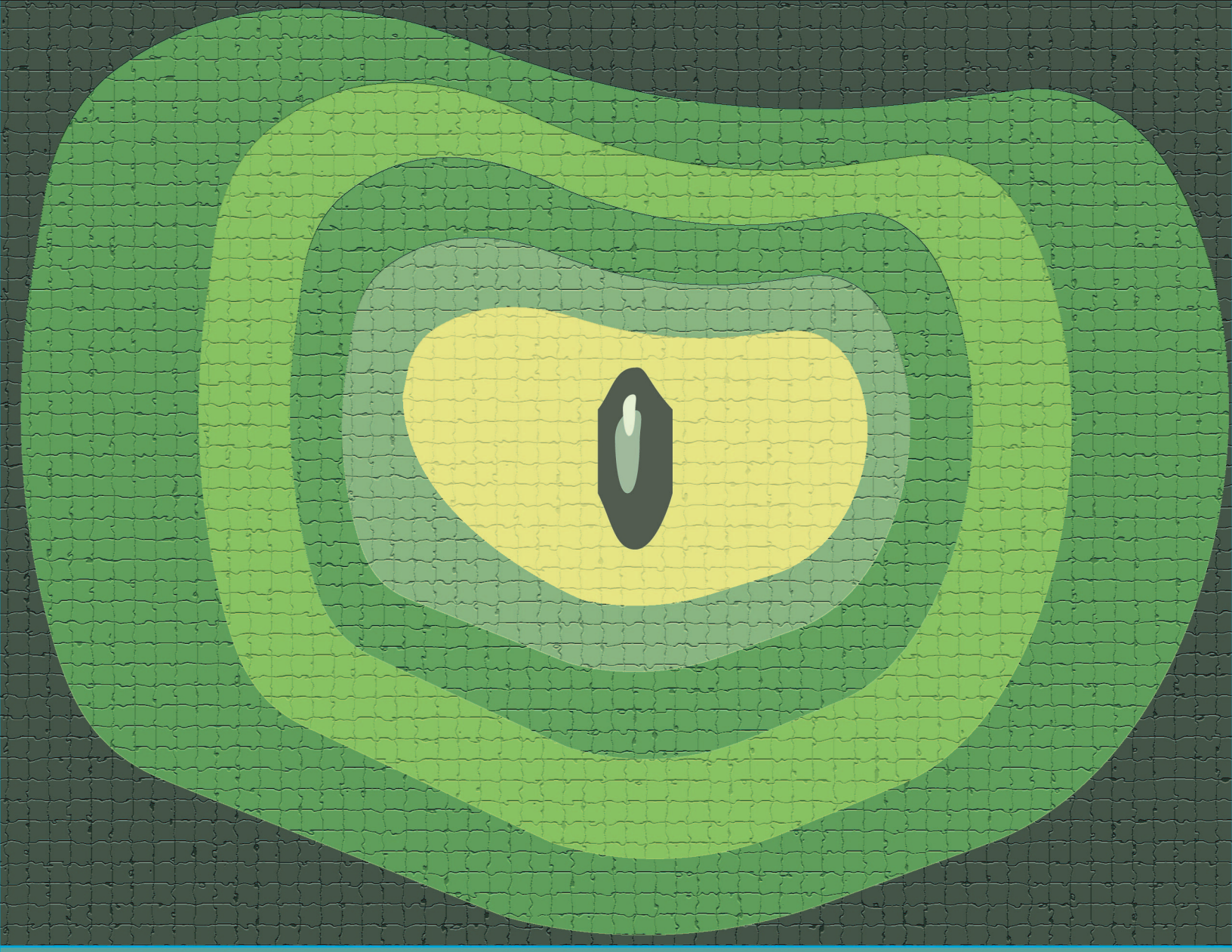


Algoritmos y programación en Python

Un enfoque práctico para programar

Roberto Enrique Alberto Lira

Universidad Juárez Autónoma de Tabasco

"Estudio en la duda. Acción en la fe"

Algoritmos y programación en Python
Un enfoque práctico para programar

Roberto Enrique Alberto Lira

Universidad Juárez Autónoma de Tabasco

"Estudio en la duda. Acción en la fe"

Guillermo Narváez Osorio

Rector

Gerardo Delgadillo Piñon

Director de la División Académica de Ciencias Básicas

Algoritmos y programación en Python
Un enfoque práctico para programar

C O L E C C I Ó N
HÉCTOR GARCÍA MOLINA
Informática y sistemas computacionales

Primera edición, 2022

www.ujat.mx

ISBN: 978-607-606-595-2

Para su publicación esta obra ha sido dictaminada por el sistema académico de pares ciegos. Los juicios expresados son responsabilidad del autor o autores y fue aprobada para su publicación.

Maquetación: Roberto Enrique Alberto Lira
Corrección de estilo: Francisco Cubas Jiménez
Portada: Walk Iria Chi Balan

Hecho en Villahermosa, Tabasco, México.

Dedicado a
mi familia: mis padres, hermanos y
sobrinos

Un agradecimiento especial a
quienes se tomaron el tiempo de
leer y hacer observaciones a esta
obra.

Agradecimiento especial al
Mtro. José Edilberto Rodríguez Cervera
por su apoyo como compañero y amigo
en la Universidad

A mis compañeros de la academia
de ciencias computacionales de la DACB
quienes fueron mis maestros y ahora mis amigos

Solo me resta agradecer a mi alma mater:
Universidad Juárez Autónoma de Tabasco

Índice general

Prefacio

Debido a las dificultades que se le presentan a los estudiantes de los primeros cursos de programación, el autor se dio a la tarea de observar y analizar en donde tienen sus orígenes esas dificultades; llegando a la conclusión de que en muchos casos los estudiantes no poseen bases sólidas en lógica y aritmética. Estas disciplinas son sumamente importantes para que un estudiante pueda realizar fluidamente un curso básico de programación. Por lo que en esta obra se abordan la lógica y la aritmética antes de empezar a programar.

También el autor ha notado que muchos libros de programación se centran en la sintaxis de un lenguaje en particular, dejando de lado la parte fundamental de la programación, la cual es la lógica. De igual modo, algunos libros resultan sumamente complejos de seguir para los estudiantes que aún inician en el maravilloso mundo de la programación. Por lo anterior, en esta obra se presenta un enfoque un tanto diferente de los libros tradicionales que enseñan a programar, esperando aportar un granito de arena para mejorar la comprensión de la programación.

Para el desarrollo de esta obra se ha empleado el lenguaje de programación Python (en su versión 3), el cual se seleccionó por su simpleza y su intuitiva sintaxis. Estas bondades del lenguaje permiten que el estudiante de programación se centre más en la lógica del problema que en la sintaxis en sí. Cabe señalar que Python es uno de los lenguajes más empleados actualmente y con mucho futuro por delante.

El contenido de este libro está sobre la base de que se parte sin conocimiento alguno de programación, por lo que se ha tratado de explicar cada concepto desde el nivel más básico posible. Cabe señalar que se espera que el lector tenga cierto conocimiento del funcionamiento de una computadora. En particular, son necesarios conocimientos básicos, como instalar y ejecutar programas, conocer y saber moverse por la estructura de directorios de su sistema operativo, etc.

Este libro se ha escrito empleando los sistemas operativos Windows y Mac OS. La instalación de las herramientas de desarrollo presentada en el capítulo 2 se centra principalmente en Windows. Sin embargo, es importante mencionar que el código que en este libro se muestra funciona en cualquier sistema operativo.

El contenido de este libro se ha dividido en 9 capítulos. El primer capítulo sienta las bases para que el lector desarrolle habilidades de lógica, principalmente habilidades del pensamiento y de aritmética. Estas habilidades son necesarias para la implementación y comprensión de programas. El capítulo 2 proporciona los fundamentos tanto de

algoritmos como de Python, con la finalidad de que el lector pueda dar seguimiento de forma fluida a los conceptos que se presentan posteriormente. Los tres capítulos siguientes abordan las estructuras de control de flujo con las que se puede realizar cualquier programa de computadora. El capítulo 6 muestra las posibles formas de estructurar los datos con los que se trabaja al momento de programar. El capítulo 7 nos presenta el concepto en programación de funciones, las cuales permiten mejorar la calidad de nuestro código. En el capítulo 8 se realiza un breve análisis de la complejidad de los algoritmos, con la finalidad de determinar cuáles podrían darnos mejores resultados al programar. Finalmente, unas vez adquiridas las habilidades de lógica y el conocimiento básico del lenguaje Python, en el capítulo 9 se procede a aplicar estas habilidades en programas enfocados a la programación competitiva.

Finalmente, es importante mencionar que la funcionalidad de los programas en Python presentados en esta obra, ha sido cuidadosamente probada. Sin embargo, para los ejercicios mostrados en dicha obra, pueden existir varias soluciones. Por lo que el autor, presenta la solución en programa que desde su punto de vista considera la más adecuada.

Convenciones usadas en este libro

Cuando se mencione en el texto algún elemento del lenguaje, éste aparecerá resaltado y en una topografía distinta como la palabra `hola`.

Los ejemplos de código se muestran del siguiente modo:

```
1 def saludo():
2     print("Hola mundo...")
3     print("Esta es mi primera función...")
4
5 saludo()
```

Cuando ejecutamos el programa, tenemos como salida:

```
Hola mundo...
Esta es mi primera función...
```

Cada programa cuenta con números de línea para una mejor referencia. Siempre que sea posible, los ejemplos de código van seguidos por la salida del programa (es decir, el resultado de ejecutar el código).

Debido a su extensión, en el primer capítulo se van proponiendo ejercicios para el lector por cada tema visto. En los siguientes capítulos, los ejercicios se proponen al final.

Capítulo 1

Pensamiento Lógico

1.1. Habilidades del pensamiento

La habilidad del pensamiento es la capacidad de desarrollo de procesos mentales que permiten dar solución a diversas circunstancias. Existen diferentes tipos de habilidades del pensamiento, por ejemplo: para expresar ideas, para presentar argumentos lógicos, para recordar experiencias pasadas, etc.

> Desarrollar las habilidades básicas de pensamiento implica que el individuo logre tanto la fase cognitiva —vivir el proceso— como las fase metacognitiva —darse cuenta del proceso—. Su utilidad es doble: por un lado, sirven para la mayoría de los asuntos cotidianos; por el otro, sirven de base para los asuntos que deseamos comprender de manera más profunda, rigorosa, completa y clara. (Ledama, 2016)

1.1.1. Lógica

Conceptos básicos

Lógica. Disciplina que estudia los métodos y principios que se usan para distinguir el razonamiento bueno (correcto) del malo (incorrecto) (Johnsonbaugh, 2005).

Inferencia. Proceso de razonamiento, compuesto por proposiciones, por el cual se deriva o extrae una conclusión de una o varias premisas.

Enunciado. Entidad lingüística conformada por palabras.

Proposición. Información contenida en un enunciado que es verdadera o falsa.

Argumento. Conjunto de proposiciones que sirven de premisas que conducen a una conclusión.

Premisa. Proposición aseverada o supuesta que sirve de apoyo o razón para aceptar la conclusión de un argumento.

Conclusión. Es la proposición aseverada con base en otras proposiciones (premisas) del argumento.

Lógica proposicional

La lógica proposicional es el nivel más básico de la lógica, se encarga de analizar las relaciones entre proposiciones, así como la verdad o la falsedad de éstas.

Elementos de la lógica proposicional

Variables. Las variables proposicionales son los símbolos que sustituyen a las proposiciones. Se llaman de ese modo porque su significado cambia en las diferentes argumentaciones o expresiones donde se utilicen.

Las letras más comunes para asignar las variables son p, q, r, s, t.

Conectores. Alteran, relacionan o conectan enunciados simples haciéndolos complejos. Los más frecuentes son la negación ($\neg$), la conjunción ($\wedge$), la disyunción ($\vee$), el condicional ($\longrightarrow$) y el bicondicional ($\longleftrightarrow$).

Auxiliares. Cuando los enunciados complejos en un solo reglón son muchos, se utilizan los símbolos auxiliares. No tienen ningún significado lógico, pero se usan con el objetivo de clarificar la comprensión de los enunciados. Los símbolos auxiliares son los paréntesis $(\dots)$ y los corchetes $[\dots]$.

Inferencia lógica

Inferencia. Es un razonamiento, compuesto por proposiciones, por el cual se deriva o extrae una conclusión de una o varias premisas.

El término inferencia es considerado como sinónimo de "derivación" o "deducción".

Tipos de inferencia

1. Según el número de premisas
   Inferencia inmediata. Es una forma de razonamiento que presenta una sola premisa, de la cual derivamos una conclusión.
   P: Si José vive lejos
   C: entonces llegará tarde.

Inferencia mediata. Es una forma de razonamiento compuesto por dos o más premisas de las cuales se deriva la conclusión.
P1. Todos los amigos se aprecian
P2. Luis y María son amigos
C. Luis y María se aprecian.

2. Según la forma de razonamiento
Inferencia deductiva. Es una forma de razonamiento cuya conclusión se deriva del contenido directo de las premisas enunciadas, haciendo referencia expresa de los términos enunciados.
P1. Ningún herbívoro come carne.
P2. Las jirafas comen hierba.
C. Ningún jirafa come carne.

Inferencia inductiva. Es un razonamiento cuyas premisas representan casos particulares de las cuales se deriva una conclusión que resulta un principio general.
P1. El gato tiene cola
P2. El perro tiene cola.
P3. La vaca tiene cola.
C. Toda animal de cuatro patas tiene cola.

1.1.2. Tablas de verdad

Conjunción

En razonamiento formal, una **conjunción lógica** ($\wedge$) entre dos proposiciones es un conector lógico cuyo valor de la verdad resulta en cierto solo si ambas proposiciones son ciertas, y en falso de cualquier otra forma (Alberto, Schwer, Cámara, y Fumero, 2005)(Gómez, s.f.).

Ejemplo
P=Está lloviendo
Q=Hace frío

P	Q	P∧Q
V	V	V
V	F	F
F	V	F
F	F	F

Tabla 1.1: Conjunción

Disyunción

En razonamiento formal, una **disyunción lógica** (o) entre dos proposiciones es un conector lógico ($\vee$), cuyo valor de la verdad resulta en falso solo si ambas proposiciones son falsas, y en cierto de cualquier otra forma (Alberto y cols., 2005).

Ejemplo
P=Está lloviendo
Q=Hace frío

P	Q	P$\vee$Q
V	V	V
V	F	V
F	V	V
F	F	F

Tabla 1.2: Disyunción

Condicional

El **condicional**, denotado por el símbolo $\longrightarrow$ conecta dos proposiciones de la siguiente forma: P$\longrightarrow$Q. La proposición P se denomina **antecedente** y la proposición Q se denomina **consecuente**. Esta proposición condicional puede expresarse como: si P entonces Q. La tabla de verdad es la siguiente:

Sea:
P=Está lloviendo
Q=Hace frío

P	Q	P$\longrightarrow$Q
V	V	V
V	F	F
F	V	V
F	F	V

Tabla 1.3: Condicional

Bicondicional

La **bicondicional**, denotada por el símbolo $\longleftrightarrow$ conecta dos proposiciones de la siguiente forma: P$\longleftrightarrow$Q. Esta proposición condicional puede expresarse como: P si y sólo si Q. La tabla de verdad es la siguiente:

Sea:
P=Está lloviendo
Q=Hace frío

P	Q	P$\longleftrightarrow$Q
V	V	V
V	F	F
F	V	F
F	F	V

Tabla 1.4: Bicondicional

Negación

La **negación** de una proposición P, se denota como: $\neg$P. Esta proposición $\neg$P puede expresarse como: "no P". Dicha proposición tiene el valor de verdad V cuando P tiene el valor de verdad F y viceversa. La tabla de verdad es la siguiente:

Ejemplo
P=Está lloviendo

P	$\neg$P
V	F
F	V

Tabla 1.5: Negación

Construcción de las tablas de verdad

Ejemplo 1:
Construir la tabla de verdad de la siguiente proposición:
P$\longrightarrow$(P$\wedge\neg$P)

P	$\neg$P	P$\wedge\neg$P	P$\longrightarrow$(P$\wedge\neg$P)
V	F	F	F
F	V	F	V

Tabla 1.6: Tabla de verdad de ejemplo 1

En la proposición del ejemplo anterior solo se tiene una variable (es decir, P), esta variable puede tomar dos valores: verdadero o falso (ver Tabla 1.6 columna 1). Cabe destacar que el número de filas de una tabla se encuentra relacionado con el número de

variables y la relación es mediante la fórmula 2^n, en donde n es el número de variables. Por ejemplo, si se tienen 3 variables (P, Q, R), el número de filas de la tabla sería de $2^3 = 8$.

Ya que se tienen definidos los valores de las variables (ver Tabla 1.6 columna 1), como primer paso, se realiza lo que se encuentra entre paréntesis. Lo que implica que se debe obtener la negación de P (segunda columna de Tabla 1.6). Teniendo la negación de P, se procede a realizar lo que falta de la expresión que se encuentra entre paréntesis (tercera columna de Tabla 1.6). Finalmente se realiza la expresión completa (cuarta columna de Tabla 1.6).

Ejemplo 2:
Construir la tabla de verdad de la siguiente proposición:
$(P \vee Q) \vee \neg Q$

P	Q	P∨Q	¬Q	(P∨Q)∨¬Q
V	V	V	F	V
V	F	V	V	V
F	V	V	F	V
F	F	F	V	V

Tabla 1.7: Tabla de verdad de ejemplo 2

Primero vemos cuántas variables tiene la proposición, en este caso, son 2 (P y Q). Utilizamos la fórmula 2^n para determinar el número de filas de la tabla, que sería $2^2 = 4$ (ver Tabla 1.7 columnas 1 y 2). En seguida, realizamos lo que se encuentra entre paréntesis en la proposición, es decir, $P \vee Q$ (ver Tabla 1.7 columna 3). Posteriormente, obtenemos la negación de Q (ver Tabla 1.7 columna 4). Finalmente, resolvemos la expresión (ver Tabla 1.7 columna 5) con los datos obtenidos en las columnas anteriores de la tabla.

Circuitos lógicos

Un **circuito digital** es aquel que maneja la información en forma binaria, es decir, con valores de "1 y 0".

Estos dos niveles lógicos de voltaje fijos representan:

- "1"nivel alto o "high"
- "0"nivel bajo o "low"

Lógica y circuitos eléctricos

Podemos ver los valores "verdadero - falso", del predicado P, como valores "1 y 0". En donde un 0 equivale a falso y un 1 a verdadero.

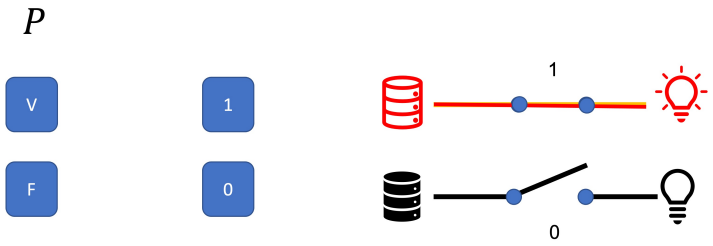


Figura 1.1: Circuito eléctrico

Conjunción

La conjunción se representa mediante un circuito en serie. Recordemos que en la conjunción para que un estado sea verdadero, ambas proposiciones deben ser verdaderas; para todos los demás casos, es falsa. Partiendo de esta definición de conjunción, podemos representarla mediante un canal en el cual fluye corriente eléctrica. Este canal tiene dos "puentes", si los dos puentes están cerrados fluye la corriente hasta su punto final. En cualquier otro caso, la energía deja de fluir.

