Ciclo que procesa los países

        lista = contenido[linea].split(',')
        linea += 1

        nom = lista[0]
        print(nom)

        for i in range(1, cp + 1):
            gf = int(lista[2 * i - 1])
            gc = int(lista[2 * i])
            print('%d, %d' % (gf, gc))
```

El primer ciclo **for _ in range(1, len(contenido)):**: se encarga de recorrer todo el archivo de datos y el ciclo interno **for i in range(1, cp + 1):**: recorre en cada

Línea la información correspondiente a cada partido, compuesto por dos datos que son los goles a favor y goles en contra del partido.

La variable lista de tipo *list*, contiene en el índice cero el nombre del equipo, en las posiciones impares $2 * i - 1$ los goles a favor y en las posiciones pares $2 * i$ los goles en contra.

6.13.4 La lista externa e interna de tamaño conocido.

Los archivos también pueden estar estructurados de manera anidada, es decir, presentar dos estructuras diferentes internamente, en este caso se presenta el archivo *ordenes.txt* con una cantidad de elementos conocidos a procesar en su primera línea, y de cada orden de compra se tiene la siguiente información: Número de la orden, Nombre del cliente y la cantidad de órdenes del cliente, seguidamente se escribe la lista interna correspondiente a varios renglones de especificaciones: Área a cubrir [m^2], dimensiones de la baldosa deseada (largo y ancho [cm]) y precio por caja de la baldosa deseada.

ordenes.txt

```
2
21102010, Juan Aguilar, 3
25500.22, 10, 20, 12
260.35, 10, 10, 10.3
500.52, 30, 30, 15.9
22102010, Maria Silva, 4
120.36, 10, 10, 10.3
150.26, 20, 25, 20.3
613.21, 10, 20, 36.2
145.20, 12, 15, 25.6
```

El siguiente código (ver Código 6-11) realiza el manejo de la información del archivo mediante dos ciclos *for* que recorren la información de la lista externa y la información de la lista interna con tamaños conocidos (n y m respectivamente).

Código 6-11. Lectura de n (conocido) elementos de una lista externa y m elementos de una lista interna.

```
with open('ordenes.txt','r') as arch1:
    contenido = arch1.readlines()
    n = int(contenido[0]) # cantidad de elementos de la lista externa
```

```

linea = 1 # Posición del primer elemento de la lista externa

for _ in range(1, n + 1): # CICLO QUE PROCESA LAS ORDENES (lista
externa)
    # LECTURA DE LAS ORDENES
    listaext = contenido[linea].split(',')
    linea += 1 # se incrementa el contador de líneas de contenido
    noc = int(listaext[0])
    nombre = listaext[1]
    m = int(listaext[2]) # cantidad de elementos de la lista interna

    print('%d, %s, %d' % (noc, nombre, m))

    for _ in range(m): # CICLO QUE PROCESA LOS RENGLONES (lista interna)
        listaint = contenido[linea].split(',')
        linea += 1 # se incrementa el contador de líneas de contenido
        area = float(listaint[0])
        l = float(listaint[1])
        a = float(listaint[2])
        monto = float(listaint[3])
        print('%.2f, %.2f, %.2f, %.2f' % (area, l, a, monto))

```

6.13.5 Lista externa de tamaño desconocido y lista interna de tamaño conocido.

En caso de no estar incluido el tamaño (n) de la lista externa como se aprecia en el siguiente archivo.

ordenes.txt

21102010, Juan Aguilar, 3
 25500.22, 10, 20, 12
 260.35, 10, 10, 10.3
 500.52, 30, 30, 15.9
 22102010, Maria Silva, 4
 120.36, 10, 10, 10.3
 150.26, 20, 25, 20.3
 613.21, 10, 20, 36.2
 145.20, 12, 15, 25.6

En este caso se puede leer el archivo (ver Código 6-12) utilizando un ciclo *while* condicionado mediante una variable contador denominada *linea* encargada de contar hasta el final del archivo y un ciclo *for* para el recorrido de la lista interna (m es conocido), observe que el contador *linea* se incrementa en ambos ciclos:

Código 6-12. Lectura de n (desconocido) elementos de una lista externa y m elementos de una lista interna

```

with open('ordenes 2.txt','r') as arch1:
    contenido = arch1.readlines()
    # cantidad de elementos de la lista externa desconocida
    linea = 0 # Posición del primer elemento de la lista externa

    while linea < len(contenido): # Ciclo que procesa las ordenes (lista externa)
        # Lectura de las ordenes
        listaext = contenido[linea].split(',')
        linea += 1 # se incrementa el contador de líneas de contenido
        noc = int(listaext[0])
        nombre = listaext[1]
        m = int(listaext[2]) # cantidad de elementos de la lista interna
        print('%d, %s, %d' % (noc, nombre, m))

        for _ in range(m): # CICLO QUE PROCESA LOS RENGLONES (lista interna)
            listaint = contenido[linea].split(',')
            linea += 1 # se incrementa el contador de líneas de contenido
            area = float(listaint[0])
            l = float(listaint[1])
            a = float(listaint[2])
            monto = float(listaint[3])
            print('%.2f, %.2f, %.2f, %.2f' % (area, l, a, monto))

```

6.13.6 Lista externa e interna de tamaño desconocido.

Sí se desconoce el tamaño de la lista externa e interna, pero se dispone de una variable *centinela* en cada línea de la lista interna como se muestra en el siguiente archivo de datos *recepción.txt*. En el mismo se presenta una estructura con dos listas bien demarcadas: en una línea la **Fecha de Recepción del paquete** y en las siguientes líneas **nombre del cliente**, **ciudad destino**, **peso del paquete**, **dimensiones del paquete: Largo, Ancho y Alto del paquete (metros)** y **la centinela**; hasta llegar a la siguiente fecha de recepción del paquete.

recepción.txt

15/02/2018
Lola mento, Pto La Cruz, 3, 0.25, 0.25, 0.5, 0
Alan Brito, San Cristobal, 4.5, 1.5, 1.0, 2.0, 0
Pepe Trueno, Caracas, 25, 1.5, 0.5, 0.25, 1
18/02/2019
Sere Brito, Maracay, 3.5, 1.5, 1.0, 2.0, 0
Lola mento, Caracas, 3, 0.5, 0.5, 1.0, 0
Alan Brito, San Cristobal, 4.5, 1.5, 1.0, 2.0, 0
Pepe Trueno, Pto La Cruz, 25, 1.5, 0.5, 0.25, 1
19/02/2019

Sere Brito, Maracay, 3.5, 1.5, 1.0, 2.0, 0
Lola mento, Pto La Cruz, 3, 0.5, 0.5, 1.0, 0
Alan Brito, Barquisimeto, 4.5, 0.5, 0.25, 0.5, 0
Pepe Trueno, Caracas, 25, 1.5, 0.5, 0.25, 1

El siguiente código (ver Código 6-13) permite procesar este caso.

Código 6-13 Código para leer todas las líneas de la lista externa y de la lista interna del archivo recepción.txt:

```
with open('reception.txt', 'r') as arch1:  
    # registros, contiene toda la información del archivo  
    contenido = arch1.readlines()  
    linea = 0  
  
    while linea < len(contenido):  
        # fecha_recepcion contiene ['15', '02', '2018\n']  
        fecha_recepcion = contenido[linea].split('/')  
        linea += 1  
        cent = 0  
        print(fecha_recepcion[0], '/', fecha_recepcion[1], '/',  
              fecha_recepcion[2].strip('\n'))  
  