P	Q	P∧ Q
1	1	1
1	0	0
0	1	0
0	0	0

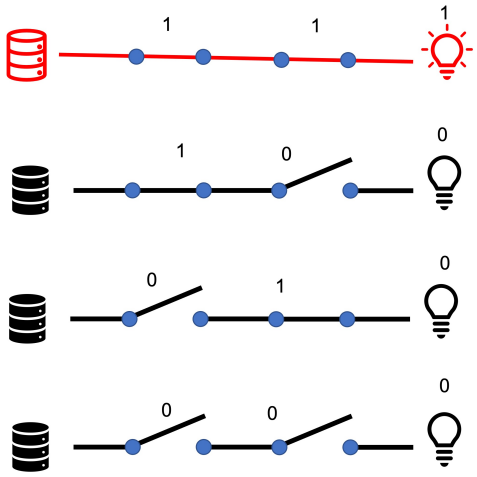


Figura 1.2: Circuito eléctrico de la conjunción

Disyunción

Se representa mediante un circuito conectado en paralelo. Recordemos que en la disyunción para que un estado sea falso, ambas proposiciones deben ser falsas; para todos

los demás casos, es verdadero. Partiendo de esta definición de disyunción, podemos representarla mediante un canal en paralelo, en el cual fluye electricidad. Este canal tiene dos "puentes", si los dos puentes están abiertos deja de fluir la energía. En cualquier otro caso, la energía fluye hasta el final del canal.

P	Q	P ∨ Q
1	1	1
1	0	1
0	1	1
0	0	0

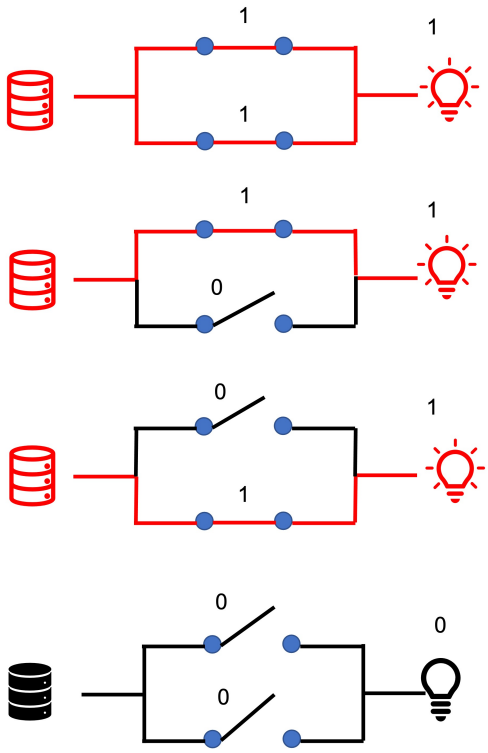


Figura 1.3: Circuito eléctrico de la disyunción

En resumen

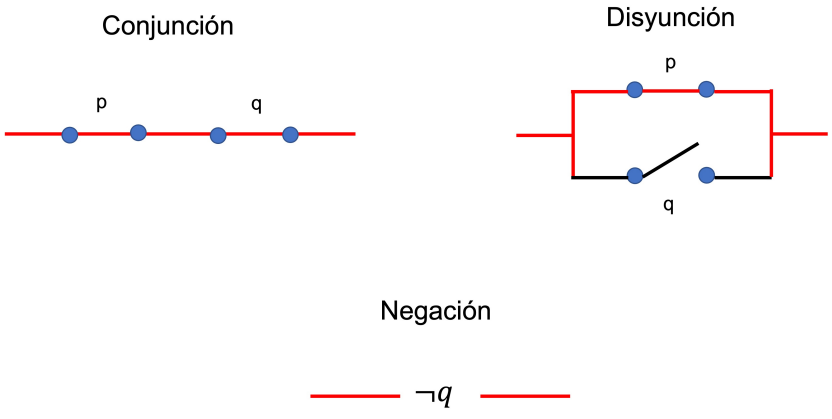


Figura 1.4: Representación de los circuitos eléctricos

Ejemplo 1
Representar la siguiente proposición a su respectivo circuito lógico.

$(\neg r \vee \neg p) \wedge p$

Solución:

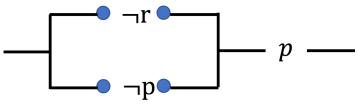


Figura 1.5: Solución ejemplo 1

Ejemplo 2
Determinar la proposición lógica a partir del circuito lógico siguiente:

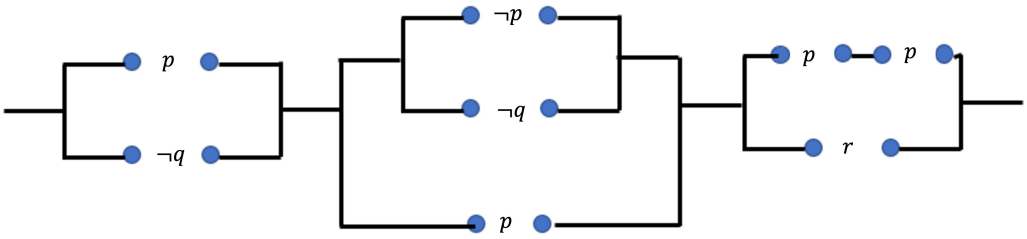


Figura 1.6: Circuito ejemplo 2

Solución

$(p \vee \neg q) \wedge [(\neg p \vee \neg q) \vee p] \wedge [(p \wedge p) \vee r]$

Ejercicios Lógica

1. Construye la tabla de verdad para la siguiente proposición:

   $(p \wedge q) \longrightarrow (q \wedge \neg p)$

2. Construye la tabla de verdad para la siguiente proposición:

   $((p \wedge (q \longrightarrow r)) \vee ((q \wedge \neg r) \longleftrightarrow \neg r)$

3. Construye el circuito lógico para la siguiente proposición:

   $[(p \vee q) \wedge ((r \wedge \neg q) \vee (q \vee \neg r))] \vee [(\neg p \vee (r \wedge \neg p))]$

4. Construye el circuito lógico para la siguiente proposición:

   $[(r \vee \neg q) \wedge ((p \wedge \neg q) \vee (q \vee \neg r))] \wedge r$

5. Construye la proposición a partir del circuito lógico

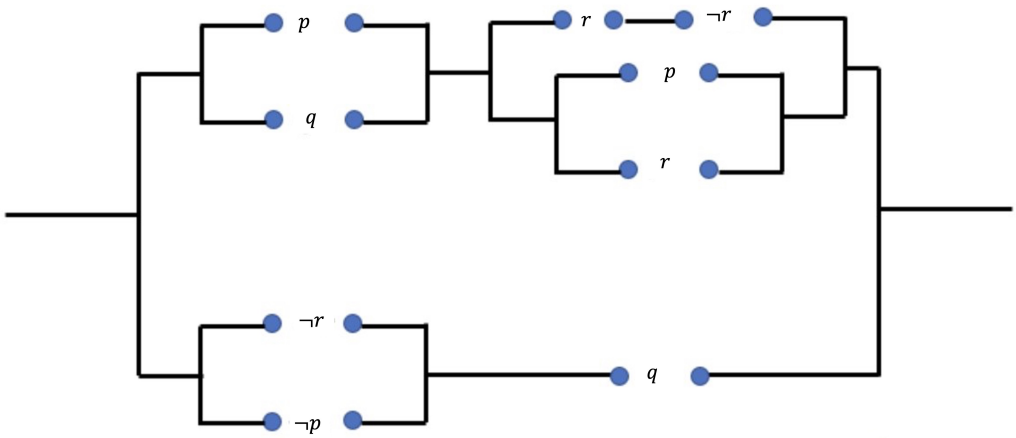


Figura 1.7: Circuito lógico

1.1.3. Habilidades lógico matemáticas

Estas habilidades pueden ser definidas como nuestra capacidad de razonamiento formal para resolver problemas relacionados con los números y las relaciones que se pueden establecer entre ellos, así como para pensar siguiendo las reglas de la lógica.

> En la inteligencia lógico-matemática se dan la mano la matemática y la lógica porque pensar a través de ambas, requiere seguir las normas de un sistema formal, desprovisto de contenidos: uno más uno es igual a dos, sean lo que sean las unidades con las que se trabaja, al igual que algo que es no puede no ser, independientemente de lo que se trate. En definitiva, estar dotados en mayor o menor medida de inteligencia lógico-matemática nos permite reconocer y predecir las conexiones causales entre las cosas que pasan (si le añado 3 unidades a estas 5, obtendré 8 porque las he sumado, etc.)(Torres, 2015)

Gracias a estas habilidades somos capaces de pensar de manera coherente, detectar regularidades en las relaciones entre las cosas y razonar lógicamente.

A continuación, se presentan ejemplos sobre algunas herramientas (acertijos, sudokus, conteos de figuras, etc.), que ayudan en el desarrollo o mejora de las habilidades lógico-matemáticas. Se sugiere al lector (en caso de conocer la herramienta) leer el ejercicio e intentar resolverlo sin ver la solución, la cual se presenta posterior al ejercicio.

Acertijos

Los acertijos lógicos son pasatiempos o juegos que consisten en hallar la solución de un enigma o encontrar el sentido oculto de una frase. Para resolver los acertijos más comunes hay que hacer uso de la imaginación y de la capacidad de deducción(Sués, 2015).

El trabajo de Juanito

Un estudiante universitario llamado Juanito por la tarde ayuda a su papá en el negocio familiar, en este negocio es común tener conversaciones como la siguiente:

- ¿Cuánto es de 20? - pregunta el cliente
- $ 600 - responde Juanito
- ¿Y cuánto cuesta 30? - pregunta el cliente
- $ 700 - responde Juanito
- Muy bien, llevaré 26 - dice el cliente
- Son $ 1,000 - responde Juanito

¿Qué vende Juanito?

Solución:

Juanito vende rótulos con letras. Cada letra de cada palabra cuesta $100. Por ejemplo, 20 (veinte) se escribe con 6 letras, y como cada letra cuesta $100, se pagan $600.

El Transporte de los estudiantes universitarios

El chofer del camión que transporta a los estudiantes universitarios realiza una maniobra interesante para poder llegar a su destino. El camión tiene una altura de 5 m y debe pasar debajo de un letrero que mide 4.89 m, lo que significa que el camión sobrepasa por 11 cm la altura del letrero. El único camino que conoce el chofer, es el que pasa por debajo del letrero. Pasando dicho letrero hay tiendas donde venden de todo tipo de herramientas.

¿Cuál es la solución más rápida y que no requiere tumbar el letrero?

Solución:

Lo más rápido fue desinflar las ruedas del camión, con lo que su altura disminuye y por lo tanto puede pasar por debajo del letrero.

Sudoku

"Es un juego matemático que se inventó a finales de la década de 1970, adquirió popularidad en Japón en la década de 1980 y se dio a conocer en el ámbito internacional en 2005 cuando numerosos periódicos empezaron a publicarlo en su sección de pasatiempos. El objetivo del sudoku es rellenar una cuadrícula de 9 × 9 celdas (81 casillas) dividida en subcuadrículas de 3 × 3 (también llamadas "cajas" o "regiones") con las cifras del 1 al 9 partiendo de algunos números ya dispuestos en algunas de las celdas. Aunque se podrían usar colores, letras, figuras, se conviene en usar números para mayor claridad, lo que importa es que sean nueve elementos diferenciados, que no se deben repetir en una misma fila, columna o subcuadrícula. Un sudoku está bien planteado si la solución es única, algo que el matemático Gary McGuire ha demostrado que no es posible si no hay un mínimo de 17 cifras de pista al principio.

Reglas y terminologías

El sudoku se presenta normalmente como una tabla de 9 × 9, compuesta por subtablas de 3 × 3 denominadas "regiones" (también se le llaman "cajas" o "bloques").

Algunas celdas ya contienen números, conocidos como "números dados" (o a veces "pistas"). El objetivo es rellenar las celdas vacías, con un número en cada una de ellas, de tal forma que cada columna, fila y región contenga los números 1–9 solo una vez.

Además, cada número de la solución aparece solo una vez en cada una de las tres "direcciones", de ahí el "los números deben estar solos" que evoca el nombre del juego."(Wikipedia, 2022)

Ejemplo:
La cuadricula de la izquierda es el estado inicial del sudoku, su solución se muestra en la cuadricula de la derecha. Estas cuadriculas tienen 9 regiones (líneas gruesas) y cada región tiene 9 cuadros (líneas delgadas).

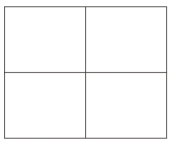
	3	4		7			1	
6			1	9		3	4	8
	9	8			2	5	6	7
8	5		7	6	1		2	3
4	2	6	8	5	3	7	9	1
	1		9	2	4	8	5	
9		1		3		2	8	4
		7	4		9	6	3	5
		5	2	8	6		7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figura 1.8: Ejemplo sudoku

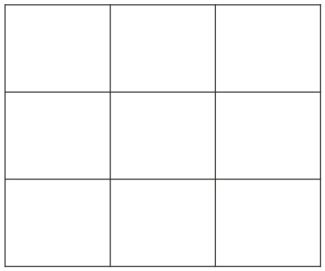
Conteo de figuras

Determinar cuántos cuadrados se encuentran en cada una de las figuras.



$= 5$

La respuesta es 5, puesto que hay 4 cuadros pequeños (digamos de dimensiones 1×1), los cuales están contenidos dentro de un cuadro más grande (dimensiones 2×2).

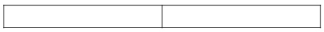


$= 14$

La respuesta es 14, porque hay 9 cuadros pequeños (digamos de dimensiones 1×1). Tenemos también 4 cuadros medianos (digamos de dimensiones 2×2). Finalmente, un cuadro grande que contiene todos los cuadros (de dimensiones 3×3).

$9 + 4 + 1 = 14$

Determinar cuántos rectángulos hay en cada figura

 $= 3$

En total hay 3 rectángulos, dos contenidos en uno más grande

$2 + 1$

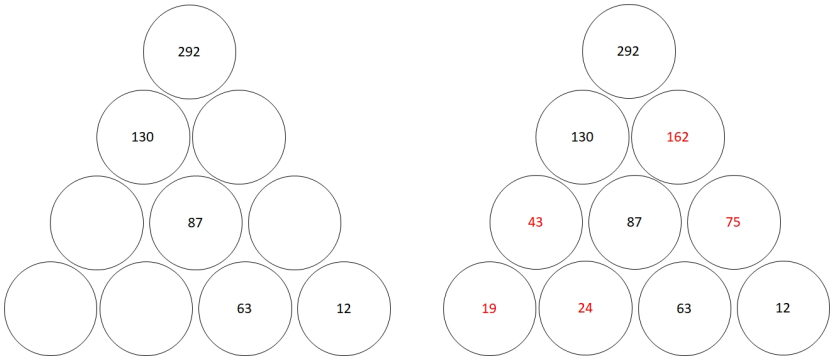
 $= 10$

En total hay 10 rectángulos

$4 + 3 + 2 + 1$

Completar valores

Completa los valores que faltan, sabiendo que el valor en cada círculo corresponde a la suma de los valores en los dos círculos debajo.



Letras en desorden

Reto: Descubre los nombres de países que están escritos en desorden.

Ejemplo:

icranaf: Francia

paesañ: España

lbirza: Brazil

puer: _____

acioblmo: _____

uiasr: _____

naraitgne: _____

Ejercicios de habilidades lógico-matemáticas

1. Resuelve el siguiente acertijo.

   Encontrando al culpable

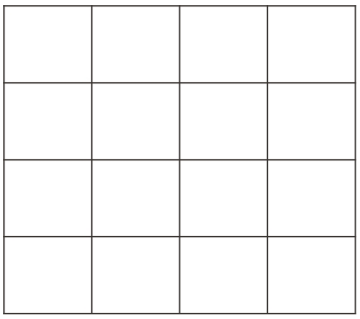
   José es un joven muy inquieto y muy inteligente que le gusta enterrar objetos "valiosos". Un día vio que uno de sus objetos no estaba donde lo había dejado enterrado. La única pista eran rodadas de neumáticos marcadas en el lodo del terreno poco transitado. José siguió las rodadas y llegó a una choza, en donde vio a 3 hombres sentados. Estos hombres no tenían lodo, ni vehículo, sin embargo, al verlos José dedujo quién era el sospechoso.

   ¿Cómo pudo José resolver el caso tan rápido?

2. Resuelve el siguiente sudoku.

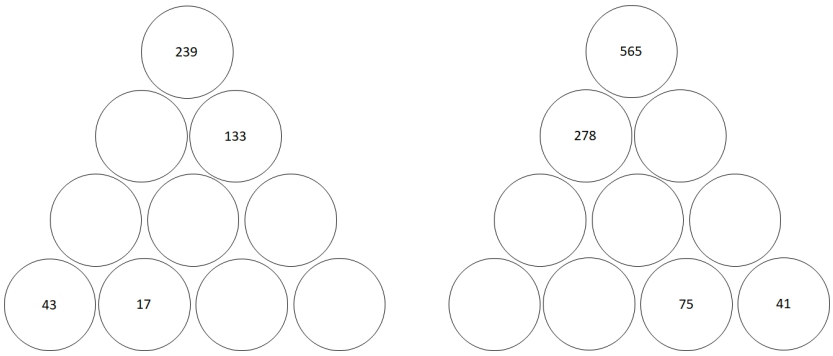
2	1		9	4			3	
				3		6		
	3	7	6		2	1		
		3	4		7	8	2	9
	7						1	
8	2	1	5		3	4		
		8	1		5	3	4	
		2		7				
	4			6	9		7	8

3. Determina cuántos cuadrados se encuentran en la siguiente figura.



4. Determina cuántos rectángulos se encuentran en la siguiente figura.

5. Completa los valores que faltan, sabiendo que el valor en cada círculo corresponde a la suma de los valores en los dos círculos debajo.



1.1.4. Teoría de conjuntos

La teoría de conjuntos se entiende como un contenido del área de matemáticas, pero sus utilidades van mucho más allá del desarrollo del pensamiento lógico-matemático. Comprender la teoría de conjuntos nos permite utilizar los conjuntos como herramienta para **analizar**, **clasificar** y **ordenar** los conocimientos adquiridos desarrollando la compleja red conceptual en la que almacenamos nuestro aprendizaje.

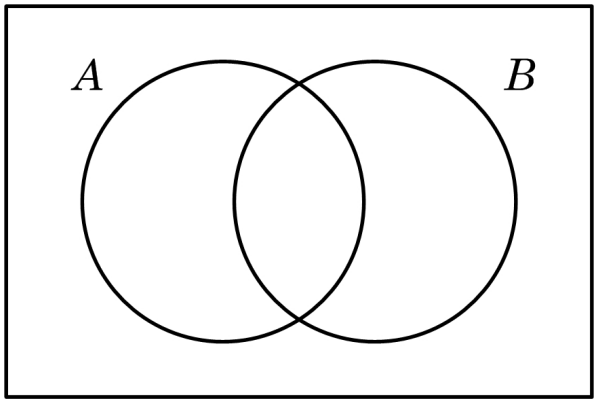
Conceptos básicos

- **Conjunto**. Un conjunto es la agrupación, clase, o colección de objetos o elementos que pertenecen y responden a la misma categoría (Grimaldi y Mateos, 1998). Los conjuntos se pueden definir de al menos cuatro formas:

    1. **Extensión o enumeración**: Sus elementos son encerrados entre llaves y separados por comas. Cada conjunto describe un listado de todos sus elementos. Además, sus elementos no se repiten.

A = {a, e, i, o, u}
B = {0, 2, 4, 6, 8, 10}

2. **Compresión**: Sus elementos se determinan a través de una condición que se establece entre llaves.
   A = {x | x es una vocal}
   B = {x | x es un número par menor que 11}
3. **Diagramas de Venn**: Regiones cerradas que nos permiten visualizar las relaciones entre los conjuntos.



4. **Descripción verbal**: Se trata de un enunciado que describe una característica común a todos los elementos del conjunto.

   "Los animales que tienen esqueleto interno".

- **Elemento**. Es cada uno de los objetos por los cuales está conformado un conjunto.
- **Pertenencia.** Sea $x \in A$ (El elemento x pertenece al conjunto A). El símbolo $\notin$ se utiliza cuando un elemento no pertenece al conjunto.
- **Subconjunto**. Sean los conjuntos A={ 9, 10, 11, 12, 13 } y B={ 10, 11, 13 }. Entonces, decimos que B es subconjunto de A. En general, si A y B son dos conjuntos, decimos que B es un subconjunto de A si todo elemento de B lo es también de A.

  Por lo tanto, si B es un subconjunto de A se escribe $B \subseteq A$.

  Si B no es subconjunto de A se indicará con una diagonal $B \nsubseteq A$.

Tipos de conjuntos

Conjunto universo (U, Ω)

"El conjunto universo es un conjunto formado por todos los objetos de estudio en un contexto dado. Se denota como U, también se puede denotar como Ω"(Elmer, 2015)
U = {x | x es una persona}
A = {x | x es una mujer}
B = {x | x es un hombre}

Conjunto vacío ({ }, ∅)

"Se denomina así al conjunto que no tiene ningún elemento. A pesar de no tener elementos se le considera como conjunto. Se denota como {}, aunque también se puede denotar como ∅"(Elmer, 2015)
Ejemplos:
Conjunto de los días de la semana que terminan en e.
Conjunto de números pares que no sean múltiplos de 2.

Conjunto unitario

"Es el conjunto que tiene un solo elemento."(Elmer, 2015)
Ejemplo: Conjunto de los días del año en que se celebra el día del maestro en México

Conjuntos disjuntos

"Se llaman conjuntos disjuntos aquellos que no tienen ningún elemento que pertenezca a ambos al mismo tiempo."(Elmer, 2015)
Ejemplo: Los dos conjuntos siguientes:
{x | x es un ave}
{x | x es un mes del año}
Son disjuntos ya que no tienen ningún elemento común.

Conjuntos iguales

Los conjuntos son iguales, si tienen los mismos elementos, por ejemplo: El conjunto { x, y, z } también puede escribirse:
{ x, z, y }, { y, x, z }, { y, z, x }, { z, x, y }, { z, y, x }
En teoría de conjuntos se acostumbra a no repetir los elementos, por ejemplo:
El conjunto { a, a, a, b, b, } simplemente será { a, b }.

Conjunto infinito

Es aquel conjunto cuya cantidad de elementos no se puede contar, es decir, es aquel conjunto en que sus elementos no se pueden nombrar o enumerar. Son ejemplos de conjuntos infinitos, los conjuntos numéricos: los naturales, los enteros, los reales, los racionales, los imaginarios, los complejos.

Conjunto numerable

> En Teoría de Conjuntos se definen los conjuntos numerables (o conjuntos contables) como aquellos conjuntos que continen un número natural de elementos. Los conjuntos numerables pueden ser finitos o infinitos de manera que se puede establecer una relación de numeración entre dichos conjuntos y el conjunto de los números naturales. (Matematicas10, 2018)

Por ejemplo, A = {batería, guitarra, teclado, saxofón} es contable ya que se le puede asociar el subconjunto de los números naturales {1, 2, 3, 4}

OPERACIONES CON CONJUNTOS

Unión

La unión de dos conjuntos A y B es el conjunto A ∪ B que contiene cada elemento que está por lo menos en uno de ellos.

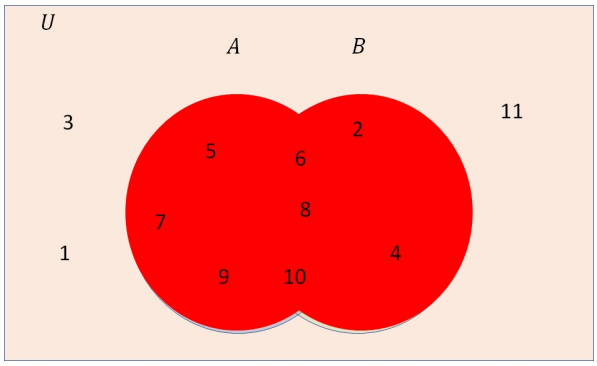


Figura 1.9: Gráfico de la unión de conjuntos

Ejemplo (ver fig. 1.9)
U={1,2,3,4,5,6,7,8,9,10,11}
A={5,6,7,8,9,10}

B={2,4,6,8,10}
A ∪ B={2,4,5,6,7,8,9,10}

Intersección

La intersección de dos conjuntos A y B es el conjunto A ∩ B que contiene todos los elementos comunes de A y B.
Ejemplos:
U={1,2,3,4,5,6,7,8,9,10,11}
A={5,6,7,8,9,10}
B={2,4,6,8,10}
A ∩ B={6,8,10}

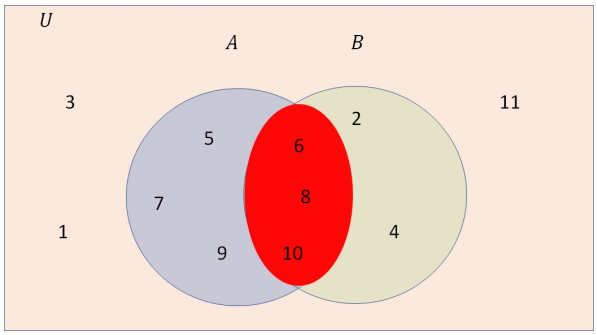


Figura 1.10: Gráfico de la intersección de conjuntos

Diferencia

La diferencia entre dos conjuntos A y B es el conjunto A - B que contiene todos los elementos de A que no pertenecen a B.
Ejemplos:
U={1,2,3,4,5,6,7,8,9,10,11}
A={5,6,7,8,9,10}
B={2,4,6,8,10}
A-B={5,7,9}

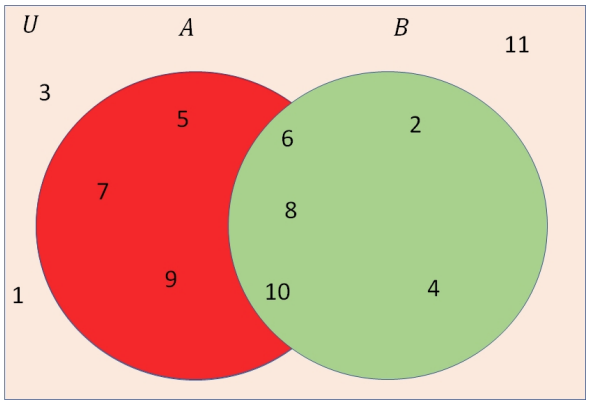


Figura 1.11: Gráfico de la resta de conjuntos

Complemento

El complemento de un conjunto A es el conjunto $\overline{A}$ que contiene todos los elementos (respecto de algún conjunto referencial) que no pertenecen a A.
Ejemplos:
U={1,2,3,4,5,6,7,8,9,10,11}
A={5,6,7,8,9,10}
$\overline{A}$={1,2,3,4,11}

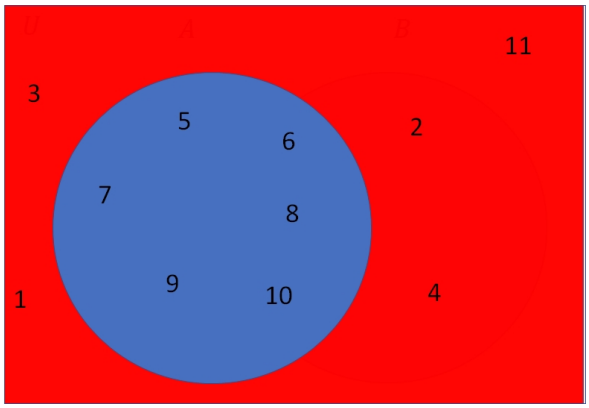


Figura 1.12: Gráfico del complemento de conjuntos

Ejercicios de conjuntos

1. Dados los conjuntos A y B
   A = {2,4,6,8,10,11,12,14,16}
   B = {1,3,5,7,9,11,13,15,17}

Encuentre:
A ∩ B
A ∪ B
A - B
B - A

2. Dados los conjuntos A y B, A = {x | x es una letra del abecedario}, a≤x≤m (x una letra entre a y m)
   B = x | x es una vocal
   Encuentre:
   A ∩ B
   A ∪ B
   A - B
   B - A

3. Encuentre $\overline{A \cup B}$
   A = {2,4,6,8,10,11,12,14,16}
   B = {1,3,5,7,9,11,13,15,17}
   U = {x | x es un número natural}, $1 \leq x \leq 25$

4. Encuentre $\overline{A \cap B}$
   A = {2,4,6,8,10,11,12}
   B = {1,3,5,7,9,11,13}
   U = {x | x es un número natural}, $1 \leq x \leq 15$

5. En una clase se presentó un examen de dos preguntas:
   15 personas respondieron bien la primera pregunta.
   10 personas respondieron bien la segunda pregunta.
   8 personas respondieron bien las dos preguntas.
   5 personas no respondieron ninguna.
   ¿Cuántas estudiantes presentaron el examen? Represente gráficamente

1.2. Aritmética en la lógica

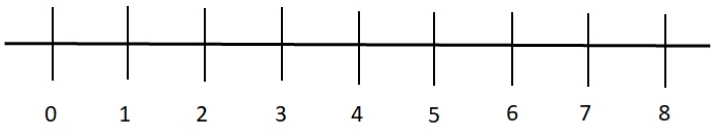
La aritmética(Wikipedia, 2019) es la rama de la matemática cuyo objeto de estudio son los números y las operaciones elementales hechas con ellos: adición, sustracción, multiplicación y división.

1.2.1. Los números naturales

"Los números naturales son los que usamos para contar; 1, 2, 3, 4,....El conjunto de los números naturales, como es obvio, contiene un número infinito de elementos." (M. Peters, 2022)

Representación de los números naturales

Los números naturales se puede representar en una recta. Partiendo de cero debes trazar marcas con las mismas distancias. Las marcas se corresponderán con los números 1, 2, 3, 4, 5...



De esta manera puedes comparar dos números, pues al colocarlos sobre la recta el que quede a la derecha será el mayor. Como consecuencia, el número que queda a la izquierda es el menor.
Ejemplo:
10 es mayor que 3 y se representa 10>3
O también podemos decir que:
3 es menor que 10 y se representa 3<10

OPERACIONES BÁSICAS CON NÚMEROS NATURALES

Suma o adición

- Operación interna: La suma de dos números naturales es otro número natural.
- Propiedad conmutativa: El orden de los sumandos no altera el resultado: a + b = b + a
  Ejemplo
  3 + 4 = 4 + 3 = 7
- Propiedad asociativa: Para sumar tres o más números naturales, el orden en que se haga no modifica el resultado: a + (b + c) = (a + b) + c
  Ejemplo
  2+(7+1) = (2+7)+1 = 10
- El elemento neutro es el 0: a + 0 = 0 + a
  Ejemplo
  4 + 0 = 0 + 4 = 4

Resta o sustracción

- Operación no interna: La resta de dos números naturales no siempre es otro número natural. Para que así fuera, el orden tiene que ser número mayor – número

menor.
Ejemplo:
$6 - 9$ no es un número natural porque no tiene sentido (hasta el momento) quitarle 9 unidades a un conjunto cuando solo hay 6.

- No se cumple la propiedad conmutativa: El orden de los factores altera el resultado: $a - b \neq b - a$, pues una de esas operaciones no da como resultado un número natural (por la propiedad anterior).
  Ejemplo:
  $20 - 10 \neq 10 - 20$ porque $20 - 10 = 10$ mientras que $10 - 20$ no se puede hacer.

- No se cumple la propiedad asociativa: Al restar tres o más números naturales, el orden en que se haga modifica el resultado.
  Ejemplo:
  $10 - (4 - 1) \neq (10 - 4) - 1$
  $10 - 3 \neq 6 - 1$
  $7 \neq 5$

- No hay elemento neutro: Al no darse la propiedad conmutativa, no existe ningún elemento que la satisfaga, pero se puede decir que el 0 es el elemento neutro por la derecha, pues $a - 0 = a$.
  Ejemplo:
  $6 - 0 \neq 0 - 6$ porque $6 - 0 = 6$ mientras que $0 - 6$ no se puede hacer.

Multiplicación o producto

- Operación interna: El producto de dos números naturales es otro número natural.

- Propiedad conmutativa: El orden de los factores no altera el producto: $a \cdot b = b \cdot a$
  Ejemplo:
  $5 \cdot 4 = 4 \cdot 5$

- Propiedad asociativa: Para multiplicar tres o más números naturales, el orden en que se haga no modifica el resultado: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
  Ejemplo:
  $6 \cdot (2 \cdot 5) = (6 \cdot 2) \cdot 5$

- El elemento neutro es el 1: $a \cdot 1 = 1 \cdot a$
  Ejemplo:
  $12 \cdot 1 = 1 \cdot 12 = 12$

- Propiedad distributiva: Un factor que multiplica a una suma puedes escribirla como la suma de los productos de dicho factor con los sumandos, es decir, $a \cdot (b + c) = a \cdot b + a \cdot c$
  Ejemplo:
  $3 \cdot (6 + 2) = 3 \cdot 6 + 3 \cdot 2$

- Factor común: Una suma en la que los sumandos tienen un elemento en común, puedes escribirla como el producto de dicho elemento y la suma de los sumandos sin ese elemento. Es decir, $a \cdot b + a \cdot c = a \cdot (b + c)$
  Ejemplo:
  $2 \cdot 5 + 2 \cdot 7 = 2 \cdot (5 + 7)$

División o cociente de números naturales

- Operación no interna: La división de dos números naturales no siempre es otro número natural. Para que así fuera, el resto debería ser 0 o lo que es lo mismo, la división exacta.
  Ejemplo:
  El resultado de dividir 3 entre 5 no es un número natural porque la división no es exacta.

- No se cumple la propiedad conmutativa: El orden de los factores altera el resultado: $a / b \neq b / a$.
  Ejemplo:
  $10 / 5 \neq 5 / 10$

- No se cumple la propiedad asociativa: Al dividir tres o más números naturales, el orden en que se haga modifica el resultado.
  Ejemplo:
  $24 / (6 / 2) \neq (24 / 6) / 2$
  $24 / 3 \neq 4 / 2$
  $8 \neq 2$

- No hay elemento neutro: al no darse la propiedad conmutativa, no existe ningún elemento que la satisfaga pero puedes decir que el 1 es el elemento neutro por la derecha, pues $a / 1 = a$
  Ejemplo:
  $25 / 1 = 25$ pero $1 / 25 \neq 25$

- No existe la división entre 0: Dividir entre 0 significaría repartir algo en 0 grupos; al no haber grupos entre los que repartir, no tiene sentido hacer la división.

Ejercicios de números naturales

1. El cocinero del comedor de la División Académica de Ciencias Básicas cuenta con 10 docenas de tomates para preparar 3 tipos de salsas y con los tomates que sobren cocinará platillos mexicanos. Para cada salsa se utilizan 10 tomates y cada platillo mexicano lleva 1 tomate.
   ¿Cuántos platillos mexicanos podrán hacerse?

2. María ha comprado 70 paletas para vender en la escuela. Cada paleta le costó $ 10 y las pone en venta en $ 20 la unidad. Como solo consigue vender 40, las rebaja a $ 15, de modo que consigue venderlas todas.
   ¿Cuánto dinero obtiene María con las ventas de sus paletas?

1.2.2. Los números enteros

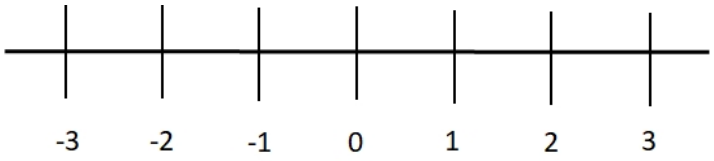
Los números enteros están formados por los números naturales, sus opuestos (es decir, con un signo "-" delante) y el cero 0. Por tanto, podemos definir un número entero como un elemento del conjunto.

Z = {...-4, -3, -2, -1, 0, 1, 2, 3, 4...}

La aritmética nos presenta los números enteros para abordar ciertas carencias que los números naturales no pueden hacer.

Representación y orden de los números enteros

Al igual que el conjunto de los números naturales, un número entero se puede representar en una recta. Partiendo del cero debes hacer marcas con la misma distancia a la derecha y a la izquierda. Las marcas de la derecha se corresponderán con los números 1, 2, 3, 4, ... y las de la izquierda con el -1, -2, -3, -4, ...



Puedes comparar dos numeros, pues al colocarlos sobre la recta, el que quede a la derecha será el mayor.
Ejemplo:

1 es mayor que -2 y se representa 1 >-2.
Observa que cuando ambos enteros son negativos, el criterio es el mismo:
Ejemplo:

-3 es mayor que -6, -3 >-6, pues al dibujarlos en la recta -3 queda a la derecha.

Valor absoluto

El valor absoluto de un número es la distancia de dicho número al 0 y se representa con el número entre las llamadas barras de valor absoluto ||.
Ejemplo:

El valor absoluto de 7, $|7|$, es 7 porque la distanacia es de 7 unidades del 0 y el valor absoluto de -3, $|-3|$, es 3 porque la distancia es de 3 unidades respecto al 0.
Observa que el valor absoluto de un número es positivo pues las distancias siempre son positivas.

OPERACIONES BÁSICAS CON NÚMEROS ENTEROS

Suma o adición

Si ambos sumandos tienen el mismo signo, se suman los valores absolutos y se mantiene el signo:
Ejemplo:

10 + 6 = 16, siendo ambos positivos,
-2 + (-6) = -8, si los dos son negativos.

Si los sumandos tienen diferente signo, se restan los valores absolutos y se pone el signo del sumando que tiene mayor valor absoluto.
Ejemplo:

Para hacer 4 + (-8) tienes que calcular los valores absolutos |4|=4, |-8|=8, restarlos 8 – 4 = 4 y como -8 tiene mayor valor absoluto, el resultado será negativo, es decir, 4 + (-8) = -4.
Y si tuvieras -2 + 3, restando los valores absolutos 3 – 2 = 1 el resultado sería -2 + 3 = 1, con signo positivo pues 3 tiene mayor valor absoluto.