        while cent == 0:  
            # registro contiene  
            # ['Lola mento', ' Pto La Cruz', ' 3', ' 0.25', ' 0.25', ' 0.5', ' 0\n']  
            lista_int = contenido[linea].split(',')  
            cliente = lista_int[0]  
            destino = lista_int[1].strip()  
            paquete = [float(lista_int[2]), float(lista_int[3]),  
                      float(lista_int[4]), float(lista_int[5])]  
            cent = int(lista_int[6])  
            linea += 1  
            print(cliente, destino, paquete[0], paquete[1], paquete[2],  
                  paquete[3], cent)
```

Unidad 7

Programación

Modular

La creación de segmentos de código divide el problema en segmentos lo más pequeño posible con el fin de resolver un problema en su totalidad. Esto facilita la revisión y expansión del código además de aprovechar su reutilización y evitar la repetición de instrucciones.

Programación Modular

La programación modular es un paradigma de programación basado en utilizar [funciones](#) o subrutinas, y únicamente tres estructuras de control:

- secuencial: ejecución de una sentencia tras otra.
- selección o condicional: ejecución de una sentencia o conjunto de sentencias, según el valor de una variable booleana.
- iteración (ciclo o bucle): ejecución de una sentencia o conjunto de sentencias, mientras una variable booleana sea verdadera.

Este paradigma se fundamenta en el teorema correspondiente, que establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo estas tres estructuras lógicas o de control.

La estructura de secuencia es la que se da naturalmente en el lenguaje, ya que por defecto las sentencias son ejecutadas en el orden en que aparecen escritas en el programa.

Para las estructuras condicionales o de selección, Python dispone de la sentencia *if*, que puede combinarse con sentencias *elif* y/o *else*.

Para los ciclos o iteraciones existen las estructuras *while* y *for*.

7.1 Ventajas de la programación modular.

Entre las ventajas de la programación modular, cabe citar las siguientes:

- Los programas son más fáciles de entender, pueden ser leídos de forma secuencial y no hay necesidad de tener que rastrear saltos de líneas (GOTO) dentro de los bloques de código para intentar entender la lógica interna.
- La estructura de los programas es clara, puesto que las sentencias están más ligadas o relacionadas entre sí.
- Se optimiza el esfuerzo en las fases de pruebas y depuración. El seguimiento de los fallos o errores del programa (*debugging*), y con él su detección y corrección, se facilita enormemente.
- Se reducen los costos de mantenimiento. Análogamente a la depuración, durante la fase de mantenimiento, modificar o extender los programas resulta más fácil.

- Los programas son más sencillos y más rápidos de confeccionar.
- Se incrementa el rendimiento de los programadores.

7.2 Funciones

En capítulos anteriores hemos aprendido a utilizar funciones. Algunas de ellas están predefinidas (`abs`, `round`, etc.) mientras que otras deben importarse de módulos antes de poder ser usadas (por ejemplo, `sin` y `cos` se importan del módulo `math`). En la siguiente tabla se tiene algunas de estas funciones.

Tabla 7-1. Funciones incorporadas en Python

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

Módulo math	
<code>pi</code>	valor de π (no es función)
<code>sqrt</code>	raíz cuadrada de un número

Módulo random	
<code>randint</code>	número entero aleatorio en el rango dado

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/library/functions.html>).

En este tema aprenderemos a definir nuestras propias funciones. Definiendo nuevas funciones escribiremos instrucciones en Python para hacer cálculos que inicialmente no están incluidos en el lenguaje y, en cierto modo, adaptando el lenguaje de programación al tipo de problema a resolver, enriqueciéndolo para que el programador pueda ejecutar acciones complejas de un modo sencillo: llamando a funciones desde su programa.

Ya el lector ha utilizado módulos, es decir, archivos que contienen funciones y variables de valor predefinido que puede importar a los programas. En este capítulo aprenderemos a crear nuestros propios módulos, de manera que reutilizar nuestras funciones en varios programas resultará extremadamente sencillo: bastará con importarlas.

Suponga que necesita calcular la suma de números enteros de 1 a 10, de 20 a 37 y de 35 a 49. Si crea un programa para agregar estos tres conjuntos de números, su código podría verse así:

```
sum = 0
for i in range(1, 11):
    sum += i
print("La suma desde 1 hasta 10 es ", sum)

sum = 0
for i in range(20, 38):
    sum += i
print("La suma desde 20 hasta 37 es ", sum)

sum = 0
for i in range(35, 50):
    sum += i
print("La suma desde 35 hasta 49 es ", sum)
```

Observe que el código para calcular estas sumas es muy similar, excepto que los números enteros de inicio y final de la suma son diferentes. Sería bueno poder escribir el código común y luego reutilizarlo. Puede hacer esto definiendo una función, que le permite crear un código reutilizable. Por ejemplo, el código anterior se puede simplificar mediante el uso de funciones, de la siguiente manera:

```
def sum(i1, i2):
    result = 0
    for i in range(i1, i2 + 1):
        result += i
    return result

def main():
    print("La suma desde 1 hasta 10 es ", sum(1, 10))
    print("La suma desde 20 hasta 37 es ", sum(20, 37))
    print("La suma desde 35 hasta 49 es ", sum(35, 49))

main() # llamada de la función principal
```

7.3 Funciones definidas por el usuario.

Una función, es la forma de agrupar expresiones y sentencias que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas. Es decir que, al escribir las instrucciones dentro de una función, al ejecutar el programa, las instrucciones contenidas en la función no serán ejecutadas si no se ha hecho una referencia a la función que lo contiene.

7.3.1 Definiendo funciones

En Python, la definición de funciones se realiza mediante la instrucción ***def*** más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre entre los cuales se escriben los parámetros que recibe la función. Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el código que la conforma, irá con sangría de 4 espacios:

```
def nombre_de_la_función(param1, param2, ...):
    ''' DOCSTRING '''
    # aquí se escribe el código de la función
    return valor_devuelto # opcional
```

Por ejemplo, la siguiente función convierte el valor ingresado de pulgadas a centímetros:

```
def pulg_cm(pulg):
    ''' Función que realiza la conversión de pulgadas a centímetros '''
    cm = pulg * 2.54
    return cm
```

7.3.2 Usando funciones

El modo general es:

```
>>> Nombre(parámetro)
```

Ejemplos:

Usando la función *pulg_cm* (función que convierte de pulgadas a centímetros):

```
>>> print(pulg_cm(5))
```

Todas las funciones devuelven un valor. Las que aparentemente no devuelven nada, están devolviendo *None*:

```
def guarda_lista(lista, nombre):
    ''' Función para guardar una lista de datos en un archivo (nombre) '''
    fh = open(nombre, 'w')

    for x in lista:
        fh.write('%s\n'%x)

    fh.close()
    return None
```

Uso de la función:

```
>>> guarda_lista([1,2,3], 'algo.txt')
```

Note que retorna la función anterior.

Los valores deberían retornarse solo vía “*return*”, para cumplir con la llamada “integridad referencial” (esto es, que una función no modifique el resto del programa). Python no obliga al programador a cumplir con dicha propiedad ya que es posible alterar un dato mutable dentro de una función y esta modificación puede ser vista desde fuera de la misma.

Cuando la función, haga un **retorno de datos**, éstos, pueden ser asignados a una variable:

```
def función():
    return 'Hola Mundo'

frase = función()
print (frase)
```

7.3.3 Ámbito de una función

Los valores declarados en una función son reconocidos solamente dentro de la función. Cuando un valor no es encontrado en el ámbito donde se la invoca, se busca en el ámbito inmediatamente anterior.

Se puede usar **global** para declarar variables globales dentro de funciones, aunque su uso no es recomendable. Estas variables globales cambian el contenido de las variables del módulo que contiene a la función.

```

def test():
    z = 10
    print ('valor de z: %s' % z)
    return None

def test2():
    global z
    z = 10
    print ('valor de: %s' % z)
    return None

```

Probando el ámbito de las variables:

```

>>> z = 50
>>> test() # se hace el llamado de la primera función
Valor de z: 10
>>> z
50
>>> test2() # se hace el llamado de la segunda función
Valor de z: 10
>>> z
10

```

La función *test* no modifica el valor externo de la variable *z*, originalmente 50, la segunda función *test2* si modifica el valor de *z* ya que cambia de 50 a 10.

7.4 Los parámetros

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno [7].

```

def función(par1, par2, ...):
    # código de la función

```

Los parámetros, **se indican** entre los paréntesis, **a modo de variables**, a fin de poder utilizarlos como tales, dentro de la misma función. Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, **a modo de variables de ámbito local**. Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder:

```

def mi_función(nombre, apellido):
    nombre_completo = nombre, apellido
    print(nombre_completo)

```

Si quisiéramos acceder a esas variables locales, fuera de la función, se obtiene un error:

```

def mi_función(nombre, apellido):
    nombre_completo = nombre, apellido

```

```
print(nombre_completo)
print(nombre) # Retornará el error: NameError: name 'nombre' is not defined
```

Al llamar a una función, siempre se le deben pasar sus argumentos en el mismo orden en el que los espera. Pero esto puede evitarse, haciendo uso del paso de argumentos como *keywords* (ver más abajo: “Keywords como parámetros”).

7.4.1 Parámetros por omisión

En Python, también es posible, asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre
saludar('Mundo') # Imprime: Hola Mundo
```

PEP 8: Funciones A la definición de una función la deben anteceder dos líneas en blanco.

Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo =.