Propiedades de la suma

- Operación interna: La suma de dos números enteros es otro número entero.
- Propiedad conmutativa: El orden de los sumandos no altera el resultado:$a + b = b + a$.
  Ejemplo:
  -5 + 4 = 4 + (-5)
- Propiedad asociativa: Para sumar tres o más números enteros, el orden en que se haga no modifica el resultado: $a + (b + c) = (a + b) + c$
  Ejemplo:
  $5 + (-7 + 4) = (5 + (-7)) + 4$
  $5 + (-3) = -2 + 4$
  $2 = 2$
- El elemento neutro es el 0: $a + 0 = 0 + a$
  Ejemplo:
  $-5 + 0 = 0 + (-5)$

- El inverso aditivo de un número es aquel que hace que la suma valga 0: $a+(-a) = -a+a = 0$
  Ejemplo:
  $3+(-3) = -3+3 = 0$

Resta o sustracción

Para restar dos números enteros no tienes más que transformar la resta en una suma, $a-b = a+(-b)$ y operar como en el paso anterior:
Ejemplo:
5 – 9 = 5 + (-9) = -4
-3 – 6 = -3 + (-6) = -9

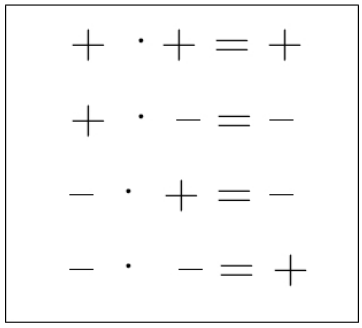
Propiedades de la resta

- Operación interna: La resta de dos números enteros es otro número entero.
- No se cumple la propiedad conmutativa: El orden de los factores altera el resultado: $a-b \neq b-a$.
  Ejemplo:
  20 – 10 ≠ 10 – 20
  10 ≠ -10
- No se cumple la propiedad asociativa: Al restar tres o más números naturales, el orden en que se haga modifica el resultado.
  Ejemplo:
  10 – (-4 – 1) ≠ [10 – (-4)] – 1
  10 – (-5) ≠ 14 – 1
  15 ≠ 13
- No hay elemento neutro: Al no darse la propiedad conmutativa, no existe ningún elemento que la satisfaga, pero puedes decir que el 0 es el elemento neutro por la derecha, pues a – 0 = a
  Ejemplo:
  -3 – 0 = -3 pero 0 – (-3) = 3

Multiplicación o producto

Para multiplicar dos números enteros tienes que multiplicar los valores absolutos de ambos factores. El signo vendrá determinado por la siguiente regla:

Regla de los signos



Ejemplo:
9 · 3 = 27
9 · (- 3) = -27
-9 · 3 = -27
-9 · (- 3) = 27

Propiedades de la multiplicación

- Operación interna: El producto de dos números enteros es otro número entero.
- Propiedad conmutativa: El orden de los factores no altera el producto: $ab = ba$
  Ejemplo:
  4 · (-2) = -2 · 4 = -8
- Propiedad asociativa: Para multiplicar tres o más números enteros, el orden en que se haga no modifica el resultado: $a(bc) = (ab)c$
  Ejemplo:
  8 · [(-3) · 2] = [(8 · (-3)] · 2
  8 · (-6) = -24 · 2
  -48 = -48
- El elemento neutro es el 1: a · 1 = 1 · a
  Ejemplo:
  -12 · 1 = 1 · (-12) = -12
- Propiedad distributiva: Un factor que multiplica a una suma puedes escribirla como la suma de los productos de dicho factor con los sumandos, es decir, $a(b+c) = ab + ac$
  Ejemplo:
  -9 · (-6 + 5) = -9 · (-6) + (-9) · 5
- Factor común: Una suma en la que los sumandos tienen un elemento en común, puedes escribirla como el producto de dicho elemento y la suma de los sumandos

sin ese elemento. Es decir, $ab + ac = a(b + c)$
Ejemplo:
$2 \cdot 5 + 2 \cdot (-7) = 2 \cdot [5 + (-7)]$

División o cociente

Para dividir dos números enteros tienes que dividir los valores absolutos de ambos factores. El signo vendrá determinado por la siguiente regla:

Regla de los signos

$$+ \,/\, + = +$$
$$+ \,/\, - = -$$
$$- \,/\, + = -$$
$$- \,/\, - = +$$

Ejemplo:
$4 \,/\, 2 = 2$
$4 \,/\, (-2) = -2$
$-4 \,/\, 2 = -2$
$-4 \,/\, (-2) = 2$

Propiedades de la división

- Operación no interna: La división de dos números enteros no siempre es otro número entero. Para que así fuera, el residuo debería ser 0 o lo que es lo mismo, la división exacta.
  Ejemplo:
  El resultado de dividir 4 entre 3 no es un número entero porque la división no es exacta.

- No se cumple la propiedad conmutativa: El orden de los factores altera el resultado: $a \,/\, b \neq b \,/\, a$.
  Ejemplo:
  16 pequeñas rebanadas de pastel divididas entre 4 niños da como resultado 4 rebanadas por niño. Sin embargo, 4 rebanadas de pastel divididas entre 16 niños no se puede realizar porque hay más niños que rebanadas.

- No se cumple la propiedad asociativa: Al dividir tres o más números enteros, el orden en que se haga modifica el resultado.

Ejemplo:
24 / (-12 / 4) $\neq$ [24 / (-12)] / 4
24 / (-3) $\neq$ -2 / 4

- No hay elemento neutro: puedes decir que el 1 es el elemento neutro por la derecha, pues a / 1 = a.
  Ejemplo:
  11 / 1 = 11; pero 1 / 11 $\neq$ 11

- No existe la división entre 0: Dividir entre 0 significaría repartir algo en 0 grupos; al no haber grupos entre los que repartir, no tiene sentido hacer la división.

Ejercicios de números enteros

1. Las lluvias comenzaron en el estado de Tabasco. Durante el día hace calor y por la noche refresca. Regularmente las lluvian inician a las 19 hrs y la temperatura desciende 2° cada hora hasta las 8 am del siguiente día.

   a) ¿Cuántos grados bajó la temperatura de las 19 hrs hasta las 3 am?

   b) Sí a las 19 hrs la temperatura estaba en 40°, ¿qué temperatura había a las 8 am?

2. Los padres de María le dan $ 3,560 a la semana para que viaje y compre lo que necesita en sus estudios universitarios. Sin embargo, ella gasta $ 4,600 a la semana; afortunadamente tiene muy buenas amigas que le ayudan a solventar sus gastos. ¿Cuál es la situación de María con sus amigas al mes?

1.2.3. Máximo común divisor

En matemáticas, se define el máximo común divisor (MCD) de dos o más números enteros al mayor número entero que los divide sin dejar residuo alguno.

Ejemplo:

Hallar el máximo común divisor de 48 y 60.

Solución. Los divisores de 48 son: 1,2,3,4,6,8,12,16,24,48

Los divisores de 60 son: 1,2,3,4,5,6,10,12,15,20,30,60

Por lo tanto: MCD(48,60)=12

Por definición, si a es un entero positivo, entonces MCD(a,0)=a

Algoritmo de Euclides

El algoritmo de Euclides es un método que sirve para calcular el MCD de dos o más números enteros. Para definir su procedimiento, es necesario hablar antes del algoritmo de la división, el cual enuncia lo siguiente: "Si a y b son enteros y b es mayor que cero, existen dos enteros q y r, únicos, tales que $a = bq+r$, con $0 \leq r < b$"(Espinosa Armenta, 2017). En la expresión $a = bq + r$, q es llamado el cociente y r el residuo en la división de a entre b. Observe que si r=0, entonces b divide a a.

A partir de la definición del algoritmo de la división, el algoritmo de Euclides enuncia lo siguiente: "Sean a y b dos enteros positivos tales que $a \geq b$. Definimos $r_0 = a$ y $r_1 = b$, y apliquemos repetidamente el algoritmo de la división para obtener

$$r_0 = r_1 q_1 + r_2, \qquad 0 < r_2 < r_1$$

$$r_1 = r_2 q_2 + r_3, \qquad 0 < r_3 < r_2$$

.

.

$$r_{n-2} = r_{n-1} q_{n-1} + r_n, \qquad 0 < r_n < r_{n-1}$$

$$r_{n-1} = r_n q_n + r_{n+1} \qquad 0 = r_{n+1}$$

Entonces $MCD(a, b) = r_n$ el último residuo distinto de cero."(Espinosa Armenta, 2017)

Ejemplo:

Aplicando el algoritmo de Euclides encuentra el MCD de 48 y 60.

$$60{=}48(1){+}12$$

$$48{=}12(4){+}0$$

Por lo tanto, $MCD(60, 48) = 12$

Ejemplo:

En la Carrera de Ciencias Computacionales hay 100 alumnos en total, 60 hombres y 40 mujeres. El profesor Francisco desea realizar un concurso de programación, por lo que decide agrupar a los alumnos del mismo género en laboratorios de cómputo, en los cuales debe haber el mismo número de estudiantes. El profesor, quiere que cada laboratorio cuente con el mayor número posible de cada género.

1. Hallar el número de alumnos que debe ir en cada laboratorio de cómputo
2. Hallar el número de laboratorios de cómputo que tienen solo hombres
3. Hallar el número de laboratorios de cómputo que tienen solo mujeres

Solución

1. Calculamos el MCD(60,40) aplicando el algoritmo de Euclides

$$60=40(1)+20$$

$$40=20(2)+0$$

Por lo tanto, $MCD(60, 40) = 20$

1. Deben ir 20 alumnos en cada laboratorio de cómputo

   Para los incisos 2) y 3), dividimos entre 20 que es el MCD de 60 y 40. Así, 60 hombres entre 20 alumnos por laboratorio es igual a 3 y 40 mujeres entre 20 alumnos por laboratorio es igual a 2.

2. Son 3 laboratorios de cómputo de hombres
3. Son 2 laboratorios de cómputo de mujeres

Ejercicios MCD

Juanito desea guardar sus 24 libros de matemáticas y 30 de programación en cajas, las cuales deben ser lo suficientemente grandes para almacenar el mayor número de ejemplares. Juanito es muy ordenado, por lo que quiere que cada caja tenga solo libros de la misma área y con la misma cantidad de ejemplares.

1. ¿Cuántos libros irán en cada caja?
2. ¿Cuántas cajas se necesitan para los libros de matemáticas? ¿Y cuántas cajas para los libros de programación?

1.2.4. Mínimo común múltiplo

"En matemáticas, el mínimo común múltiplo (abreviado mcm) de dos o más números naturales es el menor número natural distinto de cero que es múltiplo común de todos ellos."(Rees, 1986)

Ejemplo:

Hallar el mínimo común múltiplo de 48 y 60.

Solución. Los múltiplos de 48 son: 48,96, 144, 192, 240, 288,...

Los múltiplos de 60 son: 60,120,180,240,300,...

Por lo tanto: mcm(48,60)=240

Cálculo del mínimo común múltiplo

Existe un método para calcular el mcm, sin embargo, para utilizar este método es necesario obtener previamente el MCD.

La fórmula para calcular el mcm es la siguiente:

$$mcm(a,b) = \frac{ab}{MCD(a,b)}.$$

Ejemplo, calcular el mcm de 48 y 60:

$$mcm(48,60) = (48{\cdot}60) \div MCD(48,60) = 2880 \div 12 = 240$$

Ejemplo:

1. Luis y María son amigos que viven en Cunduacán y algunas veces se encuentran en una tienda muy conocida de la ciudad. Luis va cada 15 días a esta tienda a comprar despensa. María va cada 18 días. ¿Cada cuántos días coinciden en la tienda?

Solución

$$mcm(15,18) = (15{\cdot}18) \div MCD(15,18) = 90$$

En 90 días se encontrarán en la tienda.

Ejercicio m.c.m

1. En la División Académica de Ciencias Básicas hay dos actividades complementarias:

   - un coro que ensaya cada 4 días y,

- el circulo de lectura, que se reune cada 3 días.

¿Cada cuántos días coinciden los dos grupos?

1.3. Conclusiones

En este capítulo estudiamos el pensamiento lógico, el cual se abordó en 2 partes: habilidades del pensamiento y aritmética en la lógica. Se mostraron ejemplos y se plantearon ejercicios con la finalidad de que el lector principiante desarrolle el pensamiento lógico.

Capítulo 2

Fundamentos de algoritmos y de Python

2.1. Introducción

En este capítulo se proporcionan los fundamentos tanto de algoritmos como de Python, con la finalidad de tener bases sólidas que permitan dar seguimiento al libro. Si te estás iniciando en el mundo de la programación te recomiendo que no te saltes este capítulo.

2.2. Definición y características de los algoritmos

Para implementar la solución de un problema mediante el uso de una computadora es necesario establecer una serie de pasos que permitan resolver el problema. A este conjunto de pasos se le denomina algoritmo, el cual debe tener como característica final la posibilidad de transcribirlo fácilmente a un lenguaje de programación, para esto se utilizan herramientas de programación, que son métodos que permiten la elaboración de algoritmos escritos en un lenguaje entendible.

Un algoritmo, aparte de tener como característica la facilidad para transcribirlo, debe ser (Pinales y Velázquez, 2014)

1. **Ordenado**. Debe indicar el orden en el cual debe realizarse cada uno de los pasos que conducen a la solución del problema.

2. **Definido**. Esto implica que el resultado nunca debe cambiar bajo las mismas condiciones del problema, éste siempre debe ser el mismo.

3. **Finito**. No se debe caer en repeticiones de procesos de manera innecesaria; deberá terminar en algún momento.

Por consiguiente, el algoritmo es una serie de operaciones detalladas y no ambiguas que se ejecutan paso a paso, que conducen a la resolución de un problema, y se representan mediante una herramienta.

Además de esto, se debe considerar que el algoritmo, que posteriormente se transformará en un programa de computadora, debe considerar las siguientes partes:

- **Entrada**: información de partida
- **Proceso**: operaciones y cálculos por realizar
- **Salida**: resultados obtenidos

Los pasos necesarios para realizar un algoritmo son:

1. Análisis del problema
2. Diseño del algoritmo para resolver el problema
3. Verificación del algoritmo

Hoy en día, el uso de la palabra algoritmo se refiere a cualquier procedimiento que permita resolver un problema dado.

2.3. Identificadores

Los identificadores son los nombres que se les asignan a los objetos, los cuales se pueden considerar como variables o constantes, éstos intervienen en los procesos que se realizan para la solución de un problema, por consiguiente, es necesario establecer qué características tienen.

2.3.1. Constante

Un identificador se clasifica como constante cuando el valor que se le asigna a este identificador no cambia durante la ejecución o proceso de solución del problema. Por ejemplo, en problemas donde se utiliza el valor de PI, si el lenguaje que se utiliza para codificar el programa y ejecutarlo en la computadora no lo tiene definido, entonces se puede establecer de forma constante estableciendo un identificador llamado PI y asignarle el valor correspondiente de la siguiente manera:

PI = 3.1416.

2.3.2. Variables

Los identificadores de tipo variable son todos aquellos objetos cuyo valor cambia durante la ejecución o proceso de solución del problema. Por ejemplo, el sueldo, el pago, el descuento, etcétera, que se deben calcular con un algoritmo determinado, o, en su caso, contar con el lado (L) de un cuadrado que servirá para calcular y obtener su área.

A las variables del algoritmo se les identifica con un nombre. El nombre es una cadena de caracteres alfanuméricos que debe empezar con una letra y pueden ser mayúsculas o minúsculas. Dentro del nombre puede haber números o guiones bajos. Es muy recomendable que los nombres de las variables tengan un nombre que tenga relación con lo que representa. No puede haber dos variables con nombres iguales dentro de un algoritmo. Dado que una letra mayúscula es distinta de una minúscula, la variable VAR1 es distinta a var1 (Cervantes, Báez, Arízaga, y Castillo, 2017).

En todos los lenguajes de programación existe un conjunto de palabras reservadas que no se pueden utilizar como nombres de variables. En el transcurso del libro iremos conociendo muchas de las palabras reservadas de Python.

Tipos de variables

Los elementos que cambian durante la solución de un problema se denominan variables y se clasifican dependiendo de lo que deben representar en el algoritmo, por lo que pueden ser de tipo entero, real, lógica y string. Sin embargo, existen otros tipos de variables que son permitidos con base en el lenguaje de programación que se utilice para crear los programas, por consiguiente, al estudiar algún lenguaje de programación se deben dar a conocer esas clasificaciones.

Para el caso de este libro, se denominará **variables de tipo entero** a todas aquellas cuyo valor no tenga valores decimales. Por ejemplo: a=7 y b=-8 son variables enteras.

Por su parte, las **variables de tipo real** podrán tomar valores con decimales. Por ejemplo: a1=3.1416, b2=-30.23. También son variables reales aquellas que incluyen una potencia como a_real=1.23×10^3.

En caso de que las variables tomen valores de caracteres, se designarán string o de cadena, por ejemplo: nombre="Juan Pérez", ocupación="estudiante", etc.

Una variable lógica es aquella que solamente toma los valores **Verdadero** o **Falso**. Ejemplo de variables lógicas son:

Y=Verdadero
X=Falso

Inicialización de variables

Una manera de inicializar una variable es darle valor desde el inicio del algoritmo. Esto lo podemos hacer al declarar las variables. Por ejemplo, una variable llamada contador que inicialmente va a tener un valor de 0, la podemos declarar de la siguiente forma:

```
Inicio
  contador=0
  hacer algo
Fin
```

Asignación de valores a variables

Las operaciones básicas se realizan de la manera tradicional. El resultado se asigna a la variable donde se almacena el resultado. Por ejemplo, para almacenar la suma de las variables a y b en la variable c, se realizaría de la siguiente manera:

```
c=a+b
```

Operaciones con variables enteras

Las operaciones (con excepción de la división), que se realizan con variables enteras producen resultados enteros. Ejemplo, sea a=5 y b=2

a+b es 7
a-b es 3
a*b es 10

El resultado de la división en Python nos da un número real con punto decimal:
a/b es 2.5

Si deseamos obtener un resultado entero debemos usar doble símbolo de división:
a//b es 2

Usando la división entera nuestro resultado solo toma parte entera.

2.4. Entradas y salidas de información

Los cálculos que realizan las computadoras requieren de entrada de datos para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, salidas.
Las **operaciones de entrada** (Leer) permiten leer valores y asignarlos a determinadas variables.
Las **operaciones de salida** (Escribir) permiten mostrar los resultados que produce el programa.

2.5. Pseudocódigo

Sin duda, en el mundo de la programación el pseudocódigo es una de las herramientas más conocidas para el diseño de solución de problemas por computadora. Esta herramienta permite pasar casi de manera directa la solución del problema a un lenguaje de programación específico. El pseudocódigo es una serie de pasos claros y bien detallados que conducen a la resolución de un problema.

La facilidad de pasar casi de forma directa el pseudocódigo a la computadora ha dado como resultado que muchos programadores implementen de forma directa los programas en la computadora, lo cual no es muy recomendable, sobre todo cuando no se tiene la suficiente experiencia para tal aventura, pues se podrían tener errores propios de la poca experiencia acumulada con la solución de diferentes problemas.