7.4.2 Keywords como parámetros

En Python, también es posible llamar a una función, pasándole los argumentos esperados, como pares de claves=valor:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre
saludar(mensaje="Buen día", nombre="Juan")
```

En este caso no importa el orden de los parámetros durante el llamado de la función.

7.4.3 Parámetros arbitrarios

Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de *tupla*.

Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco (*):

```
def recorrer_parametros_arbitrarios(parámetro_fijo, *arbitrarios):
    print (parámetro_fijo)

    # Los parámetros arbitrarios se corren como tuplas
    for argumento in arbitrarios:
        print (argumento)

recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1', 'arbitrario 2', 'arbitrario 3')
```

Por ejemplo,

```
>>> def menú(*platos):
...     print('Hoy tenemos: ', end='')
...     for plato in platos:
...         print(plato, end=', ')
...     return
...
>>> menú('pasta', 'pizza', 'ensalada')
Hoy tenemos: pasta, pizza, ensalada,
```

Si una función espera recibir parámetros fijos y arbitrarios, **los arbitrarios siempre deben suceder a los fijos.**

Es posible también, enviar parámetros arbitrarios como pares de clave=valor. En estos casos, al nombre del parámetro deben precederlo dos asteriscos (**):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios,
**kwords):
    print (parametro_fijo)
    for argumento in arbitrarios:
        print (argumento)

    # Los argumentos arbitrarios tipo clave, se recorren como
    # los diccionarios
    for clave in kwords:
        print ('El valor de ', clave, ' es ', kwords[clave])

    recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1',
'arbitrario 2', 'arbitrario 3', clave1='valor uno', clave2='valor
dos')
```

Por ejemplo,

```
>>> def contacto(**info):
...     print('Datos del contacto')
...     for clave, valor in info.items():
...         print(clave, ":", valor)
...     return
...
>>> contacto(Nombre = "Alf", Email = "asalber@ceu.es")
```

```

Datos del contacto
Nombre : Alf
Email : asalber@ceu.es
>>> contacto(Nombre = "Alf", Email = "asalber@ceu.es", Dirección =
"Madrid")
Datos del contacto
Nombre : Alf
Email : asalber@ceu.es
Dirección : Madrid

```

7.4.4 Desempaquetado de parámetros

Puede ocurrir, además, una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla. En este caso, el signo asterisco (*) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función:

```

def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = [1500, 10]
print (calcular(*datos))

```

El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se encuentren disponibles en un diccionario. Aquí, deberán pasarse a la función, precedidos de dos asteriscos (**):

```

def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = {'descuento': 10, 'importe': 1500}
print (calcular(**datos))

```

7.4.5 Parámetros por Valor o por Referencia

Un último factor a tener en cuenta en los parámetros a funciones en cualquier lenguaje de programación es su modificación en tiempo de ejecución. Tradicionalmente los pasos por parámetro funcionan por valor o por referencia. En el primer caso los argumentos a una función no se modifican al salir de ella (en caso de que el código interno los altere). Técnicamente, se debe al hecho de que realmente no pasamos a la función la variable en cuestión, sino una copia local a la función que es eliminada al acabar su ejecución. En el caso del paso por referencia se pasa a la función un puntero al objeto (en el caso de lenguaje C) o

simplemente una referencia (en lenguajes de alto nivel) que permite su indirección. De este modo, las modificaciones a los parámetros que se hacen dentro de la función se ven reflejadas en el exterior una vez la función termina. En Python los pasos por parámetros son en general por referencia, la excepción la conforman los tipos de datos básicos o inmutables (enteros, flotantes, ..., tuplas), que se pasan por valor. A continuación, se muestra un ejemplo de este hecho (Ejemplo 7-1).

Ejemplo 7-1. Ejemplo de paso por valor y referencia en Python

```
# Ejemplos de definiciones y llamadas de funciones
# función para verificar la modificabilidad de los parámetros

def persistenciaParametros(parametro1, parametro2):
    parametro1 = parametro1 + 5
    parametro2[1] = 6
    print (parametro1)
    print (parametro2)

numeros = [1 ,2 ,3 ,4]
valor_inmutable = 2
persistenciaParametros(valor_inmutable, numeros)
print(numeros)
print(valor_inmutable)
```

Como se puede observar a partir de la ejecución del ejemplo, la lista *numeros* es modificada en el código de la función, mientras que el valor entero de la variable *valor_inmutable* modificado dentro de la función, pierde esta modificación al devolver el control al programa principal.

7.5 Sobre la finalidad de las funciones

Una función, puede tener cualquier tipo de algoritmo y cualquier cantidad de ellos y, utilizar cualquiera de las características vistas hasta ahora. No obstante, **una buena práctica, indica que la finalidad de una función, debe ser realizar una única acción, reutilizable y, por lo tanto, tan genérica como sea posible.**

7.6 Consideraciones prácticas.

Uso de programa principal o no en los códigos siguientes los resultados son idénticos:

```
def suma(a, b):
    return a+b
res = suma(3, 5)
```

```
def suma(a, b):
    return a+b
if __name__ == "__main__":
```

```
print(res)
```

```
res = suma(3, 5)  
print(res)
```

Si se guarda la función en un archivo (por ejemplo `suma.py`) *el código a la derecha permite importar el archivo `suma.py` como un módulo (sin que se ejecute el programa principal)*

7.7 Documentación de funciones

Una práctica muy recomendable cuando se define una función es describir lo que la función hace en un comentario.

En Python esto se hace con un **docstring** que es un tipo de comentario especial se hace en la línea siguiente al encabezado de la función entre tres comillas simples "" o dobles """.

Después se puede acceder a la documentación de la función con la función `help(<nombre-función>)`.

```
>>> def area_triangulo(base, altura):  
...     """Función que calcula el área de un triángulo.  
...  
...     Parámetros:  
...         - base: Un número real con la base del triángulo.  
...         - altura: Un número real con la altura del triángulo.  
...     Salida:  
...         Un número real con el área del triángulo de base y altura  
especificadas.  
...     """  
...     return base * altura / 2  
...  
>>> help(area_triangulo)  
area_triangulo(base, altura)  
    Función que calcula el área de un triángulo.  
  
Parámetros:  
    - base: Un número real con la base del triángulo.  
    - altura: Un número real con la altura del triángulo.  
Salida:  
    Un número real con el área del triángulo de base y altura  
especificadas.
```

7.8 Uso de DOCTEST para probar funciones.

Existe en Python el módulo *doctest* que incluye la función *testmod*. Esta función busca fragmentos de texto en el *docstring* como las sesiones interactivas de Python, y luego ejecuta esas sesiones para verificar que funcionan exactamente como se muestra [3]. Por ejemplo, la función *ganancia_dB()* se puede ejecutar varias veces en la consola de Python, o se puede prever qué valores debe dar:

```
>>> ganancia_dB(1,1000)
60.0
>>> ganancia_dB(10,1000)
40.0
>>> ganancia_dB(1,1)
0.0
>>> ganancia_dB(10,1)
-20.0
```

La función incluye el string de prueba (*doctest*) en el *docstring* y la llamada a la función **testmod**.

```
from math import log10

def ganancia_dB(x, y):
    """Calcula ganancia en dB: 20log(y/x)
    >>> ganancia_dB(1,1000)
    60.0
    >>> ganancia_dB(10,1000)
    40.0
    >>> ganancia_dB(1,1)
    0.0
    >>> ganancia_dB(10,1)
    -20.0
    """
    return 20*log10(y/x)

import doctest
doctest.testmod(verbose=True)
```

Ejecutando el programa se escribe el resultado de evaluar la función con la sesión interactiva incluida en el *docstring*:

```
Trying:
ganancia_dB(1,1000)
Expecting:
60.0
ok
Trying:
ganancia_dB(10,1000)
Expecting:
40.0
ok
Trying:
ganancia_dB(1,1)
Expecting:
```

```

0.0
ok
Trying:
ganancia_db(10,1)
Expecting:
-20.0
ok
1 items had no tests:
__main__
1 items passed all tests:
4 tests in __main__.ganancia_db
4 tests in 2 items.
4 passed and 0 failed.
Test passed.

```

Ejemplo de función con error en el diseño (- en lugar de +):

```

from math import log10

def ganancia_db(x, y):
    """Calcula ganancia en dB: 20log(y/x)
    >>> ganancia_db(1,1000)
    60.0
    >>> ganancia_db(10,1000)
    40.0
    >>> ganancia_db(1,1)
    0.0
    >>> ganancia_db(10,1)
    -20.0
    """
    return 20*log10(y/x)

import doctest
doctest.testmod(verbose=True)

-1
*****
** File "J:/pruebaDoctest2.py", line 12, in __main__.suma
Failed example:
suma(1, 2)
Expected:
3
Got:
-1
*****
** File "J:/pruebaDoctest2.py", line 14, in __main__.suma
Failed example:
suma(10, 10)
Expected:
20
Got:
0
*****
1 items had failures:
2 of 2 in __main__.suma
***Test Failed*** 2 failures.
>>>

```

7.9 Recomendaciones de Escritura del Código.



1. Al inicio de cada función, agregar brevemente un comentario que explique el comportamiento general de la función.
2. Definir el nombre de funciones de acuerdo a la acción que realiza, por lo general es un verbo.
3. Definir los nombres de las variables y constantes locales de acuerdo al contenido que almacenarán.
4. Comentar cuando sea justo y necesario, usar los comentarios dentro de las funciones para describir las variables (sólo cuando su utilidad sea potencialmente dudosa) y cuando existan bloques de código difíciles de entender a primera vista; el exceso de comentarios vuelve ilegible el código.
5. Definir variables locales al inicio de la implementación de cada función, como un bloque de código bien separado del bloque que contenga las instrucciones ejecutables, esta separación puede consistir en una línea en blanco, o bien un comentario que denote la utilidad de cada bloque.
6. Las secciones deseadas en el programa son:
 1. Encabezado con datos del equipo, colegio y breve descripción del programa (no más de 5 líneas), versionado del programa.
 2. Definición de librerías utilizadas
 3. Declaración de constantes
 4. Declaración de variables globales
 5. Declaración de funciones
 6. Declaración bloque de inicialización si existiera
 7. Declaración de bloque principal
 8. Cierre del programa al salir del bloque principal si existiera

7.10 Ejercicios Resueltos

Ejercicio 7-1: Cálculo del IMC.

```
def imc():

    print("CÁLCULO DEL ÍNDICE DE MASA CORPORAL (IMC)")
    peso = float(input("¿Cuánto kg pesa? "))
    altura = float(input("¿Cuánto mide en metros? "))
    imc = peso / altura**2
    print(f"Su IMC es {round(imc, 1)}")
    print(
        "Un IMC muy alto indica obesidad. Los valores normales de imc están
entre 20 y 25,"
    )
```

```

    print(
        "pero esos límites dependen de la edad, del sexo, de la constitución
física, etc."
    )

if __name__ == "__main__":
    imc()

```

Ejercicio 7-2: Calculadora básica.

```

def add(x,y):
    return x+y

def subtract(x,y):
    return x-y

def multiply(x,y):
    return x*y

def divide(x,y):
    return x/y

x = 8
y = 4

print ('%d + %d = %d' % (x, y, add(x, y)))
print ('%d - %d = %d' % (x, y, subtract(x, y)))
print ('%d * %d = %d' % (x, y, multiply(x, y)))
print ('%d / %d = %d' % (x, y, divide(x, y)))

```

Ejercicio 7-3: Operaciones con Fracciones.