Por ejemplo, el pseudocódigo para determinar el área de un cuadrado se puede establecer de la siguiente forma:

```
Inicio
  Leer lado
  area=lado*lado
  Escribir area
Fin
```

Como se puede ver, se establece de forma precisa la secuencia de los pasos por realizar; además, si se le proporciona siempre el mismo valor a la variable lado, el resultado del volumen será el mismo y, por consiguiente, se cuenta con un final.

2.6. Un ejemplo que aplica todos los conceptos vistos

1. Escribir un algoritmo para calcular el área de un triángulo dada la base y la altura (Esta parte corresponde al análisis del problema).

Este problema ya se encuentra bien definido, el **análisis** consiste en determinar las entradas y las salidas.

Como podemos ver, el ejemplo nos pide calcular el área de un triángulo dada la base y la altura. Entonces, la palabra clave aquí es "**dada**", ya que eso significa que nuestras **entradas** serán la base y la altura. Con la base y con la altura, que nos proporcionen podemos calcular el área del triángulo, la cual será nuestra **salida**.

Además, debemos definir los tipos de variables (entero, real, string) de nuestras entradas y salidas. En este caso, es conveniente que tanto nuestras entradas como las salidas sean reales. Es decir, que puedan considerar la parte fraccionaria.

2. Diseño del algoritmo

Entradas: base (real), altura (real)
Salidas: area (real)

```
1: Inicio
2: Leer base
3: Leer altura
4: area=(base*altura)/2
5: Escribir area
6: Fin
```

3. Comprobamos nuestro algoritmo mediante una prueba de escritorio.

Nos vamos al paso 2 del pseudocódigo. Entonces, supongamos que a la variable base le asignamos un valor de 5.5, esto es base=5.5

Vamos al paso 3 del pseudocódigo. Ahora a nuestra variable altura le asignamos un valor de 7.6, esto es altura=7.6

Continuamos con el paso 4 y calculamos el área mediante la fórmula ya bien conocida: area = (5.5*7.6) / 2=20.9. El área para esta prueba es de 20.9. Finalmente, escribimos el área y terminamos.

2.7. Python

Python es un lenguaje interpretado, de alto nivel y enfocado principalmente a la legibilidad y facilidad de aprendizaje y uso.

Python es un lenguaje multiplataforma, lo que significa que puede usarse en multitud de sistemas distintos. Funciona en computadoras con sistemas operativos Linux, BSD, Windows, etc.,(Hinojosa, 2016).

Python es software libre , y se distribuye bajo la licencia *Python Software Foundation License*. Esto significa que se distribuye gratuitamente y no necesita del pago de licencias para su uso, ya sea privado o comercial.

Guido van Rossum es el creador de Python; y la filosofía que quiso darle es que el código debe ser limpio y legible, simple sin ser limitado. Dicha filosofía hace que Python sea un lenguaje ideal para aprender e iniciarse en la programación.

Python es un lenguaje completo perfectamente funcional, muy potente, y viene acompañado por una serie de paquetes que facilitan funciones para el trabajo con casi cualquier cosa.

2.7.1. El zen de Python

El zen de Python (T. Peters, 2004) viene a ser la filosofía del lenguaje. Describe una serie de reglas que deberían seguirse tanto en el desarrollo del propio lenguaje, como en los programas que se hagan con él.

- Bello es mejor que feo
- Explícito es mejor que implícito
- Simple es mejor que complejo
- Complejo es mejor que complicado
- Plano es mejor que anidado
- Disperso es mejor que denso
- La legibilidad importa
- Los casos especiales no son lo suficientemente especiales como para romper las reglas

- Aunque la practicidad gana a la pureza
- Los errores nunca deberían pasar en silencio
- A menos que se silencien explícitamente
- Frente a la ambigüedad, evita la tentación de adivinar
- Debería haber una (y preferiblemente solo una) manera obvia de hacerlo
- Aunque pueda no ser obvia al principio, salvo que seas holandés
- Ahora es mejor que nunca
- Aunque nunca es mejor que el ahora correcto
- Si la implementación es difícil de explicar, es una mala idea
- Si la implementación es fácil de explicar, podría ser una buena idea
- Los espacios de nombres son una idea genial. ¡Hagamos más de eso!

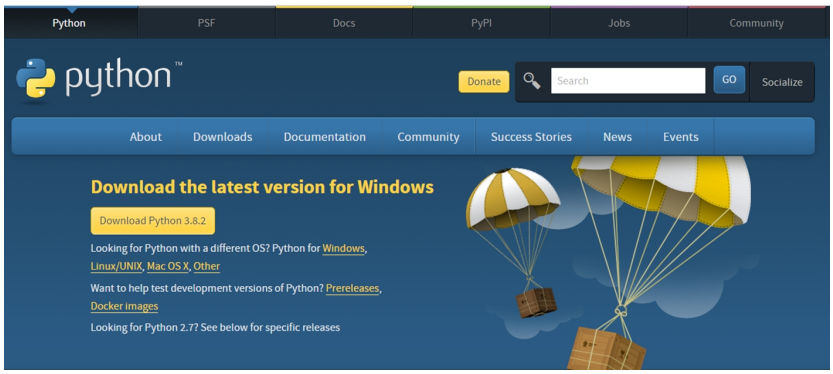
2.7.2. Instalación de Python y PyCharm en Windows

Actualmente, existen diversos entornos de desarrollo integrado (IDE) para programar en Python, por mencionar algunos: PyCharm, Spyder IDE, Visual Studio Code, Atom, IDLE, etc. También existen entornos especializados para áreas como ciencia de datos y aprendizaje automático, entre ellos: Anaconda, Miniconda, Google Colab (su uso es remoto, a través de un navegador de internet), etc.

En esta obra se utilizó el IDE PyCharm para programar los ejemplos. La elección de dicho IDE es únicamente por gusto del autor, ya que fue el primer editor que empleo para programar en Python.

Parte 1: Instalar Python

1. Ingresa a https://www.python.org/downloads/
2. Haz clic en Download Python 3.x.x

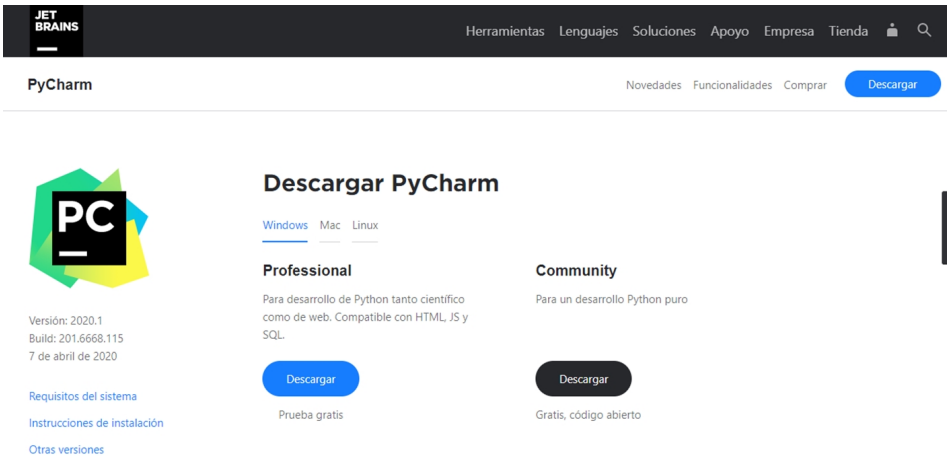


3. Ejecuta el instalador y selecciona ambas casillas, luego procede a instalarlo.

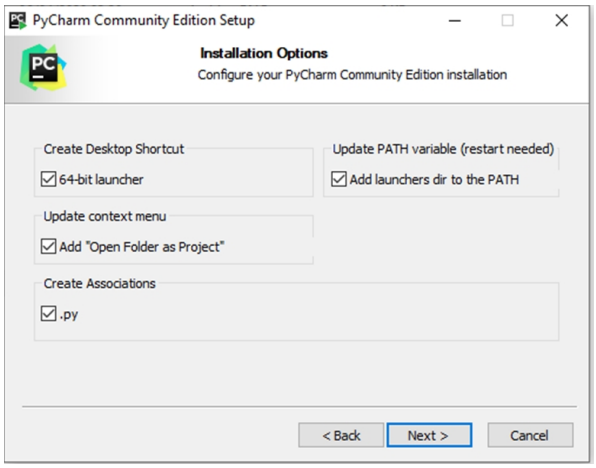
4. Comprueba la instalación escribiendo "python -V" en el CMD. De ser exitoso deberías ver la versión instalada.

Parte 2: Instalar Pycharm

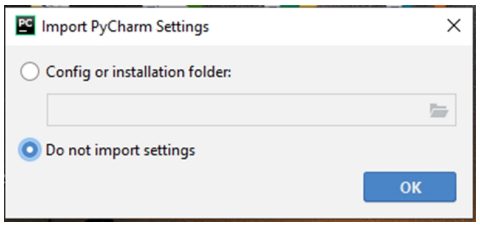
1. Ingresa a https://www.jetbrains.com/pycharm/download/#section=windows
2. Haz clic en Download Community



3. Ejecuta el instalador, selecciona las siguientes casillas y procede a instalarlo.

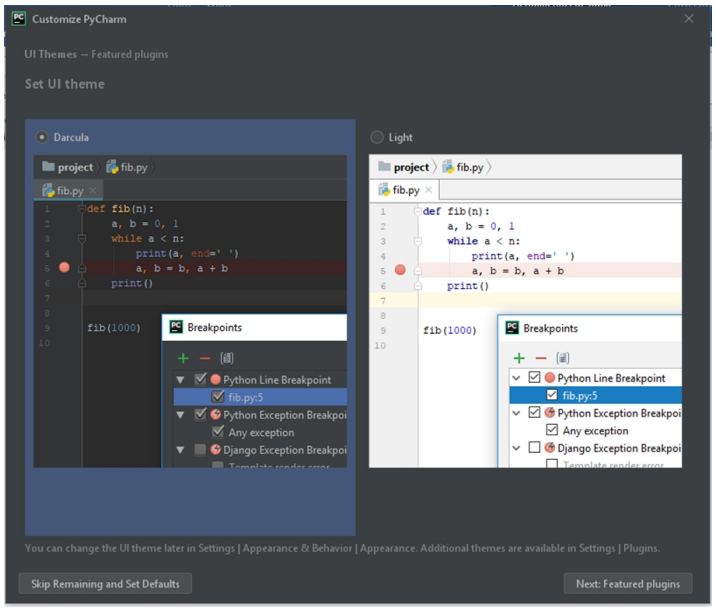


4. Inicia PyCharm, aparecerá este cuadro:



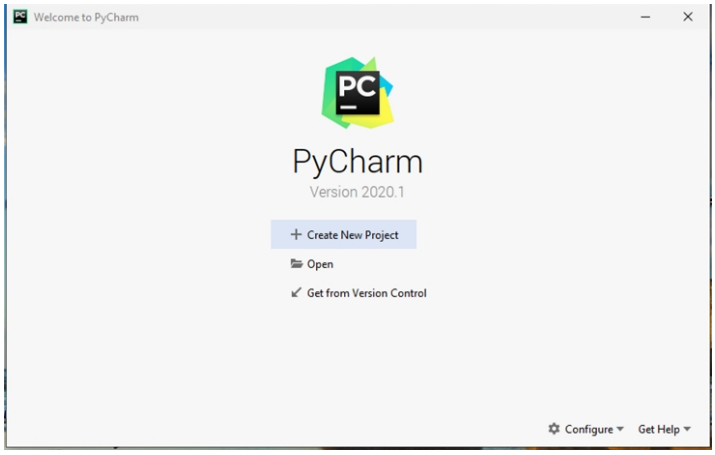
5. Deja la opción marcada y da clic en OK.

6. Dependiendo de tus gustos, selecciona el tema claro u oscuro.

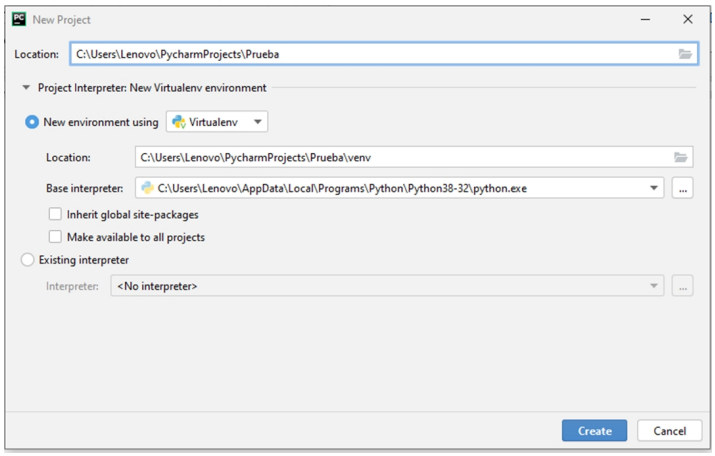


7. Haz clic en Skip Remaining and Set Defaults

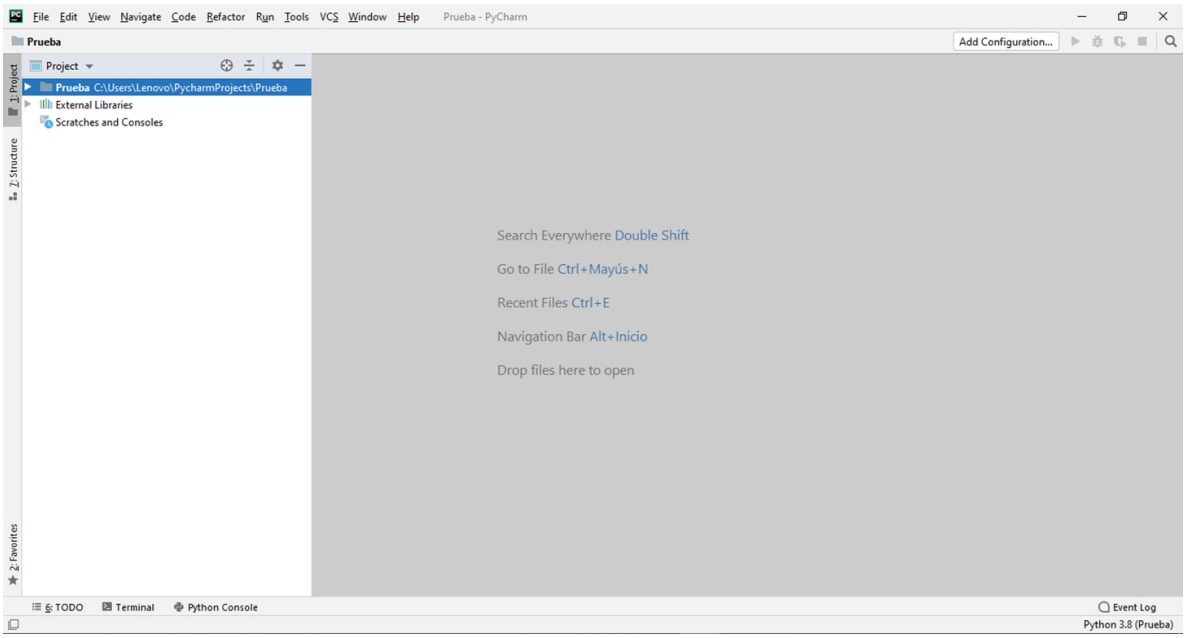
8. Aparecerá el siguiente cuadro:



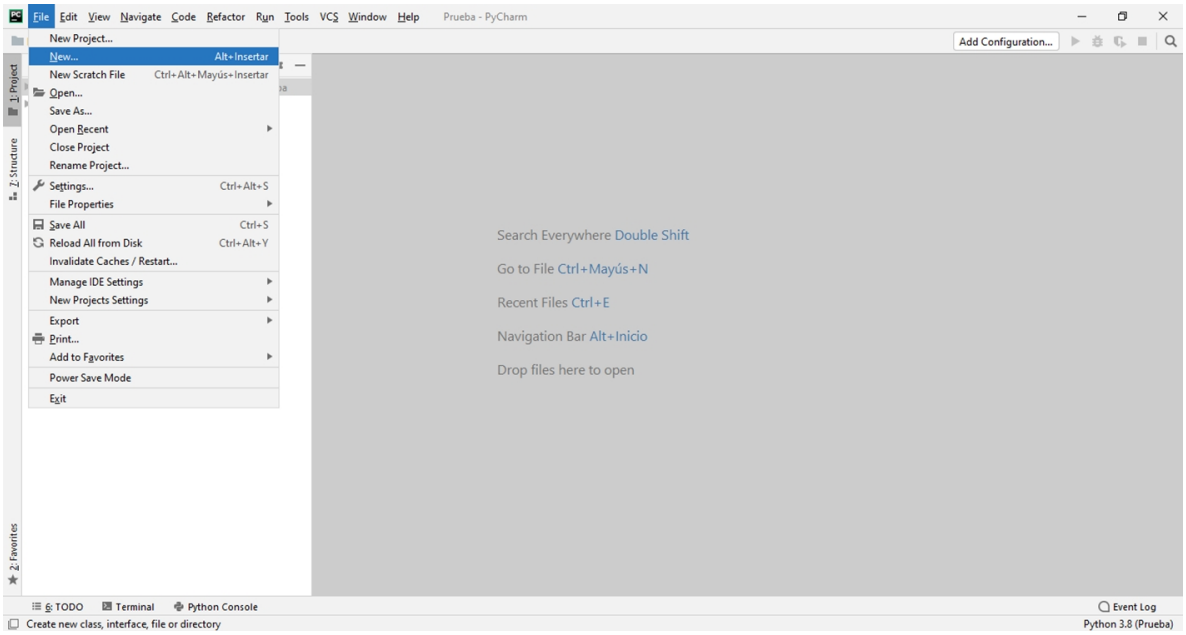
9. Haz clic en la opción Create New Project. Aparece un cuadro donde pondrás lo siguiente: Location: C:\...\PycharmProjects\NombreProyecto
Deja seleccionada la opción por default (New environment using), solo asegúrate que en Base interpreter se encuentra la ubicación donde se instaló Python.



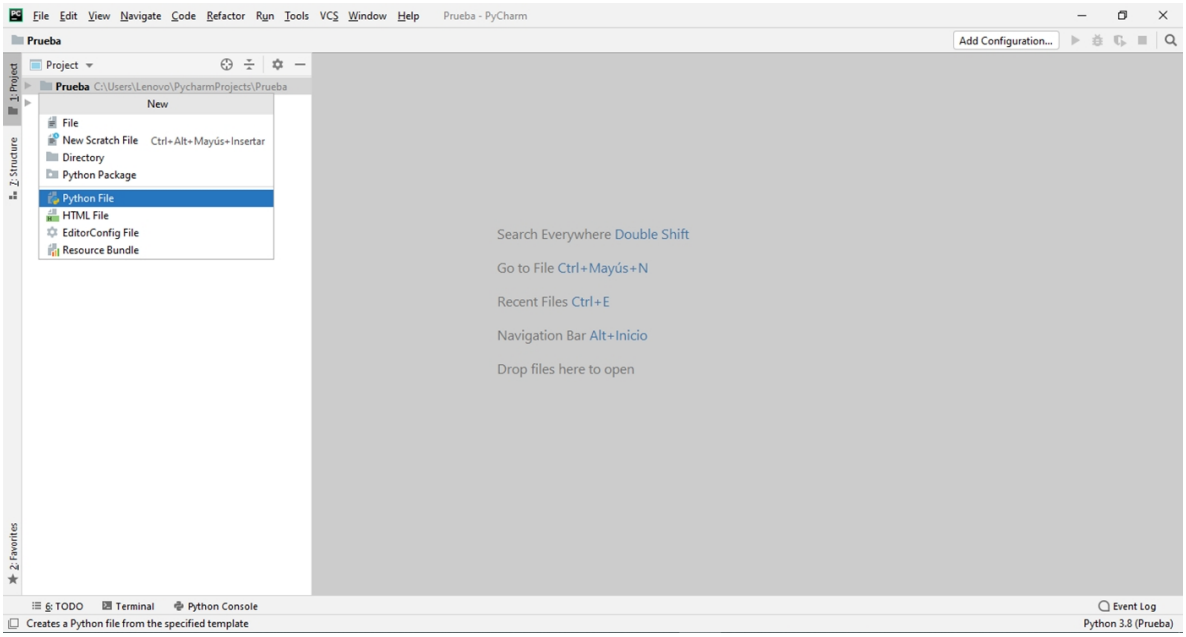
10. Aparece una ventana como la siguiente:



11. Haz clic en el menú File y selecciona la opción New.

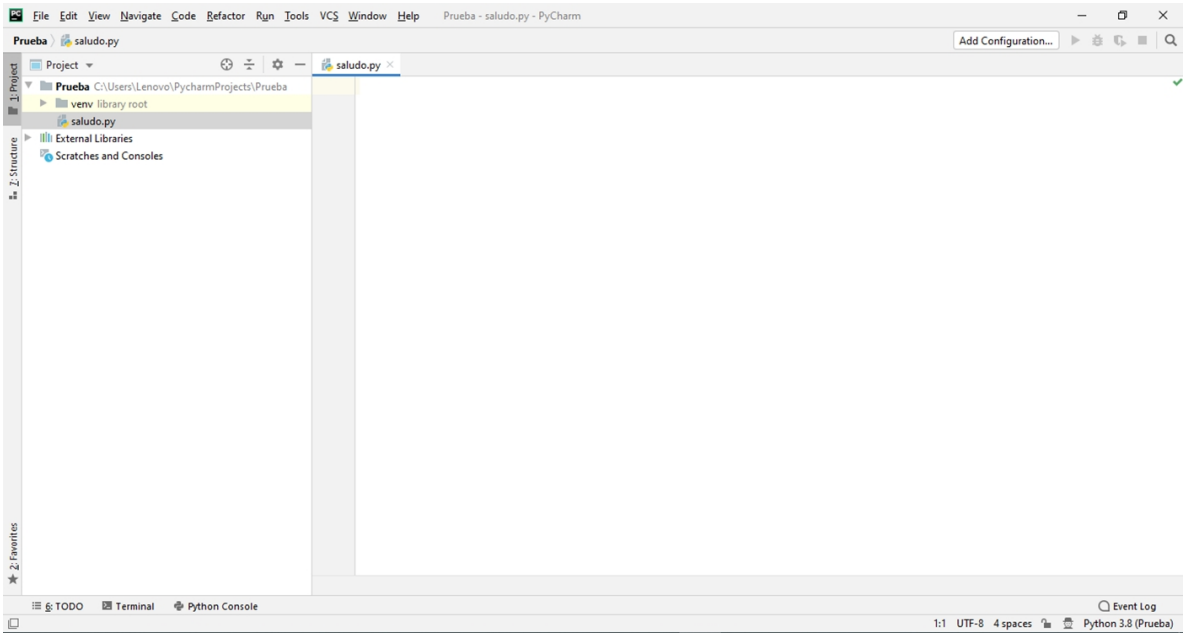


12. Aparecerá otro menú donde debes seleccionar la opción Python File. Asigna como nombre al archivo "saludo".

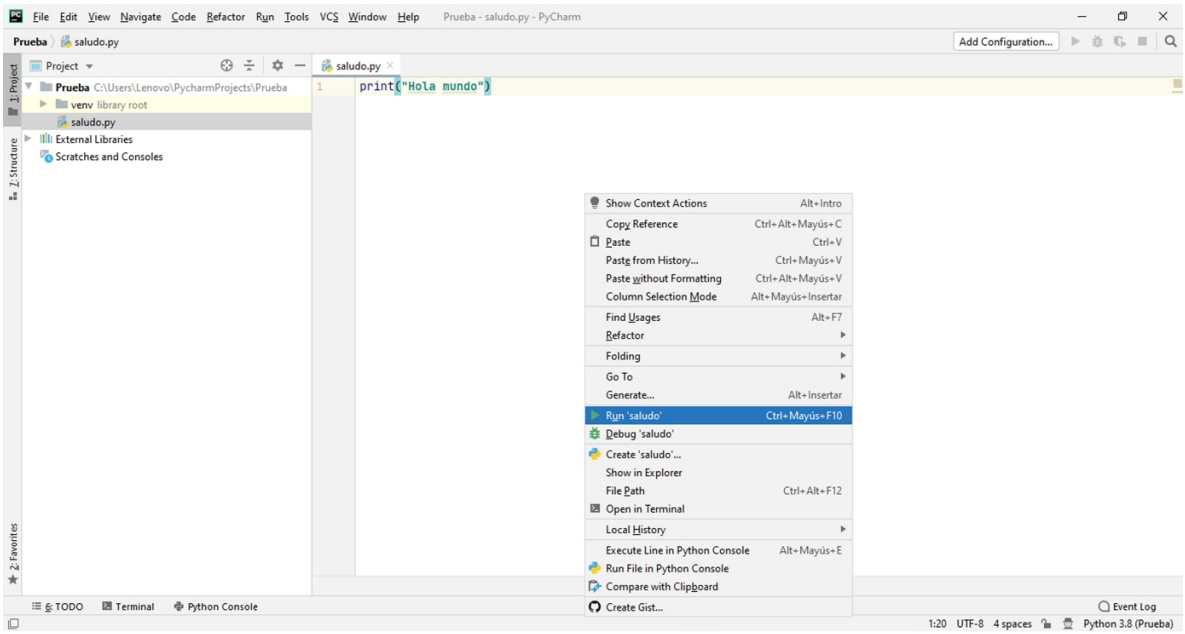


Parte 3: Probar que Python funciona bien con Pycharm

1. Si realizaste los pasos anteriores, entonces tendrás una vista como la siguiente:



2. En el panel principal escribe el comando `print(‘‘Hola mundo’’)`, haz clic derecho en el mismo panel y selecciona la opción Run “saludo”.



3. En el panel inferior se despliega el resultado.

2.7.3. Estructura de un algoritmo en Python

Un algoritmo en Python tiene la misma estructura que hemos presentado para un algoritmo en pseudocódigo. Lo único que cambia son las palabras reservadas y algunos otros signos de sintaxis.

Al igual que en pseudocódigo, un algoritmo empieza con un nombre, que en este caso no forma parte del código de Python y por lo tanto debe ser escrito como comentario.

La estructura de un algoritmo en Python está en un algoritmo principal. Los renglones de comentario deben principiar con un símbolo de numeral #. Para mostrar datos se usa la instrucción `print`. Para recibir datos se tiene la instrucción `input`.

2.7.4. Variables

Las variables en Python son de cuatro tipos: enteras, reales, alfanuméricas y lógicas, para las que se usan las palabras reservadas **int**, **float**, **str** y **bool**, respectivamente.

Cada variable debe tener un nombre, el cual debe empezar con una letra y solamente acepta letras, números y guiones bajos. El nombre de una variable no puede tener acentos ni espacios.

Cada variable del programa consta de tres partes: **nombre**, **tipo** y **valor**. En Python, una variable puede cambiar de tipo dentro del mismo algoritmo. En un principio puede ser entera y más adelante podemos redefinirla como cadena o como real. Por eso decimos que Python usa tipos dinámicos.

Una operación básica de un algoritmo es la asignación. Ésta consiste en asignarle un valor a una variable. En Python para la asignación usamos el signo de igual =, por ejemplo:

```
a=3.14            #Define una variable flotante
b=3               #Define una variable entera
cadena="Juan"     #Define una cadena o string
letra='a'         #Define un caracter
booleano=True     #Define un valor booleano
```

Para saber qué tipo de variable se está empleando usamos la instrucción type de la siguiente manera:

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

Donde vemos que la variable cadena es del tipo string. En las variables a, b, cadena, letra, booleano hemos asignado valores a estas variables. Al mismo tiempo que Python designa el tipo de la variable (con base al valor que asignamos).

2.7.5. Operadores

Los operadores indican las operaciones que deseamos realizar sobre las variables y los datos. Existen cuatro tipos de operadores:

1. Aritméticos
2. Relacionales
3. Lógicos
4. De asignación

Operadores aritméticos

Los operadores aritméticos requieren dos datos para poder efectuarse. La tabla 2.1 proporciona una lista de los operadores aritméticos básicos

Operación	**Operador**	**Ejemplo**
Suma	+	a + b
Resta	-	a - b
Multiplicación	*	a * b
División	/	a / b
División entera	//	a // b
Potencia	**	a ** b

Tabla 2.1: Operadores aritméticos

Las operaciones indicadas dan un resultado del mismo tipo si es que ambos operadores son del mismo tipo. En el caso de que uno sea entero y otro sea real el resultado es real.

Operadores relacionales

Los operadores relacionales aparecen cuando comparamos. La tabla 2.2 describe los operadores relacionales.

Operación	Operador	Ejemplo
Mayor que	>	a >b
Menor que	<	a <b
Mayor o igual que	>=	a >= b
Menor o igual que	<=	a <= b
Igual a	==	a == b
Distinto a	!=	a != b

Tabla 2.2: Operadores relacionales

Operadores lógicos

Los operadores lógicos se usan con variables lógicas (que también reciben el nombre de variables boolenas). En estos operadores ambos datos deben ser datos lógicos. Si a y b son variables lógicas, los operadores lógicos son los que se muestran en la tabla 2.3.

Operación	Operador	Ejemplo
not	not	not b
and	and	a and b
or	or	a or b

Tabla 2.3: Operadores lógicos

Operadores de asignación

Otros operadores de asignación de uso frecuente en Python se muestran en la tabla 2.4.

Python	Operador	Ejemplo
a = b	=	a = b
a = a + 1	+=	a += 1
a = a - 1	-=	a -= 1
a = a * 1	*=	a *= 1
a = a / 1	/=	a /= 1
a = a % 1	%=	a %= 1

Tabla 2.4: Operadores de asignación

En la tabla anterior, la columna 1 equivale a la columna 3 en Python, claro, la notación cambia, pero es lo mismo.

El operador % obtiene el residuo de la división de dos números enteros y se conoce como la función módulo. Solamente se puede usar con números enteros. Por ejemplo, el resultado de 5 % 2 es 1 que es el residuo de 5/2.

2.7.6. Jerarquía de operadores

En Python las expresiones se evalúan normalmente de izquierda a derecha. La excepción son las asignaciones ya que, lógicamente, la parte derecha de una asignación debe resolverse antes de hacer dicha asignación. Dentro de cada expresión, se evalúan las operaciones según la siguiente lista, de arriba abajo:

Operador	**Nombre**	**Nivel jerárquico**
()	Paréntesis	primer nivel
**	Potencia	segundo nivel
* / // %	multiplicación división división entera módulo	tercer nivel
+ -	suma resta	cuarto nivel
>= <= == !=	mayor que menor que mayor o igual que menor o igual que igualdad desigualdad	quinto nivel
"not"	negación	sexto nivel
"and"	y lógica	séptimo nivel
"or"	o lógica	octavo nivel

Tabla 2.5: Jerarquía de operaciones

Ejemplo
¿Cuál es el resultado de la siguiente expresión?
-4 * 7 + 2 ** 3 / 4 – 5

De acuerdo con la jerarquía de operaciones presentada en la tabla anterior, en esta expresión primero se realiza la potencia. Quedándonos de la siguiente manera:
-4 * 7 + 8 / 4 - 5

Seguimos con multiplicaciones y divisiones que tienen igual jerarquía. En este caso, se empieza de izquierda a derecha, así tenemos entonces:
-28 + 2 – 5

Finalmente, realizamos la suma y la resta que tienen igual jerarquía, pero tomando en cuenta que se va a empezar a operar de izquierda a derecha. Quedando el resultado siguiente:

-31

Si quisiéramos representar en Python la siguiente expresión:

$$dato = \frac{a+b}{2} \tag{2.1}$$

Una manera incorrecta de representar esa fórmula en Python es expresarlo de la forma siguiente:

```
dato = a + b / 2
```

Es incorrecto porque por la jerarquía de operaciones, primero se realizaría la división antes que la suma; y en realidad queremos que primero se haga la suma y después se divida sobre 2. La forma correcta de expresar la fórmula anterior es:

```
dato =( a + b )/2
```

2.7.7. Palabras reservadas

Las palabras reservadas o keywords de Python son aquellos nombres que incorpora el lenguaje los cuales no pueden ser reemplazados por un valor determinado (funciones, clases, variables) o alterar su funcionamiento. Por lo tanto, la lista de keywords en Python 3 resulta ser la siguiente.

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

Tabla 2.6: Palabras reservadas

2.7.8. Comentarios

En Python existen dos maneras de escribir líneas de comentarios. La primera es usando el símbolo de numeral #. El comentario empieza a partir del símbolo, por ejemplo desde el inicio:

```
# Este es un reglón de comentario.
```

o al final de una expresión de Python:

```
a = b + c #Ejemplo de suma
```

Lo que aparece después del símbolo de numeral # no se ejecuta.

La segunda manera es por lo general más usada cuando hay varias líneas de comentarios. El párrafo de comentarios abre con tres apóstrofos ''' y se termina con tres apóstrofos ''', por ejemplo:

```
'''Este es un conjunto de líneas
de comentarios que usan la forma
alternada de indicar comentarios.'''
```

2.8. Conclusiones

En este capítulo abordamos las bases del diseño de algoritmos, así como su representación en pseudocódigo. De igual forma, se presentaron los conceptos básicos del lenguaje python.

Capítulo 3

Estructuras secuenciales

3.1. Introducción

Para la solución de cualquier problema que se vaya a representar mediante pseudocódigo o código de Python, siempre tendremos que representar mediante letras, abreviaciones o palabras completas los elementos que intervienen en el proceso de solución. A estos elementos se les denomina variables o constantes. Por ejemplo: `precio`, `horas_trabajadas`, `edad`, etc. Es importante mencionar que, por convención las variables de un algoritmo no se acentúan.

Con base en esto, para facilitar la lectura de un algoritmo se recomienda crear una tabla donde se declaran las variables que se utilizarán y sus características o tipo, tal y como se muestra en la tabla 3.1, que muestra las variables que se utilizarían para obtener el área de un cuadrado.

Nombre de la variable	**Descripción**	**Tipo**
lado	Lado del cuadrado	Real, Float
area	Área del cuadrado	Real, Float

Tabla 3.1: Declaración de las variables que se utilizarán para obtener el área de un cuadrado.

Como se puede ver en la tabla 3.1, se utilizará la variable `lado` para representar el lado del cuadrado, a la cual se les podrán asignar diferentes valores. Al utilizar esos valores y aplicar la fórmula correspondiente se podrá obtener el área del cuadrado, la cual es asignada a la variable denominada `area`. Además, se describe que esas variables son de tipo real, lo cual implica que podrán tomar valores fraccionarios, pero también pudieron haber sido enteras.

3.2. Estructuras de control

Sin importar qué herramienta o técnica se utilice para la solución de un problema dado, ésta tendrá una estructura, que se refiere a la secuencia en que se realizan las operaciones o acciones para resolver el problema; esas estructuras pueden ser: secuenciales, de selección y de repetición, las cuales se analizarán en su momento.

Debe tenerse presente que la solución de un problema dado mediante el uso de una computadora es un sistema, por lo que debe tener una entrada de datos que serán procesados para obtener una salida, que es la solución o información que se busca. En la figura 3.1 se muestra el esquema de un sistema que transforma los datos en información mediante un proceso.

Figura 3.1: Sistema de transformación

3.3. Estructuras secuenciales

En este tipo de estructura las instrucciones se realizan o se ejecutan una después de la otra (en orden) y, por lo general, se espera que se proporcione uno o varios datos, los cuales son asignados a variables para que con ellos se produzcan los resultados que representen la solución del problema que se planteó. Los algoritmos tienen como fin actuar sobre los datos proporcionados por el usuario, a los que se les aplican procesos con el objetivo de generar la información o un resultado. El algoritmo es realmente la representación funcional de un sistema, como el que se muestra en la figura 3.1.

Para resolver un problema mediante la utilización de cualquier herramienta es necesario entender y establecer con qué datos se cuenta, los procesos que se deben realizar y la secuencia apropiada para obtener la solución que se desea.

3.3.1. Ejemplo suma de dos números

Se desea implementar un algoritmo para obtener la suma de dos números cualesquiera. Para poder obtener la suma es necesario contar con dos números, pues el proceso que debemos realizar es precisamente la suma de éstos, la cual se asigna a una variable que se reporta como resultado del proceso.

Los pasos por seguir son los mostrados en el pseudocódigo 1, que corresponde al

algoritmo que permite determinar la suma de dos números cualesquiera.

Algoritmo 1: Suma de dos números

Entradas: a (entero), b (entero)
Salidas: suma (entero)
1: Inicio
2: Leer a
3: Leer b
4: suma=a+b
5: Escribir suma
6: Fin

Al representar la solución del problema con pseudocódigo, se está utilizando un lenguaje común, sólo que de una forma ordenada y precisa.

Es recomendable indicar mediante una tabla las variables que se utilizan, señalando lo que representan y sus características, esta acción facilitará la lectura de la solución de un problema dado. Para el problema de la suma de dos números, la tabla 3.2 muestra las variables utilizadas en la solución.

Nombre de la variable	Descripción	Tipo
a	Primer número a sumar	Entero, Int
b	Segundo número a sumar	Entero, Int
suma	Resultado de la suma	Entero, Int

Tabla 3.2: Variables utilizadas para determina la suma de dos números cualesquiera

La representación del algoritmo en Python sería como el que se muestra en el programa 3.1.

```
1 a=int(input("Introduce el primer valor a sumar: "))
2 b=int(input("Introduce el segundo valor a sumar: "))
3 suma=a+b
4 print(f"La suma es: {suma}")
```

Programa 3.1: Suma de dos enteros

Cuando se ejecuta el programa se tiene como salida:

```
Introduce el primer valor a sumar: 12
Introduce el segundo valor a sumar: 20
La suma es: 32
```

Entradas en Python

La función `input` nos permite leer los datos que se introducen mediante el teclado. Dicha función recibe como argumento una cadena, la cual le indica al usuario lo que debe realizar. En el programa 3.1, los argumentos de la función `input` son las cadenas: "Introduce el primer valor a sumar: " e "Introduce el segundo valor a sumar: ". La función `input` devuelve una cadena, que corresponde al dato que el usuario introduce.