El departamento de matemáticas pretende corregir un examen en línea de operaciones con fracciones, que se le aplicará a un grupo de estudiantes del curso introductorio. Cuando un estudiante realiza la evaluación del tema, se crea el archivo "examen.txt", en él se almacena la siguiente información: en la primera línea del archivo se encuentra la identificación del estudiante (Nombre y cédula) y en líneas separadas se guarda, Tipo de operación a realizar (1 = Suma, 2 = Resta, 3 = Multiplicación y 4 = división), las dos fracciones a las que se les va a realizar la operación y la respuesta al ejercicio dada por el estudiante, en la forma de fracción mixta. Las fracciones que se van a utilizar se generan aleatoriamente. Además, la evaluación consiste en resolver cinco (5) operaciones, con un puntaje de 4 puntos c/u.

Elabore un programa que dado el archivo "examen.txt", el cual contiene la evaluación realizada por un estudiante, procese la información con el fin que determine e imprima por pantalla, para cada operación evaluada: la representación de la operación, la respuesta correcta y un mensaje que indique respondió bien o no el estudiante,

Además, al final se debe mostrar el puntaje total obtenido por el estudiante en el examen presentado, tomando en cuenta lo indicado anteriormente.

Requerimientos

- Desarrolle un subprograma que reciba dos valores enteros A y B; y retorne el cociente entero y el residuo de la división de A entre B. Ejemplo, Si A=7 y B=3 => Cociente entero es 2 y el residuo entero es 1.

- Desarrolle un subprograma que reciba una fracción de la forma (a/b) y retorne tres valores (c, r, b) que representan una fracción mixta de la forma (c a/b), donde:

sí $a \leq b$, c es Cero (0) y r es el valor original de A y

sí $a > b$, c es el cociente entero de A/B y R es el residuo entero de a/b

- Desarrolle un subprograma que reciba dos fracciones la forma (a/b) y (c/d), retorne la suma de las dos fracciones, representando la solución en forma de fracción, sabiendo que la fracción resultante es:

$$\text{num / den} = a*d + b*c / (b*d)$$

- Desarrolle un subprograma que reciba dos fracciones la forma (a/b) y (c/d), retorne la multiplicación de la primera fracción por la segunda, sabiendo que la fracción resultante es:

$$\text{num / den} = a*c / (b*d)$$

- Desarrolle un subprograma que reciba dos fracciones la forma (a/b) y (c/d), retorne la división de la primera fracción entre la segunda, sabiendo que la fracción resultante es:

$$\text{num / den} = a*d / (b*c)$$

- Desarrolle un subprograma que reciba dos fracciones mixtas y devuelva si las fracciones son iguales o no.

- Un subprograma que imprima una fracción por pantalla de la forma (a/b).

Ejemplo de los archivos de entrada y salida por pantalla:

1, 8, 2, 3, 4, 4, 6, 8

4, 3, 9, 6, 4, 1, 2, 9

3, 5, 9, 1, 1, 0, 5, 9

2, 6, 3, 4, 8, 1, 12, 24

4, 10, 5, 4, 3, 1, 10, 20

Pantalla de salida

$(8/ 2) + (3/ 4) = 4(6/ 8)$ Correcto

$(3/ 9) / (6/ 4) = 0(12/54)$ Incorrecta

$(5/ 9) * (1/ 1) = 0(5/ 9)$ Correcto

$(6/ 3) - (4/ 8) = 1(12/24)$ Correcto

$(10/ 5) / (4/ 3) = 1(10/20)$ Correcto

Puntaje obtenido = 16 puntos.

Código 7-1. Fracciones.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 19 08:19:29 2022
Solución del ejercicio de operaciones con fracciones
@author: Prof. Alejandro Bolívar
Fecha: 19-04-2022
"""

# subprograma de lectura de los datos de una vuelta
def leer(registro):
    linea = registro.split(',')
    op = int(linea[0])
    num1 = int(linea[1])
    den1 = int(linea[2])
    num2 = int(linea[3])
    den2 = int(linea[4])
    cocr = int(linea[5])
    numr = int(linea[6])
    denr = int(linea[7])
    return op, num1, den1, num2, den2, cocr, numr, denr

# Procedimiento que divide dos números y regresa el cociente y residuo de la
# división
def divisionentera(a, b):
    coc = a // b
    res = a % b
    return coc, res

# Procedimiento que forma una fracción mixta a partir de una fracción
def fraccionmixta(a, b):
    if a <= b: # fracción propia
        c = 0
```

```

        r = a
    else:      # fracción impropia
        c, r = divisionentera(a, b)
    return r, c, b

# Procedimiento que suma dos fracciones
def sumafraccion(a, b, c, d):
    num = a * d + b * c
    den = b * d
    return num, den

# Procedimiento que Resta dos fracciones
def restafraccion(a, b, c, d):
    num = a * d - b * c
    den = b * d
    return num, den

# Procedimiento que Multiplica dos fracciones
def multiplicafraccion(a,b,c,d):
    num = a * c
    den = b * d
    return num, den

# Procedimiento que Divide dos fracciones
def dividafraccion(a, b, c, d):
    num = a * d
    den = b * c
    return num, den

# Función que compara dos fracciones mixtas y devuelve si son iguales o no
def iguales(c1, r1, b1, c2, r2, b2):
    return (c1 == c2) and (r1 == r2) and (b1 == b2)

# procedimiento que imprime por pantalla una fracción
def imprimirfraccion(num, den):
    print(" (%d / %d) " % (num, den), end="")

def main():

    # Que tengo
    num1: int
    den1: int  # Fracción 1
    num2: int
    den2: int  # Fracción 2
    op: int
    cocr: int  # Fracción mixta resultante por el estudiante
    numr: int
    denr: int
    # Que Quiero
    punt: int = 0 # Puntaje obtenido, Acumulador
    # Variables auxiliares

```

```

coc: int
num: int
den: int # Fracción mixta correcta
i: int

arch = open("examen.txt", "r")
punt = 0
# Ciclo de lectura
for registro in arch:
    op, num1, den1, num2, den2, cocr, numr, denr = leer(registro)

    # Determinación del resultado correcto a la operación a realizar e
    # Impresión del resultado
    imprimirfraccion(num1, den1)

    if op == 1:
        num, den = sumafraccion(num1, den1, num2, den2)
        print("+", end="")
    elif op == 2:
        num, den = restafraccion(num1, den1, num2, den2)
        print("-", end="")
    elif op == 3:
        num, den = multiplicafraccion(num1, den1, num2, den2)
        print("*", end="")
    else:
        num, den = dividafraccion(num1, den1, num2, den2)
        print("/", end="")

    imprimirfraccion(num2, den2)

    # Determinación de la fracción mixta resultante
    num, coc, den = fraccionmixta(num, den)
    print("= %d" % coc, end="")
    imprimirfraccion(num, den)

    if iguales(cocr, numr, denr, coc, num, den):
        print(" Correcto")
        punt += 4
    else:
        print(" Incorrecta")

    # Impresión del puntaje obtenido
    print ("Puntaje Obtenido = %d Ptos" % punt)
arch.close()

# Mensaje al usuario
input("Pulse una tecla para finalizar")

if __name__ == "__main__":
    main()

```

7.11 Ejercicios propuestos de funciones

- 1) Valide y devuelva un valor x entre a y b
- 2) Compruebe que tres valores (x, y, z) son diferentes.
- 3) Devolver el valor intermedio de tres valores (x, y, z) diferentes.
- 4) Comprobar si un valor x es par, impar o cero.
- 5) Compruebe sí un número x es múltiplo de un valor k.
- 6) Devuelva la cantidad de dígitos de x.
- 7) Cuántos dígitos impares tiene x
- 8) Dados tres valores (x, y, z) verifique sí están ordenados ascendente, descendente o desordenados.
- 9) Cuál es el mayor de tres valores (x, y, z).
- 10) Determine si dos valores (a, b) son iguales si $|a-b| \leq \text{tolerancia}$.
- 11) Leer por teclado n dígitos (0..9) y devolver el número formado (los dígitos van de izquierda a derecha).
- 12) Dado un número entero devuelva el dígito ubicado en la n-ésima posición a partir de la derecha.
- 13) Dados tres valores (x, y, z) devolverlos ordenados.
- 14) Leer n valores enteros desde un archivo e indique cuáles valores faltan en un rango comprendido desde el menor hasta el máximo valor.
- 15) Dados dos archivos "a.txt" y "b.txt" que contienen números enteros, devuelva un archivo "unión.txt" con los números de ambos archivos, otro archivo "intersección.txt" con los números comunes de ambos archivos y otro "diferencia_simetrica.txt" con los números no comunes en ambos archivos.
- 16) Dado un archivo "datos.txt" que contiene una lista de números enteros, verifica si tienen un orden ascendente, descendente o desordenado.
- 17) Los números perfectos y números amigos
 - a. Escribir una función que devuelva la suma de todos los divisores de un número n , sin incluirlo.
 - b. Usando la función anterior, escribir una función que imprima los primeros m números tales que la suma de sus divisores sea igual a sí mismo (es decir los primeros m números *perfectos*).
 - c. Usando la primera función, escribir una función que imprima las primeras m parejas de números (a,b) , tales que la suma de los divisores de a es igual a b y la suma de los divisores de b es igual a a (es decir las primeras m parejas de números *amigos*).

- d. Proponer optimizaciones a las funciones anteriores para disminuir el tiempo de ejecución.

18) Escribir una función que reciba dos números como parámetros, y devuelva cuántos múltiplos del primero hay, que sean menores que el segundo.

- Implementarla utilizando un ciclo `for`, desde el primer número hasta el segundo.
- Implementarla utilizando un ciclo `while`, que multiplique el primer número hasta que sea mayor que el segundo.
- Comparar ambas implementaciones: ¿Cuál es más clara? ¿Cuál realiza menos operaciones?

19) **Bisiesto.** Escribe un programa que incluya la función `bisiesto(n)`, que indique si un año es o no es un año bisiesto (devuelve valor Booleano). Un año bisiesto tiene 366 días. Después de la reforma gregoriana, los años bisiestos son los múltiples de cuatro que no terminan con dos ceros, y también los años que terminan con dos ceros que, después de eliminar estos dos ceros, son divisibles por cuatro. Así, 1800 y 1900, aunque eran múltiplos de cuatro, no eran años bisiestos; Por el contrario, 2000 fue un año bisiesto. Dicho en formalismo lógico: un año bisiesto es divisible por 400 o bien divisible por 4 exceptuando los divisibles por 100. Ejemplo:

```
>>> bisiesto(2000)
True
>>> bisiesto(1900)
False
```

20) **Coordenadas cartesianas.** Diseña una función `cartesianas(mod, ang)` que reciba el valor del módulo y de su ángulo en grados y devuelva sus coordenadas cartesianas (x, y) ajustados a 4 decimales usando la función interna `round(value; digits)`.
Ejemplo:

```
>>> from math import sin, cos, pi
>>> cartesianas(1,pi/3)
(0.5, 0.866)
>>> cartesianas(1.4142,pi/4)
(1.0, 1.0)
```

21) **Impuestos.** Una empresa desea calcular la estimación del importe de impuestos que los empleados deben pagar. Los ingresos inferiores a 8.000 euros no están sujetos a impuestos; los comprendidos entre 8.000 euros y 20.000 euros, lo están al 18%; los comprendidos entre 20.000 euros y 35.000 euros, están sujetos al 27% y los superiores a 35.000 euros, lo están al 38%. Diseña una función `impuestos(x)` que calcule los impuestos correspondientes a los ingresos x .

Entrada: Un valor x de ingresos.

Salida: Los impuestos correspondientes a x con 2 decimales.

Ejemplos:

```
>>> impuestos(15000)
```

2700.0

```
>>> impuestos(55346)
```

21031.48

22) **IMC.** Diseña una función imc(p,h) que reciba el peso p y la altura h de una persona, calcule el índice de masa corporal de una persona ($IMC = \frac{peso[kg]}{altura^2[m]}$) y retorne el estado en el que se encuentra esa persona en función del valor de IMC:

- <16: criterio de ingreso hospitalario
- de 16 a 17: infrapeso
- de 17 a 18: bajo peso
- de 18 a 25: saludable
- de 25 a 30: sobrepeso
- de 30 a 35: sobrepeso crónico
- de 35 a 40: obesidad premórbida
- >40: obesidad mórbida

Entrada: Peso p y altura h.

Salida: Estado.

Ejemplos:

```
>>> imc(1.65,68)
```

'saludable'

23) **Riesgo Cardiaco.** Diseña una función que reciba el peso p en kg, la altura h en metros, el valor del colesterol de lipoproteínas de baja densidad, LDL en mg/dl y si es fumador (o no, variable booleana). La función devuelve si está en riesgo de insuficiencia cardiaca si se cumple que: el índice de masa corporal ($IMC = \frac{peso[kg]}{altura^2[m]}$) es mayor que 35 y que el colesterol (LDL) está por debajo de 71 mg/dl o por encima de 300 mg/dl, o si es fumador. Ejemplo, riesgoCardio(p,h,LDL,fuma):

```
>>> riesgoCardio(95,1.64,310,False)
```

True

```
>>> riesgoCardio(90,1.64,310,False)
```

False

24) **Número perfecto.** Diseñe una función que encuentre el primer número perfecto mayor que 28 (o un número n dado). Un número es perfecto si coincide con la suma de sus divisores (excepto él mismo). Por ejemplo, 28 es perfecto ya que $28 = 1 + 2 + 4 + 7 + 14$.

```
>>> perfecto(28)
```

496

```
>>> perfecto(500)
```

8128

25) **Números armónicos.** Diseña una función `harmon(n)` que lea un número `n` e imprima el `n`-ésimo número armónico, definido como $H_n = 1/1 + 1/2 + \dots + 1/n$.

Entrada: un número natural `n`.

Salida: H_n con 4 dígitos después del punto decimal.

```
>>> harmon(2)  
1.5000  
>>> harmon(7)  
2.5929
```

26) **Es primo?** Diseña una función `es_primo(n)` que reciba un natural `n` y devuelva `True` si el número es primo o `False` en caso contrario. Se asume que `n > 1`.

```
>>> es_primo(15)  
False  
>>> es_primo(349)  
True
```

27) **Cuántos primos.** Diseña una función `cuantos_primos(n)` que devuelva el número de primos que existen en el intervalo $(1, n)$ (se excluyen). Usar la función `es_primo`.

```
>>> cuantos_primos(15)
```

6

28) **Seno por serie de Taylor.** Diseña una función `seno_t(x, error=1e-9)` que reciba un valor en grados, lo transforme en radianes y que aproxime el valor de su seno, según su desarrollo en serie de Taylor. La función puede tener el valor del error por omisión, que sirve para terminar la iteración cuando un término sea menor que el error.

$$\text{seno}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

```
>> seno_t(45)  
0.7071067811796194
```

29) **Seno por serie de Taylor v2.** Diseña una función `seno_t2(x, error=1e-9)` que reciba un valor en grados, lo transforme en radianes y que aproxime el valor de su seno, según su desarrollo en serie de Taylor. No utilizar ni las funciones factorial de `math` ni elevar x^n , usar los términos anteriores para reducir el tiempo de cálculo.

7.12 Guía de estilo - PEP8

En Python existen las denominadas PEP's (*Python Enhancement Proposals*), en concreto la PEP 8 hace referencia a las convenciones de estilo de programación en Python.

Entre las convenciones principales, destacan:

- Usar 4 espacios para indentar.
- Tamaños de línea con un máximo de 79 caracteres.
- Las funciones y las clases se deben separar con dos líneas en blanco, mientras que los métodos de clase solo con uno.
- Los *import* deben de estar separados uno en cada línea. Se permite: `from math import sin, pi`
- Las sentencias *import* deben de estar siempre en la parte superior del archivo agrupadas de la siguiente manera:
 - Librería estándar
 - Librerías de terceros
 - import's de la aplicación local
- Usar espacios alrededor de los operadores aritméticos, a excepción de cuando forman parte de los argumentos de una función.
- No se deben de realizar comentarios obvios.
- No se deben comparar booleanos mediante ==

7.13 Resumen.

Una función puede tener ninguno, uno o más parámetros y que los mismos pueden o no tener valores por defecto. En el caso de tener más de uno, se separan por comas tanto en la declaración de la función como en la invocación.

Es altamente recomendable documentar cada función que se escribe, para poder saber qué parámetros recibe, qué devuelve y qué hace sin necesidad de leer el código.

Las funciones pueden imprimir mensajes para comunicarlos al usuario, y/o devolver valores. Cuando una función realice un cálculo o una operación con sus parámetros, es recomendable que devuelva el resultado en lugar de imprimirllo, permitiendo realizar otras operaciones con él.

No es posible acceder a las variables definidas dentro de una función desde el programa principal, si se quiere utilizar algún valor calculado en la función, será necesario devolverlo.

Si una función no devuelve nada, por más que se la asigne a una variable, no quedará ningún valor asociado a esa variable.

Unidad 8

La programación como técnica de resolución de problemas numéricos y/o de distintas áreas de ingeniería

En este tema se utiliza la biblioteca Sympy para la realización de cálculos simbólicos, se resuelven ejercicios de diferentes temas asociados al Cálculo Matemático.

Cálculo simbólico con SymPy

SymPy es una librería de Python desarrollada para resolver problemas de matemáticas simbólicas. Existen diversos softwares comerciales que realizan estas tareas: Maple, Mathematica, MATLAB, entre otros, pero requieren una licencia de uso que puede resultar poco accesible en algunos casos. En cambio, SymPy se distribuye bajo licencia BSD, que en resumen permite el uso libre de la misma.

8.1 Importando SymPy

Para importar SymPy y disponer de todos los módulos y funciones que le componen puede hacerse de diversas formas:

1. Forma tradicional