Para este ejemplo el valor devuelto por `input` no se puede asignar directamente a las variables `a` y `b`, ya que como dijimos anteriormente, dichos valores son cadenas. Entonces, lo que se tiene que hacer es convertir las cadenas a enteros mediante la función `int`, la cual recibe como argumento el dato que queremos convertir a entero. Con la conversión realizada procedemos a almacenar los datos proporcionados por el usuario en sus respectivas variables, quedando de la siguiente forma:

```
a=int(input(‘‘Introduce el primer valor a sumar: ’’))
b=int(input(‘‘Introduce el segundo valor a sumar: ’’))
```

Salidas en Python

Una vez que la información introducida por el usuario fue procesada, es necesario mostrarla de alguna forma. Al proceso de mostrar dicha información se le denomina salida. En Python la salida estándar se muestra en la pantalla de la computadora, y dicha salida es mostrada gracias a la función `print`, la cual recibe como argumento la información que se quiere mostrar.

Con base en el programa 3.1, tenemos que la salida queda de la siguiente forma:

```
print(f‘‘La suma es: {suma}’’)
```

Analicemos como funciona `print`. Primero, vemos que recibe como argumento la cadena `f‘‘La suma es: {suma}’’`. Donde suma es la variable que almacenó la suma de los dos números introducidos por el usuario. La `f` y las llaves {} que están en la cadena permiten que se puedan imprimir los valores de las variables. Esto quiere decir que si, por ejemplo, en `print` se hubiera puesto como argumento la cadena `‘‘La suma es: suma’’`, en lugar de mostrar el valor de la variable suma, se hubiera mostrado la palabra suma. Entonces, es indispensable poner la f antes de introducir la cadena y usar las llaves para incluir el valor de la variable que se quiere mostrar.

3.3.2. Ejemplo área y perímetro de un rectángulo

Se requiere conocer el área y el perímetro de un rectángulo. Realiza un algoritmo para tal fin y represéntalo mediante pseudocódigo para realizar este proceso.

Como se sabe, para poder obtener el área del rectángulo, primero se tiene que conocer la base y la altura, y una vez obtenidas se calcula el resultado.

La tabla 3.3 muestra las variables que se van a utilizar para elaborar el algoritmo correspondiente.

Nombre de la variable	Descripción	Tipo
base	Base del rectángulo	Real, Float
altura	Altura del rectángulo	Real, Float
area	Área del rectángulo	Real, Float
perimetro	Perímetro del rectángulo	Real, Float

Tabla 3.3: Área y perímetro de un rectángulo

Fórmulas para calcular área y perímetro del rectángulo:
Área=base*altura
Perímetro=2*base+2*altura

La estructura del pseudocódigo 2 muestra el algoritmo que permite obtener el área y el perímetro del rectángulo.

Algoritmo 2: Rectángulo

Entradas: base (real), altura (real)
Salidas: area (real), perimetro (real)

1: Inicio
2: Leer base
3: Leer altura
4: area=base*altura
5: perimetro=2*base+2*altura
6: Escribir perimetro
7: Escribir area
8: Fin

La representación del algoritmo en Python sería como el que se muestra en el programa 3.2.

```
#Entradas
base=float(input("Introduce la base: "))
altura=float(input("Introduce la altura: "))
#Proceso
area=base*altura
perimetro=2*base+2*altura
#Salidas
print(f"El área del rectángulo es: {area}")
print(f"El perímetro del rectángulo es: {perimetro}")
```

Programa 3.2: Rectángulo

Cuando ejecutamos el programa, tenemos como salida:

```
Introduce la base: 15
Introduce la altura: 7
El área del rectángulo es: 105.0
El perímetro del rectángulo es: 44.0
```

La función float

En el apartado **Entradas en Python** de este capítulo, ya habíamos explicado el uso de la función `int`, la cual recibe como argumento un elemento (por ejemplo, una cadena, un valor real, etc.) y convierte ese elemento a entero. La función `float` tiene el mismo objetivo, que es convertir un elemento que recibe como argumento. Cabe señalar que dicha conversión será a un valor real.

Dado el análisis que se realizó en el pseudocódigo 2, se llega a la conclusión de que la base y la altura del rectángulo son de tipo real. Por lo que en programa 3.2 se utiliza la función `float`, para convertir a real los valores que el usuario introduzca.

3.3.3. Bibliotecas

Muy a menudo, Python requiere usar funciones que vienen en bibliotecas (el concepto es similar a una biblioteca de la vida real, por ejemplo, si se necesita una información vamos a la biblioteca y consultamos el libro que tiene dicha información). Estas bibliotecas deben llamarse antes de usar las funciones de ella. Por ejemplo, las funciones trigonométricas, la raíz cuadrada, los logaritmos, etc., vienen incluidos en la biblioteca de funciones matemáticas denominada `math`. Para poder usar sus funciones en nuestro programa usamos al principio del programa lo siguiente:

```
import math
```

Y para usar las funciones usamos, por ejemplo, para obtener el coseno de π:

```
math.cos(math.pi)
```

Lo que da como resultado:

-1.0

Donde hemos usado la función `cos` y la constante `pi` que son parte de la biblioteca `math`. Otra forma de importar las funciones de una biblioteca es usando:

```
from math import *
```

Al hacer esto ya no necesitamos escribir el nombre de la biblioteca cada vez que usamos una función. Ya simplemente escribimos la función como en:

```
cos(pi)
```

Algunas funciones de la biblioteca `math` se dan en la siguiente tabla:

Función	**Ejemplo Python**	**Notación matemática**
Valor absoluto	`math.abs(-5)`	$\|-5\|$
Función exponencial	`math.exp(1)`	e^x
Potencia a^b	`math.pow(2,3)`	2^3
Raíz cuadrada	`math.sqrt(9.0)`	$\sqrt{9.0}$
Coseno de un ángulo en radianes	`math.cos(0.7)`	$cos(0.7)$
Seno de un ángulo en radianes	`math.sin(0.707)`	$sen(0.707)$
Tangente de un ángulo en radianes	`math.tan(1)`	$tan(1)$
Obtención de valor de pi	`math.pi`	π
Obtención de valor de e	`math.e`	e
...	...	...

3.3.4. Ejemplo área de una circunferencia

Se requiere obtener el área de una circunferencia. Realizar el algoritmo correspondiente y representarlo mediante pseudocódigo y código en Python.

Nombre de la variable	**Descripción**	**Tipo**
radio	Radio de la circunferencia	Real, Float
pi	El valor de 3.141...	Real, Float
area	Área de la circunferencia	Real, Float

Tabla 3.4: Área de la circunferencia

Fórmulas para calcular área de la circunferencia: $area = PI * radio^2$

De igual forma que en los problemas anteriores, es importante establecer la tabla de variables que se utilizarán para la solución del problema, pero previamente se analizará qué se requiere para obtener el área de la circunferencia.

Algoritmo 3: Área circunferencia

Entradas: radio (real)
Salidas: area (real)
1: Inicio
2: Leer radio
3: area=pi*radio^2
4: Escribir area
5: Fin

Si se analiza la fórmula que se utiliza para tal fin, se puede establecer que se requiere solamente un valor de radio y que se debe dar un valor constante, que es el valor de PI (3.14...). Con esto se puede establecer la tabla 3.4 con las variables correspondientes.

La estructura del pseudocódigo 3 muestra el algoritmo que permite obtener el área de la circunferencia.

La representación del algoritmo en Python sería como el que se muestra en el programa 3.3.

```
1 from math import pi
2
3 radio=float(input("Introduce el radio de la circunferencia: "))
4 area=pi*(radio**2)
5 print(f"El área de la circunferencia es: {area}")
```

Programa 3.3: Circunferencia

Cuando ejecutamos el programa, tenemos como salida:

```
Introduce el radio de la circunferencia: 10
El área de la circunferencia es: 314.1592653589793
```

Previamente se mencionó el uso de la biblioteca `math`, la cual se incorpora a este programa. Para este ejemplo en particular solo se necesita el uso de la constante `pi`, por ese motivo es lo único que se importa de `math` (ver la primera línea del programa). Si se hubieran requerido más funcionalidades de `math`, entonces hubiera sido necesario usar la línea siguiente al inicio del programa.

```
from math import *
```

3.3.5. Ejemplo intercambio de variables

Se requiere intercambiar el valor de dos variables, que almacenan valores introducidos por el usuario. Por ejemplo, si `a=10` y `b=20`, entonces el resultado debe ser `a=20` y `b=10`. Realizar el algoritmo correspondiente y representarlo mediante pseudocódigo y código en Python.

Nombre de la variable	**Descripción**	**Tipo**
a	Almacena el primer valor	Entero, Int
b	Almacena el segundo valor	Entero, Int

Tabla 3.5: Variables

Primero, establecemos nuestra tabla de variables que utilizaremos para la solución del problema.

El pseudocódigo 4 muestra el algoritmo que permite hacer el cambio de valores de las variables.

Algoritmo 4: Intercambia variables

Entradas: a (entero), b (entero)
Salidas: a (con el nuevo valor), b (con el nuevo valor)

1: Inicio
2: Leer a
3: Leer b
4: copia=a
5: a=b
6: b=copia
7: Escribir a
8: Escribir b
9: Fin

Se leen los valores de `a` y `b`, posteriormente se saca una copia a la variable `a` (línea 4 del pseudocódigo 4). Esta copia es indispensable porque en la línea 5 se sobreescribe el valor de `a` con el valor de `b`. Eso quiere decir que tanto `a` como `b` tienen el mismo valor. Sin embargo, como ya se había guardado el valor original de `a` (en la variable `copia`), simplemente se asigna este valor a la variable `b` (línea 6). Finalmente, se escriben los valores intercambiados de `a` y `b`.

El programa 3.4 es la codificación en Python del algoritmo anterior:

```
1  #Entradas
2  a=int(input("Introduce el valor de a: "))
3  b=int(input("Introduce el valor de b: "))
4  #Proceso
5  copia=a
6  a=b
7  b=copia
8  #Salidas
9  print(f"El nuevo valor de a es: {a}")
10 print(f"El nuevo valor de b es: {b}")
```

Programa 3.4: Intercambia variables

Cuando ejecutamos el programa, tenemos como salida:

```
Introduce el valor de a: 10
Introduce el valor de b: 20
El nuevo valor de a es: 20
El nuevo valor de b es: 10
```

3.3.6. Ejemplo venta de pozol

Don Manuel hace masa de pozol[1], la cual vende a una pozolería de la ciudad. Este señor lleva su registro en gramos de la masa que produce, pero cuando entrega le pagan $ 25 por kilogramo. Realiza un algoritmo, representalo en pseudocódigo y escribe un

[1]El pozol está hecho con masa de maíz y cacao molido

programa en python que ayude a don Manuel a saber cuánto recibirá por la producción de un día.

Si se analiza el problema se puede establecer que los datos que se necesitan para resolver el problema son los que se muestran en la tabla 3.6.

Nombre de la variable	**Descripción**	**Tipo**
gramos_producidos_producidos	Cantidad de gramos producidos	Real, Float
kilos	Cantidad de kilos que produce	Real, Float
pago_recibido	Ganancia por la entrega de la masa	Real, Float

Tabla 3.6: Variables

El pseudocódigo 5 muestra el algoritmo que permite a don Manuel saber cuánto recibirá por la entrega de su producción.

Algoritmo 5: Producción de masa de pozol

Entradas: gramos_producidos (real)
Salidas: pago_recibido (real)
1: Inicio
2: Leer gramos_producidos
3: kilos = gramos_producidos * 0.001
4: pago_recibido = kilos * 25
5: Escribir pago_recibido
6: Fin

El programa 3.5 es la codificación en Python del algoritmo anterior:

```
1 gramos_producidos=float(input("Escribe los gramos de pozol: "))
2 kilos=gramos_producidos*0.001
3 pago_recibido=kilos*25
4 print(f"La ganancia de la masa de pozol es: {pago_recibido:.2f}")
```

Programa 3.5: Producción de pozol

Para este programa solo se desea mostrar dos decimales, por esa razón después de la variable `pago_recibido` (última línea del programa) se incluye la notación `:.2f`. Si se requirieran más decimales solo se cambia el 2 por el valor de decimales que se necesitan.

Como salida se obtiene lo siguiente:

```
Escribe los gramos de pozol: 100000
La ganancia de la masa de pozol es: 2500.00
```

3.3.7. Ejemplo sueldo semanal

Se requiere determinar el sueldo semanal de un trabajador con base en las horas que trabaja y el pago por hora que recibe. Realiza el pseudocódigo y el programa en Python que representen el algoritmo de solución correspondiente.

Para obtener la solución de este problema es necesario conocer las horas que labora cada trabajador y cuánto se le debe pagar por cada hora de trabajo, con base en esto se puede determinar que las variables que se requieren utilizar son las que se muestran en la tabla 3.7.

Nombre de la variable	**Descripción**	**Tipo**
horas_trabajadas	Horas trabajadas	Real, Float
pago_hora	El pago por hora	Real, Float
sueldo_semanal	Sueldo semanal	Real, Float

Tabla 3.7: Variables sueldo semanal

El pseudocódigo 6 muestra el algoritmo que permite determinar el sueldo semanal de un trabajador.

Algoritmo 6: Sueldo semanal

Entradas: horas_trabajadas (real), pago_hora (real)
Salidas: sueldo_semanal (real)
1: Inicio
2: Leer horas_trabajadas
3: Leer pago_hora
4: sueldo_semanal = horas_trabajadas * pago_hora
5: Escribir sueldo_semanal
6: Fin

El siguiente programa es la codificación en Python del algoritmo 6.

```
1 horas_trabajadas=float(input("Introduce las horas trabajadas: "))
2 pago_hora=float(input("Introduce el pago por hora: "))
3
4 sueldo_semanal=horas_trabajadas*pago_hora
5
6 print(f"El sueldo semanal del trabajador es de: {sueldo_semanal:.2f}")
```

Programa 3.6: Intercambia variables

El programa 3.6 tiene como salida:

```
Introduce las horas trabajadas: 23
Introduce el pago por hora: 34.765
El sueldo semanal del trabajador es de: 799.60
```

3.4. Conclusiones

En este capítulo se abordaron las estructuras secuenciales. A través de diversos ejemplos (con una dificultad gradual) mostramos su representación tanto en pseudocódigo como en lenguaje python.

3.5. Ejercicios

1. Se tienen tres variables A, B y C. Escribir las instrucciones necesarias para intercambiar sus valores entre sí del modo siguiente:

   - B toma el valor de A
   - C toma el valor de B
   - A toma el valor de C

2. Don Manuel además de producir masa de pozol, produce agua del mismo producto, la cual vende a un comerciante de la ciudad. Manuel lleva su registro de producción en galones, pero su pago lo recibe en litros (1 gal = 3.79 L). Realiza un algoritmo, representalo en pseudocódigo y escribe un programa en python que ayude a don manuel a saber cuánto recibirá por la entrega de su producción.

3. Una empresa importadora desea determinar cuántos dólares puede adquirir con cierta cantidad de dinero mexicano. Realiza el pseudocódigo y el programa en Python.

Capítulo 4

Estructuras selectivas

4.1. Introducción

Las estructuras secuenciales vistas en el capítulo anterior no abordan situaciones en las cuales se debe elegir un camino u otro. Por ejemplo, ¿qué pasa si se requiere un algoritmo para saber si una persona es mayor o menor de edad? Con una estructura secuencial resolver este tipo de cuestionamiento sería una tarea muy difícil de realizar, ya que este tipo de preguntas obligan al algoritmo a tomar decisiones. Es ahí donde entran las estructuras selectivas, las cuales a partir de una condición siguen un flujo de instrucciones u otro.

4.2. Estructuras selectivas

Existen dos tipos de estructuras selectivas: simples y compuestas. La estructura de una selectiva simple tiene la forma:

Algoritmo 7: Estructura selectiva simple

```
1: si condicion es verdadera entonces
2:     Hacer acción 1
3: fin_si
```

Como se muestra en la estructura selectiva simple, la condición es una expresión booleana que es evaluada como verdadero o falso. Si es verdadero, se realizan las acciones que se encuentren dentro de la estructura, que son aquellas que están después de la palabra clave "**si**", y antes de "**fin_si**". En el caso del algoritmo 7, se refiere a la línea 2, pero es posible que pueda ser un bloque de instrucciones (ver línea 2 de algoritmo 7).

Ejemplo: Se requiere determinar si una persona es mayor de edad.

Algoritmo 8: Selectiva simple ejemplo

Entradas: edad
Salidas:

```
1: Leer edad
2: si edad>=18 entonces
3:     Escribir "Eres mayor de edad"
4: fin_si
```

El programa 4.1 muestra la codificación en Python del algoritmo para saber si una persona es mayor de edad. Primero, se pide al usuario que introduzca su edad, la cual es almacenada en la variable edad. Posteriormente, mediante una selectiva simple (if) analizamos si la edad es mayor o igual a 18 (condición). Si edad es mayor o igual a 18 se imprime una cadena con el texto "Eres mayor de edad". Es importante mencionar que, en los programas de Python no existe una marca de termino de la estructura selectiva, como en el caso de su representación algorítmica que se usa la palabra "**fin_si**"

```
1 edad=int(input("Escribe tu edad: "))
2
3 if edad>=18:
4     print("Eres mayor de edad...")
```

Programa 4.1: Mayor de edad

El programa 4.1 tiene como salida:

```
Escribe tu edad: 24
Eres mayor de edad...
```

Por su parte, una estructura selectiva compuesta tiene la forma:

Algoritmo 9: Estructura selectiva compuesta

```
1: si condicion es verdadera entonces
2:     Hacer acción 1
3: de_lo_contrario
4:     Hacer acción 2
5: fin_si
```

Ejemplo: Determinar si una persona es mayor o menor de edad.

Algoritmo 10: Selectiva compuesta ejemplo

Entradas: edad
Salidas:

```
1: Leer edad
2: si edad>=18 entonces
3:     Escribir "Eres mayor de edad"
4: de_lo_contrario
5:     Escribir "Eres menor de edad"
6: fin_si
```

En la estructura selectiva compuesta si la condición es verdadera se realizan las acciones dentro del si. Si no es verdadera, se realizan las acciones que se encuentran en el **de_lo_contrario**.

El programa 4.2 muestra la codificación en Python del algoritmo para saber si una persona es mayor o menor de edad. Primero se pide al usuario que introduzca su edad, la cual es almacenada en la variable edad. Posteriormente, mediante una selectiva compuesta (if-else) analizamos si la edad es mayor o igual a 18 (mayoría de edad en México). Si edad es mayor o igual a 18 se imprime una cadena con el texto "Eres mayor de edad". En caso contrario, se imprime la cadena "Eres menor de edad".

```
1 edad=int(input("Escribe tu edad: "))
2
3 if edad>=18:
4     print("Eres mayor de edad...")
5 else:
6     print("Eres menor de edad...")
```

Programa 4.2: Mayor y menor de edad

El programa 4.2 tiene como salida:

```
Escribe tu edad: 12
Eres menor de edad...
```

4.2.1. Ejemplo: "CompuMax"

La tienda de computadoras "CompuMax" tiene una promoción: a todas las computadoras con un precio superior a $15,500.00 se les aplicará un descuento de 16 %. A todos los demás equipos se les aplicará sólo el 7 %. Diseñar un algoritmo que indique el precio final que debe pagar un cliente por la compra de un equipo, así como el descuento realizado. Representa dicho algoritmo como pseudocódigo y código en python.