```
>>> import sympy
```

Es la manera más habitual, se carga toda la librería y se accede a cada una de las funciones mediante la sintaxis:

```
>>> r=sympy.funcion(args)
```

2. Importando funciones seleccionadas

```
>>> from sympy import Symbol,integrate,sin,cos
```

De este modo se importan solamente las funciones que vayan a utilizarse, es recomendable cuando se utilizará un número reducido de las mismas. Proporciona cierta ventaja dado que para acceder a una función no es necesario anteponer el nombre de la librería (sympy), aunque esto mismo represente una desventaja en aquellos casos en los que existen funciones de diferentes librerías con el mismo nombre.

3. Utilizando un alias o seudónimo

```
>>> import sympy as sp
```

Funciona del mismo modo que para el primer caso, con la diferencia que el usuario puede asignarle un nombre más corto o bien más representativo para hacer las llamadas a funciones. Para los ejemplos que se mostrarán en esta entrada se utilizará la segunda forma.

8.2 Declarando una variable simbólica

Para declarar una variable simbólica podemos utilizar la función `Symbol`, para ello primero importamos la función y posteriormente declaramos una variable simbólica "x":

```
>>> from sympy import Symbol
>>> x=Symbol('x')
>>> x
x
>>> x+2
x + 2
```

Como puede verse, una vez se ha declarado la variable simbólica podemos utilizarla para formar expresiones algebraicas de todo tipo. Existe una forma más "simple" de declarar una variable simbólica, para ello habrá de importarse del módulo "abc" la letra correspondiente, por ejemplo:

```
>>> from sympy.abc import x
```

O bien:

```
>>> from sympy.abc import x,y,z
```

Lo anterior en el caso de que se requieran múltiples variables simbólicas.

Sympy permite hacer operaciones analíticas o con símbolos en lugar de con valores numéricos Al igual que en Python existen varios tipos de datos numéricos como enteros (`int`), decimales (`float`) o booleanos (`bool:True`, `False`, etc.), `Sympy` posee tres tipos de datos propios: `Real`, `Rational` e `Integer`, es decir, números reales, racionales y enteros. Esto quiere decir que `Rational(1,2)` representa $\frac{1}{2}$, `Rational(5,2)` a $\frac{5}{2}$, etc. en lugar de 0.5 o 2.5.

```
>>> import sympy as sp
>>> a = sp.Rational(1,2)
>>> a
1/2
>>> a**2
1
>>> sp.Rational(2)**50/sp.Rational(10)**50
1/88817841970012523233890533447265625
```

8.3 Números complejos

Cálculo simbólico con Sympy

Manipulaciones algebraicas

La unidad imaginaria es denotada por `I` en Sympy.

```
In [12]:  
1+1*sp.I  
Out[12]:  
1 + I  
In [13]:  
sp.I**2  
Out[13]:  
-1  
In [14]:  
(x * sp.I + 1)**2  
Out[14]:  
(I*x + 1)**2
```

También existen algunas constantes especiales, como el número e o π sin embargo, éstos se tratan con símbolos y no tienen un valor numérico determinado. Eso quiere decir que no se puede obtener un valor numérico de una operación usando el valor pi de Sympy, como $(1+\pi)$, como lo haríamos con el de Numpy, que es numérico:

```
>>> sp.pi**2  
pi**2  
>>> sp.pi.evalf()  
3.14159265358979  
>>> (sp.pi + sp.exp(1)).evalf()  
5.85987448204884
```

como se ve, sin embargo, se puede usar el método `evalf()` para evaluar una expresión para tener un valor en punto flotante (float).

Para hacer operaciones simbólicas hay que definir explícitamente los símbolos que vamos a usar, que serán en general las variables y otros elementos de nuestras ecuaciones:

```
>>> x = sp.Symbol('x')  
>>> y = sp.Symbol('y')
```

Y ahora ya se puede utilizar:

```
>>> x+y+x-y  
2*x  
>>> (x+y)**2  
(x + y)**2
```

8.4 Manipulaciones algebraicas

8.4.1 Factorizar una expresión algebraica.

Para factorizar una expresión algebraica podemos utilizar la función `factor`, por ejemplo, suponga que se quiere factorizar la expresión (x^2+2x+1) :

```
>>> from sympy import factor,Symbol
>>> x=Symbol('x')
>>> factor(x**2+2*x+1)
(x + 1)**2
```

8.4.2 Expandir una expresión algebraica

Enseguida se muestra un ejemplo de cómo "expandir" o multiplicar dos expresiones algebraicas.

```
>>> from sympy import Symbol,expand
>>> x=Symbol('x')
>>> expand((x+2)*(x-3))
x**2 - x - 6
>>> ((x+y)**2).expand()
2*x*y + x**2 + y**2

>>> expand((x+1)**2)
x2 + 2x + 1

>>> factor(x**2+6*x-16)
(x - 2)(x + 8)
```

Es posible hacer una substitución de variables usando `subs(viejo, nuevo)`:

```
>>> ((x+y)**2).subs(x, 1)
(1 + y)**2
>>> ((x+y)**2).subs(x, y)
4*y**2
```

8.4.3 Simplificación

Usa `simplify` si quieres transformar una expresión en algo más sencillo

```
In [19]: simplify((x+x*y)/x)
Out[19]: 1 + y
```

Simplificación es un término vago, es por ello que existen alternativas más precisas que `simplify`: `powsimp` (simplificación de exponentes), `trigsimp` (para expresiones trigonométricas), `logcombine`, `radsimp`, `together`.

8.5 Operaciones algebraicas

Cálculo simbólico con Sympy Ecuaciones algebraicas

Podemos usar `apart(expr, x)` para hacer una descomposición parcial de fracciones:

```
>>> 1/(x+2)*(x+1)
1
_____
(2 + x)*(1 + x)
>>> sp.apart(1/(x+2)*(x+1), x)
1
_____
1 + x - 2 + x
>>> (x+1)/(x-1)
-(1 + x)
_____
1 - x
>>> sp.apart((x+1)/(x-1), x)
2
1 - -----
1 - x
```

8.6 Ecuaciones algebraicas

También se pueden resolver sistemas de ecuaciones de manera simbólica:

```
>>> # Una ecuación, resolver x
>>> sp.solve(x**4 - 1, x)
[I, 1, -1, -I]
>>> solve(x**2-3*x+2)
[1, 2]

>>> solve(x**3+5)
[-³√5, ³√5/2 - √3i³√5, ³√5/2 + √3i³√5]

>>> solve((x**2+2)>0)
Ix = 0 ∧ -∞ < Rx ∧ Rx < ∞
```

```
# Sistema de dos ecuaciones. Resuelve x e y
>>> sp.solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
{y: 1, x: -3}
```

```
>>> solve([x+y-1,x-y])
{ x : 1/2,  y : 1/2 }

>>> solve([x + y + z - 1, x + y + 2*z - 3 ])
{x : -y - 1,  z : 2}
```

8.7 Cálculo de límites

Sympy puede calcular límites usando la función `limit()` con la siguiente sintaxis: `limit(función, variable, punto)`, lo que calcularía el límite de $f(x)$ cuando variable \rightarrow punto:

```
.. code:: ipython3
x = sp.Symbol("x")
sp.limit(sin(x)/x, x, 0)
1
```

es posible incluso usar límites infinitos:

```
>>> sp.limit(x, x, oo)
oo
>>> sp.limit(1/x, x, oo)
0
>>> sp.limit(x**x, x, 0)
1
```

8.8 Cálculo de derivadas

La función de Sympy para calcular la derivada de cualquier función es `diff(func, var)`. Veamos algunos ejemplos:

```
>>> x = sp.Symbol('x')
>>> diff(sp.sin(x), x)
cos(x)
>>> diff(sp.sin(2*x), x)
2*cos(2*x)
>>> diff(sp.tan(x), x)
1 + tan(x)**2
```

Se puede comprobar que es correcto calculando el límite:

```
>>> dx = sp.Symbol('dx')
>>> sp.limit((sp.tan(x+dx) - sp.tan(x))/dx, dx, 0)
1 + tan(x)**2
```

Cálculo simbólico con Sympy

Expansión de series

También se pueden calcular derivadas de orden superior indicando el orden de la derivada como un tercer parámetro opcional de la función `diff()`:

```
>>> sp.diff(sp.sin(2*x), x, 1)          # Derivada de orden 1
2*cos(2*x)
>>> sp.diff(sp.sin(2*x), x, 2)          # Derivada de orden 2
-4*sin(2*x)
>>> sp.diff(sp.sin(2*x), x, 3)          # Derivada de orden 3
-8*cos(2*x)
```

8.9 Expansión de series

Para la expansión de series se aplica el método `series(var, punto, orden)` a la serie que se desea expandir:

```
>>> from sympy import series
>>> sp.cos(x).series(x, 0, 10)
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)
>>> (1/sp.cos(x)).series(x, 0, 10)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 + O(x**10)
e = 1/(x + y)
s = e.series(x, 0, 5)
pprint(s)
```

La función `pprint` de Sympy imprime el resultado de manera más legible:

$$\frac{1}{y} + \frac{x^4}{y^5} - \frac{x^3}{y^4} + \frac{x^2}{y^3} - \frac{x}{y^2} + O(x^5)$$

Si usamos una `qtconsole` (iniciada con `ipython qtconsole`) e ejecutamos `init_printing()`, las fórmulas se mostrarán con *LATEX*, si está instalado.

8.10 Integración

Las integrales son unos de los conceptos básicos en la formación matemática de un ingeniero, es en términos básicos la operación inversa de la derivación. Pero, además del concepto puramente matemático, las integrales tienen múltiples interpretaciones geométricas y físicas.

En un curso ordinario de cálculo se nos enseñan métodos para resolver de forma analítica funciones que sean integrables. Por ejemplo, todos sabemos que la integral de una función constante será:

$$\int a \, dx = ax + C$$

Y lo sabemos porque nos hemos aprendido reglas básicas de integración y por supuesto a identificar el tipo de función. Actualmente existen paquetes de álgebra simbólica que son capaces de realizar esta tarea: identificar el caso que se tiene y aplicar métodos computacionales, hasta cierto grado complejos, para determinar la solución.

Y claro, SymPy es uno de esos sistemas de álgebra computacional (CAS), en el que solo necesitamos escribir nuestra función a integrar, utilizar por ahí alguna rutina y obtener un resultado rápidamente. Pero claro, para ello debemos aprender mínimamente la sintaxis y eso es justo lo que veremos enseguida.

8.10.1 Integrales simples

Vamos a ver cómo resolver integrales simples indefinidas, si, de esas que vemos en un primer curso. Para resolverlas tendremos que utilizar la función `integrate`. Por ejemplo, se tiene la siguiente función $f(x)=x^2-3x+2$

Como primer paso debemos importar lo que necesitaremos del paquete [SymPy](#):

```
import sympy as sp
from sympy import integrate, init_printing
from sympy.abc import x
init_printing()
```

Del módulo `abc` importamos la variable simbólica `x` e `integrate` para resolver nuestra integral. Ahora, podemos *guardar* la función a integrar en una variable o bien pasarlá directamente como argumento:

```
f = x**2 - 3*x + 2
sp.integrate(f)
```

$$\frac{x^3}{3} - \frac{3x^2}{2} + 2x$$

En este caso no hemos tenido inconvenientes, porque en la expresión a integrar sólo existe una variable simbólica, pero si la expresión tuviese más de una, habría que especificar de manera explícita la variable respecto a la cual se integra, de lo contrario Python nos *mostrará* un error:

```
from sympy.abc import a,b,c
f = a*x**2+b*x+c
sp.integrate(f)
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-3-476839d3c49d> in <module>()
      1 from sympy.abc import a,b,c
      2 f = a*x**2+b*x+c
----> 3 integrate(f)
```

Cálculo simbólico con Sympy

Integración

```
~\Anaconda3\lib\site-packages\sympy\integrals\integrals.py           in
integrate(*args, **kwargs)
    1289     risch = kwargs.pop('risch', None)
    1290     manual = kwargs.pop('manual', None)
-> 1291     integral = Integral(*args, **kwargs)
    1292
    1293     if isinstance(integral, Integral):
~\Anaconda3\lib\site-packages\sympy\integrals\integrals.py in __new__(cls,
function, *symbols, **assumptions)
    73         return function._eval_Integral(*symbols, **assumptions)
    74
--> 75         obj = AddWithLimits.__new__(cls, function, *symbols,
**assumptions)
    76         return obj
    77
~\Anaconda3\lib\site-packages\sympy\concrete\expr_with_limits.py      in
__new__(cls, function, *symbols, **assumptions)
    375             " more than one free symbol, an integration
variable should"
    376             " be supplied explicitly e.g., integrate(f(x,
y), x)"
--> 377             % function)
    378             limits, orientation = [Tuple(s) for s in free], 1
    379
```

ValueError: specify dummy variables for $a*x^{**2} + b*x + c$. If the integrand contains more than one free symbol, an integration variable should be supplied explicitly e.g., `integrate(f(x, y), x)`

Pues eso, si intentamos integrar la función $f(x)=ax^2 + bx + c$ sin especificar la variable de integración, Python nos mandará un error que es bastante sugerente al respecto. Así, lo correcto sería:

```
sp.integrate(f, x)
```

$$\frac{ax^3}{3} + \frac{bx^2}{2} + cx$$

Veamos algunos ejemplos:

```
>>> sp.integrate(6*x**5, x)
x**6
>>> sp.integrate(sp.sin(x), x)
-cos(x)
>>> sp.integrate(sp.log(x), x)
-x + x*log(x)
>>> sp.integrate(2*x + sp.sinh(x), x)
cosh(x) + x**2
```

También con funciones especiales:

```
| >>> sp.integrate(exp(-x**2)*erf(x), x)
```

```
| pi**(1/2)*erf(x)**2/4
```

8.10.2 Integrales definidas

Una integral definida usualmente se utiliza para calcular el área bajo la curva de una función en un intervalo finito. En SymPy, para calcular una integral definida se utiliza la función `integrate`, considerando el hecho que deben adicionarse los límites de evaluación mediante la sintaxis:

```
sp.integrate(fun, (var, a, b))
```

Donde `fun` es la función, `var` la variable respecto a la cual se integra, `a` el límite inferior y `b` el límite superior.

Para ejemplificar vamos a resolver la siguiente integral definida:

$$\int_0^{\frac{x}{2}} \cos x \, dx$$

```
from sympy import cos,pi
sp.integrate(cos(x), (x,0,pi/2.0))
1
```

Otro ejemplo:

$$\int_0^5 x \, dx$$

```
sp.integrate(x, (x,0,5))
25
2
sp.integrate(x**3, (x, -1, 1))
0
sp.integrate(sin(x), (x, 0, pi/2))
1
sp.integrate(cos(x), (x, -pi/2, pi/2))
2
```

Y también integrales impropias:

```
sp.integrate(exp(-x), (x, 0, oo))
1
sp.integrate(log(x), (x, 0, 1))
-1
```

Algunas integrales definidas complejas es necesario definirlas como objeto Integral() y luego evaluarlas con el método evalf():

```
from sympy import Integral, evalf
integ = sp.Integral(sin(x)**2/x**2, (x, 0, oo))
integ.evalf()
1.6
```

8.10.3 Integrales múltiples

Ahora vamos a resolver integrales dobles (la sintaxis/metodología de resolución que se revisará aplica para cualquier integral múltiple). Por ejemplo, vamos a resolver la integral dada por:

$$\int_a^b \int_c^d dy dx$$

Recuerde que este tipo de integrales múltiples se resuelven de forma *iterada*, yendo *de dentro hacia afuera*, es decir, para la integral anterior se procedería:

$$I_1 = \int_c^d dy \rightarrow I = \int_a^b I_1 dx$$

En Python/SymPy se hace exactamente lo mismo:

```
from sympy.abc import x,y,z,a,b,c,d
from sympy import simplify
I1 = sp.integrate(1, (y,c,d))
sp.simplify(sp.integrate(I1, (x,a,b) ) )
(a-b)(c-d)

>>> fxy = cos(x)+sin(y)
>>> integrate(fxy,(x,0,pi/2),(y,0,pi))
2π

>>> fxy2 = x**2 + y**2
>>> integrate(fxy2,(y,0,2),(x,0,1))
10
3
```

8.11 Ecuaciones diferenciales

Sympy es capaz de resolver (algunas) ecuaciones diferenciales ordinarias. sympy.ode.dsolve funciona de la siguiente forma

```
In [4]: f(x).diff(x, x) + f(x)
Out[4]:

$$\frac{d^2}{dx^2}(f(x)) + f(x)$$


In [5]: dsolve(f(x).diff(x, x) + f(x), f(x))
Out[5]: C_1\sin(x) + C_2\cos(x)
```

Se pueden usar argumentos en las keywords para ayudar a encontrar el mejor sistema de resolución posible. Por ejemplo, si a priori conoces que estás tratando con ecuaciones separables, puedes usar la palabra clave (keyword) `hint='separable'` para forzar a `dsolve` a que lo resuelva como una ecuación separable.

```
In [6]: dsolve(sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x),
f(x), hint='separable')
Out[6]: -log(1 - sin(f(x))**2)/2 == C1 + log(1 - sin(x)**2)/2
```

Otro ejemplo:

```
>>> ec_dif = f(x).diff(x) + k*f(x) # Ecuación diferencial
>>> dsolve(ec_dif, f(x)) # Resolver respecto a f(x)

$$f(x) = C_1 e^{-kx}$$


>>> ec_dif2 = f(x).diff(x,2) + f(x)
>>> dsolve(ec_dif2, f(x)) # Resolver respecto a f(x)

$$f(x) = C_1 \sin(x) + C_2 \cos(x)$$

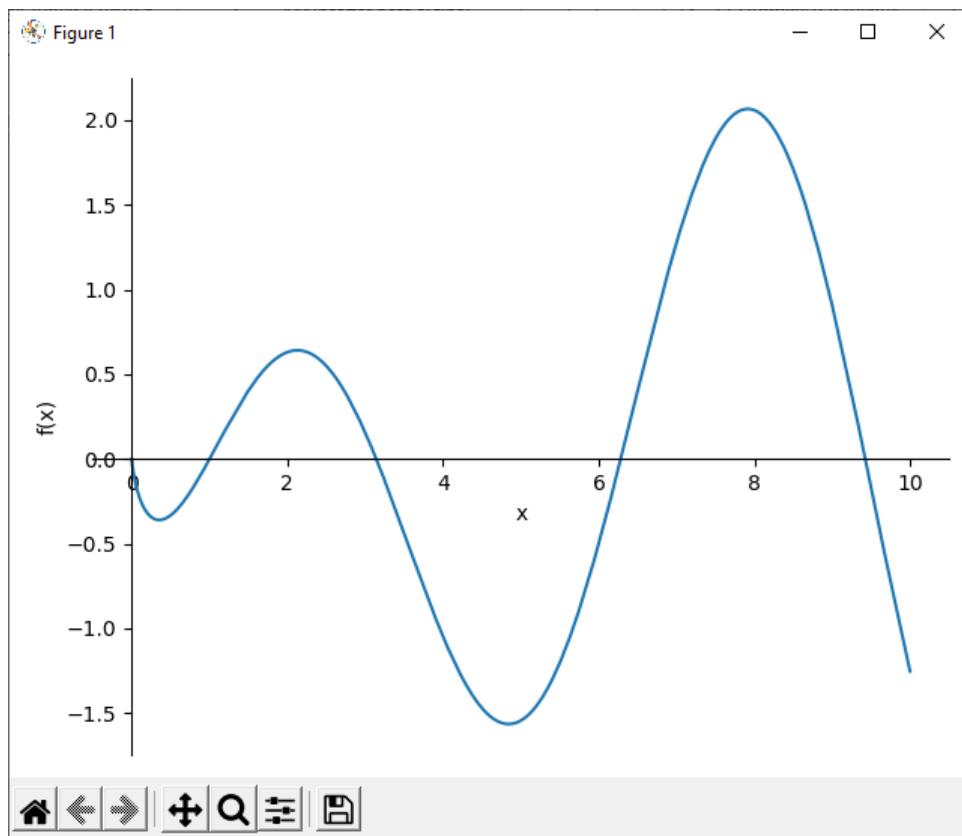
```

8.12 Gráficos con Sympy

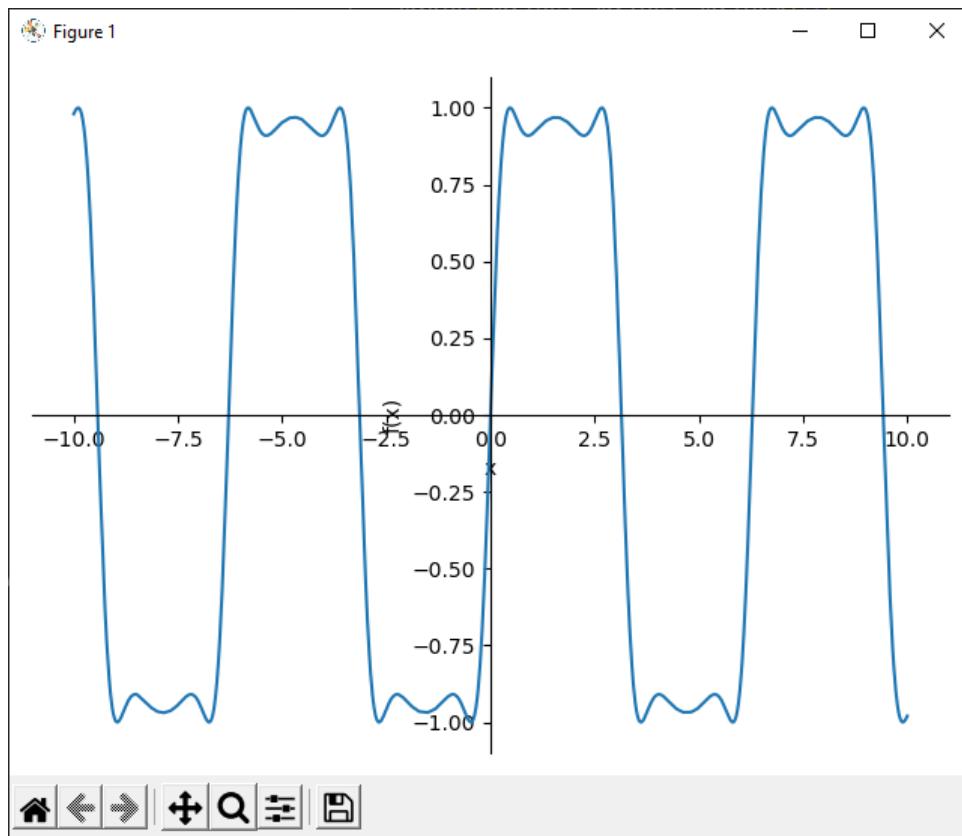
Sympy trae su propio módulo para hacer gráficos, que en realidad **utiliza matplotlib**, pero con la ventaja de que no es necesario darle valores numéricos para representar las funciones, si bien se pueden indicar límites.

```
import sympy as sp
x = sp.Symbol('x')
sp.plot(sp.sin(x)*sp.log(x))
```

Cálculo simbólico con Sympy
Gráficos con Sympy



```
sp.plot(sp.sin(2*sp.sin(2*sp.sin(x))))
```



8.13 Exportando a LATEX

Cuando hemos terminado los cálculos, podemos pasar a *LATEX* la ecuación que queremos:

```
In [5]: int1 = sp.integrate( sp.sin(x)**3 /(2*x), x)
In [6]: sp.latex(int1)
Out[6]: '\int \frac{\sin^3\left(x\right)}{2\ x}\, dx'
```

8.14 Ejemplos prácticos.

Ley de Gases Ideales.

```
...
Ley de Gas Ideal
...
import sympy as sym

P, V, n, R, T = sym.symbols('P, V, n, R, T')

# Gas constant
R = 8.314 # J/K/gmol
R *= 1000 # J/K/kgmol

# Moles of air
mA = 1 # kg
mwAir = 28.97 # kg/kg-mol
n = mA/mwAir

# Temperatura
T = 298

# Equation
eqn = sym.Eq(P*V, n*R*T)

# Solve for P
f = sym.solve(eqn, P)
print(f[0])

# Use the sympy plot function to plot
sym.plot(f[0], (V, 1, 10), xlabel='Volume m**3', ylabel='Pressure Pa')
```

Cálculo simbólico con Sympy
Ejemplos prácticos.

Referencias

- [1] L. Joyanes Aguilar, Fundamentos de programación: Algoritmos, estructura de datos y objetos, 4a. ed., Madrid: McGraw-Hill, 2008.
- [2] A. Dasso y A. Funes, Introducción a la Programación, Universidad Nacional de San Luis, 2014.
- [3] P. Gomis, Fundamentos de Programación en Python, Catalunya: Universitat Politècnica de Catalunya, 2018.
- [4] A. Marzal Varó, I. Gracia Luengo y P. García Sevilla, Introducción a la Programación con python 3, Castellón de la Plana: Publicacions de la Universitat Jaume I, 2014.
- [5] A. Soto Suarez y M. Arriagada Benítez, Introducción a la Programación con Python, Pontificia Universidad Católica de Chile, 2015.
- [6] J. M. R. Torres, Manual básico, iniciación a Python 3, 2020.
- [7] E. Bahit, Python para Principiantes, Buenos Aires, 2012.

Referencias
Ejemplos prácticos.

- [8] M. J. Mathieu, *Introducción a la programación*, PRIMERA EDICIÓN EBOOK ed., México: Grupo Editorial Patria, S.A. de C.V., 2014.
- [9] E. Bahit, *Introducción al Lenguaje Python*, Buenos Aires, 2018.