El pseudocódigo 11 representa el algoritmo para determinar el precio que debe pagar un cliente por un equipo, así como su descuento.

Algoritmo 11: Ejemplo CompuMax

Entradas: costo_equipo (real)
Salidas: descuento (real), precio_final (real)

```
1: Inicio
2: Leer costo_equipo
3: si costo_equipo>15500 entonces
4:     descuento=costo_equipo * 0.16
5: de_lo_contrario
6:     descuento=costo_equipo * 0.07
7: fin_si
8: precio_final=costo_equipo-descuento
9: Escribir precio_final
10: Escribir descuento
11: Fin
```

A partir del pseudocódigo 11 se define la tabla de variables 4.1 para solucionar el problema.

Nombre de la variable	Descripción	Tipo
costo_equipo	Precio del equipo de Cómputo	Real, Float
descuento	Descuento a aplicar al equipo	Real, Float
precio_final	Precio final del equipo	Real, Float

Tabla 4.1: Variables del ejemplo CompuMax

El código en Python que da solución a este ejercicio se muestra a continuación:

```
1  costo_equipo=float(input("Introduce el costo del equipo: "))
2
3  if costo_equipo>15500:
4      descuento=costo_equipo*0.16
5  else:
6      descuento=costo_equipo*0.07
7
8  precio_final=costo_equipo-descuento
9  print(f"El precio a pagar es: {precio_final}")
10 print(f"El descuento es: {descuento}")
```

Programa 4.3: CompuMax

El programa 4.5 tiene como salida:

```
Introduce el costo del equipo: 23456
El precio a pagar es: 19703.04
El descuento es: 3752.96
```

4.3. Estructuras if-elif-else

En este capítulo hasta el momento se han abordado algoritmos que pueden tomar sólo dos caminos (posibilidades), sin embargo, algunos problemas requieren más caminos por analizar. Es aquí donde entran las estructuras if-elif-else, las cuales son una extensión de las estructuras selectivas compuestas.

Ejemplo: se requiere saber si un número es positivo, negativo o cero.

El pseudocódigo 12 nos muestra la solución a este ejercicio. Primero se lee el número que se va a procesar. Posteriormente, se inicia con la estructura **si-delocontrarioSi-delocontrario**, donde primero se evalúa si `numero` es mayor que cero. Si el número es mayor que cero, entonces se escribe el texto **‘‘`el número es positivo`’’**, y terminamos. En caso contrario, pregunta si `numero` es igual a cero, si esto es verdadero, entonces se escribe el texto **‘‘`El número es cero`’’** y finalizamos. Si `numero` no es mayor que cero ni es cero, entonces se va al último caso donde se imprime el texto **‘‘`El número es`**

`negativo''` y terminamos.

Algoritmo 12: Positivo, negativo o cero

Entradas: numero (entero)
Salidas:

```
1: Inicio
2: Leer numero
3: si numero>0 entonces
4:     Escribir "El número es positivo"
5: de_lo_contrario si numero==0 entonces
6:     Escribir "El número es cero"
7: de_lo_contrario
8:     Escribir "El número es negativo"
9: fin_si
10: Fin
```

El código en Python muestra el funcionamiento de la estructura **if-elif-else** (si-delocontrarioSi-delocontrario, en pseudocódigo). Primero se lee el número que se va a analizar y se evalúa, si es mayor a cero se imprime el texto `''El número es positivo''` y terminamos, si no, si `numero` es igual a cero se imprime `''El número es cero''` y termina. De lo contrario (si no es mayor que cero ni es cero), se escribe el texto `''El número es negativo''` y fin.

```
numero = int(input("Escribe un número: "))

if numero > 0:
    print("El número es positivo")
elif numero == 0:
    print("El número es cero")
else:
    print("El número es negativo")
```

Programa 4.4: Positivo negativo o cero

El programa 4.6 tiene como salida:

```
Escribe un número: 0
El número es cero
```

4.3.1. Ejemplo: Calculadora básica

Diseñar un algoritmo el cual dados dos números (distintos de cero) y una operación (es decir: suma, resta, multiplicación o división) muestre el resultado de operar dichos números. La operación se indicará proporcionando el primer carácter de su nombre, por ejemplo, si se requiere hacer una suma, proporcionar S o s. Representa el algoritmo mediante pseudocódigo y programa en python.

Procedemos a diseñar nuestro pseudocódigo (ver pseudocódigo 13). Primero leemos los valores para operar (`num1` y `num2`), así como la operación por realizar con ellos (suma, resta, multiplicación o división). El carácter almacenado en la variable operación

se convierte a mayúsculas, con la finalidad de que el usuario pueda escribir `S` o `s` en el caso de la suma; y así para cualquier otra operación. Evidentemente también dicha operación se pudo convertir a minúsculas.

Algoritmo 13: Calculadora

Entradas: num1 (real), num2 (real)
Salidas: resultado (real)

```
1: Inicio
2: Leer num1
3: Leer num2
4: Leer operacion
5: Convertir operacion a mayúsculas
6: si operacion==‘S’ entonces
7:    resultado=num1+num2
8:    Escribir resultado
9: de_lo_contrario si operacion==‘R’ entonces
10:   resultado=num1-num2
11:   Escribir resultado
12: de_lo_contrario si operacion==‘P’ or operacion== ‘M’  entonces
13:   resultado=num1*num2
14:   Escribir resultado
15: de_lo_contrario si operacion==‘D’ entonces
16:   resultado=num1/num2
17:   Escribir resultado
18: de_lo_contrario
19:   Escribir “Operación incorrecta”
20: fin_si
21: Fin
```

A continuación, procedemos a identificar las variables que utilizaremos para resolver este ejercicio. Para este fin, la tabla 4.2 describe dichas variables.

Nombre de la variable	**Descripción**	**Tipo**
num1	Primer número a operar	Real, Float
num2	Segundo número a operar	Real, Float
operacion	Operación por realizar	Cadena, String
resultado	Almacena el resultado	Real, Float

Tabla 4.2: Variables del ejemplo calculadora

Ahora, mediante la estructura `si-delocontrarioSi-delocontrario`, se procede a evaluar los posibles casos. Si operación es “S”, entonces en la variable `resultado` se almacena el valor de la suma de los números leídos al inicio y se imprime el resultado. De lo contrario, si operación es “R”, la variable `resultado` almacena la resta de `num1` y `num2` e imprime el resultado. De lo contrario, si operación es P o M, realiza la multiplicación

de ambos números, almacena el resultado en la variable `resultado` e imprime dicha variable. De lo contrario, si operación es "D", realiza la división de `num1` sobre `num2`, almacena el resultado y lo imprime. Si operación no es ninguno de los casos descritos anteriormente, entra al último caso e imprime el texto "Operación incorrecta".

El código en Python de este ejercicio se describe a continuación:

```
num1 = float(input("Escribe el primer número: "))
num2 = float(input("Escribe el segundo número: "))
operacion = input("Escribe la operación a realizar: ").upper()

if operacion=='S':
  resultado=num1+num2
  print(f"El resultado es: {resultado:.2f}")
elif operacion=='R':
  resultado=num1-num2
  print(f"El resultado es: {resultado:.2f}")
elif operacion=='P' or operacion=='M' :
  resultado=num1*num2
  print(f"El resultado es: {resultado:.2f}")
elif operacion=='D':
  resultado=num1/num2
  print(f"El resultado es: {resultado:.2f}")
else:
  print("Operación incorrecta")
```

Programa 4.5: Calculadora

El programa 4.7 tiene como salida:

```
Escribe el primer número: 5
Escribe el segundo número: 6
Escribe la operación a realizar: h
Operación incorrecta
```

4.3.2. Ejemplo: Alitas y más

Alitas y más es una empresa 100 % mexicana que vende comida rápida. Los fines de semana tienen una promoción para sus clientes. Esta promoción es la siguiente: cada alita de pollo cuesta $ 10 pero si el cliente compra más de 19 pero menos de 30, cada alita le sale en $ 7; si el cliente compra 30 alitas o más paga $ 5 por cada una. Realiza un algoritmo y representalo mediante pseudocódigo y código en python, para ayudar a esta empresa a ver cuánto debe cobrar al cliente por su pedido.

El pseudocódigo 14 representa la solución a este ejercicio. Primero se necesita saber la cantidad de alitas que comprará el cliente, este paso se realiza en la línea 2. Posteriormente, mediante una estructura si-delocontrarioSi-delocontrario, procedemos a evaluar los posibles casos que se pueden presentar. El primer caso (línea 4), nos dice si el número de alitas está entre 1 y 19, el costo de cada alita es de $ 10. De lo contrario, si el número de alitas es superior a 19 pero menor o igual a 29 (línea 6), entonces el costo de cada alita es de $ 7. En caso contrario (línea 8), si el número de alitas es mayor de 30 el costo de cada alita es de $ 5. Finalmente, si el número de alitas no cae en ningún

caso (por ejemplo, se digita un valor negativo), entonces se notifica que se ha producido un error.

Algoritmo 14: Alitas y más

Entradas: can_alitas (Entero)
Salidas: pago_total (Entero)

```
1: Inicio
2: Leer can_alitas
3: costo_alita=0
4: si can_alitas>=1 and can_alitas<=19 entonces
5:    costo_alitas=10
6: de_lo_contrario si can_alitas>19 and can_alitas<=29 entonces
7:    costo_alitas=7
8: de_lo_contrario si can_alitas>=30 entonces
9:    costo_alita=5
10: de_lo_contrario
11:    Escribir "Error: cantidad de alitas incorrecto"
12: fin_si
13: pago_total=costo_alita*can_alitas
14: Escribir pago_total
15: Fin
```

Procedemos a establecer la tabla de variables (ver tabla 4.3).

Nombre de la variable	Descripción	Tipo
can_alitas	Cantidad de alitas	Entero, Int
costo_alita	Costo de cada alita	Real, Float
pago_total	El pago por el pedido de alitas	Real, Float

Tabla 4.3: Variables del ejemplo "Alitas y más"

El programa en Python 4.8 es la solución al este ejercicio:

```python
can_alitas=int(input("Introduce la cantidad de alitas a vender: "))

costo_alita=0
if can_alitas > 0 and can_alitas <= 19:
    costo_alita=10
elif can_alitas > 19 and can_alitas <= 29:
    costo_alita=7
elif can_alitas >= 30:
    costo_alita=5
else:
    print("Error: cantidad de alitas incorrecto...")

pago_total=costo_alita*can_alitas
print(f"La cantidad que debe pagar el cliente es: {pago_total:.2f}")
```

Programa 4.6: Alitas y más

El programa 4.8 tiene como salida:

```
Introduce la cantidad de alitas a vender: 40
La cantidad que debe pagar el cliente es: 200.00
```

4.4. Conclusiones

En este capítulo se abordaron las estructuras selectivas simples y compuestas. A través de diversos ejemplos mostramos su representación tanto en pseudocódigo como en el lenguaje python. Dichos ejemplos realizan tareas basadas en condiciones, a través de las cuales es el algoritmo toma de decisiones.

4.5. Ejercicios

1. Carlos es un joven empresario que se dedica a la impresión 3D. Para este fin de año quiere imprimir esferas navideñas y hacer algunas promociones para animar a sus clientes. Nubia, su hermana, le sugirió que si un cliente hace un pedido de 20 esferas o más el costo por unidad sea de $ 12, en caso contrario el costo por unidad será de $ 20. Realiza un algoritmo y representalo mediante pseudocódigo y programa en python, para ayudar a Carlos a vender sus esferas.

2. Una empresa desea calcular la nomina de sus empleados con base en las horas trabajadas y el pago por hora. Esta empresa paga horas extras a sus empleados de la siguiente forma: Después de las 40 horas de trabajo, cada hora se paga al triple. Contruye el pseudocódigo y el programa en Python para ayudar a la empresa en el cálculo de su nómina.

3. Se requiere un algoritmo para determinar cuál de tres cantidades proporcionadas es la mayor. Represéntalo mediante pseudocódigo y programa en Python.

4. Doña Francisca tiene una pequeña empresa que se dedica al lavado de ropa. La dinámica es: el cliente lleva la ropa, se pesa y se calcula el pago que debe realizar el cliente. Doña Francisca tiene una promoción para sus clientes:

   - Por 1 kilo de ropa cobra $ 30
   - Por 2 kilos de ropa cobra $ 25
   - Por 3 kilos de ropa cobra $ 20
   - Por 4 kilos o más de ropa cobra $ 15

   Realiza un algoritmo representalo mediante pseudocódigo y programa en Python, para ayudar a Doña Francisca con su promoción.

Capítulo 5

Estructuras repetitivas

5.1. Introducción

Hasta el momento, las soluciones planteadas a los problemas propuestos han sido para una persona, un objeto o cosa, pero siempre de manera unitaria, tanto en las soluciones que se plantearon con estructuras secuenciales como con las selectivas; sin embargo, debemos considerar que, cuando se plantean problemas como calcular un sueldo, cabe la posibilidad de que el cálculo se tenga que hacer para dos o más empleados, un proceso de cálculo que por lógica debe ser el mismo para cada uno, pero donde existe la posibilidad de que los parámetros que determinan ese sueldo sean los que cambien. Por tal motivo se emplean estructuras denominadas repetitivas, de ciclo o de bucle, e independientemente del nombre que se les aplique, lo que importa es que permiten que un proceso pueda realizarse un número determinado de veces, donde solo cambien los parámetros que se utilizan en el proceso.

5.2. Estructuras repetitivas o de ciclo

Cuando se requiere que un proceso se efectúe de manera cíclica, se emplean estructuras que permiten el control de ciclos. Esas estructuras se emplean con base en las condiciones propias de cada problema, los nombres con los que se conocen éstas son: "Mientras que" y "Desde, hasta que". En la figura 5.1 se presentan las formas de estas estructuras mediante pseudocódigo.

Mientras condición lógica
    Proceso
Fin Mientras

Desde valor inicial **Hasta** valor final
    Proceso
Fin Desde

Figura 5.1: Estructuras repetitivas

Para el caso de la estructura "Mientras que", el ciclo se repite hasta que la condición

lógica resulta ser falsa; además, como se puede ver en la figura 5.1, en dicha estructura primero se evalúa y luego se realiza el proceso.

Las estructuras de tipo "Desde" se aplican cuando se tiene definido el número de veces que se realizará el proceso dentro del ciclo; lo que la hace diferente de las otras es que aquellas se pueden utilizar hasta que las condiciones cambien dentro del mismo ciclo. Estas condiciones pueden deberse a un dato proporcionado desde el exterior, o bien, al resultado de un proceso ejecutado dentro de éste, el cual marca el final. Además, en el ciclo "Desde", su incremento es automático, por lo cual no se tiene que efectuar mediante un proceso adicional, como en los otros dos tipos.

En los siguientes ejemplos se mostrará la aplicación de los dos tipos de ciclos antes mencionados.

5.2.1. Ejemplo: Suma diez cantidades con ciclo "Mientras que"

Diseñar un algoritmo que sume diez cantidades dadas por el usuario empleando la estructura repetitiva `Mientras que`. Representar dicho algoritmo mediante pseudocódigo y código en python.

Con base en lo que se requiere determinar se puede establecer que las variables requeridas para la solución del problema son las mostradas en la tabla 5.1.

Nombre de la variable	**Descripción**	**Tipo**
con	Contador	Entero, Int
n	Valor por sumar	Real, Float
suma	Suma de valores	Real, Float

Tabla 5.1: Variables del ejemplo suma cantidades

La solución de este problema mediante el ciclo `Mientras`, que también es conocido como ciclo `while` en Python, se puede establecer mediante el pseudocódigo 15.

Algoritmo 15: Ejemplo suma diez cantidades

Entradas: n (real)
Salidas: suma (real)

```
1:  Inicio
2:  con=1
3:  suma=0
4:  mientras con<=10 hacer
5:      Leer n
6:      suma=suma+n
7:      con=con+1
8:  fin_mientras
9:  Escribir suma
10: Fin
```

De esta solución planteada se puede ver, primero que el contador del ciclo `con` se inicializa en uno, posteriormente, se verifica que éste sea menor o igual a diez, que es lo que debe durar el ciclo (diez veces), ya dentro del ciclo el contador se incrementa por cada vuelta que dé y se realice el proceso de leer un valor y acumularlo en la suma.

En general, todo ciclo debe tener un valor inicial, un incremento y un verificador que establezca el límite de ejecución.

El código en Python para este ejemplo se muestra a continuación (ver Programa 5.1):

```
con=1
suma=0

while con<=10:
    n=float(input("Introduce un valor a sumar: "))
    suma=suma+n
    con=con+1

print(f"La suma total es: {suma}")
```

Programa 5.1: Suma diez cantidades

El programa 5.1 tiene como salida:

```
Introduce un valor a sumar: 1
Introduce un valor a sumar: 2
Introduce un valor a sumar: 3
Introduce un valor a sumar: 4
Introduce un valor a sumar: 5
Introduce un valor a sumar: 6
Introduce un valor a sumar: 7
Introduce un valor a sumar: 8
Introduce un valor a sumar: 9
Introduce un valor a sumar: 10
La suma total es: 55.0
```

La estructura mientras se utiliza regularmente cuando se tiene definida una condición lógica para detener el ciclo.

5.2.2. Ejemplo suma diez cantidades con ciclo "Desde"

Diseñar un algoritmo que sume diez cantidades dadas por el usuario empleando la estructura repetitiva `Desde`. Representar dicho algoritmo mediante pseudocódigo y código en python.

El ciclo `Desde` también es conocido como ciclo `for` en Python. Se utilizarán las mismas variables mostradas en la tabla 5.1. El pseudocódigo 16 muestra la solución

correspondiente utilizando el ciclo `Desde`.

Algoritmo 16: Ejemplo Suma diez cantidades con "Desde"

Entradas: n (real)
Salidas: suma (real)

```
1: Inicio
2: suma=0
3: desde con=1 hasta 10 hacer
4:     Leer n
5:     suma=suma+n
6: fin_desde
7: Escribir suma
8: Fin
```

El código en Python para este ejemplo se muestra a continuación (ver programa 5.2):

```
1 suma=0
2
3 for con in range(1,11):
4     n=float(input("Introduce un valor a sumar: "))
5     suma=suma+n
6
7 print(f"La suma total es: {suma}")
```

Programa 5.2: suma diez cantidades con Desde

El programa 5.2 tiene como salida:

```
Introduce un valor a sumar: 1
Introduce un valor a sumar: 2
Introduce un valor a sumar: 3
Introduce un valor a sumar: 4
Introduce un valor a sumar: 5
Introduce un valor a sumar: 6
Introduce un valor a sumar: 7
Introduce un valor a sumar: 8
Introduce un valor a sumar: 9
Introduce un valor a sumar: 10
La suma total es: 55.0
```

La estructura `desde` se utiliza cuando se conoce con certeza el número de veces que se debe repetir el ciclo.

Se debe observar que el incremento de la variable que controla el ciclo no se indica en este tipo de estructura, ya que el incremento o decremento de la variable se realiza de manera automática. Esta automatización se logra gracias a la función `range`. Los tres argumentos del tipo `range(m, n, s)` son:

- `m`: valor inicial
- `n`: valor final -1 (es -1 porque finaliza un valor anterior al establecido)
- `s`: salto (la cantidad que se avanza cada vez)

Si se escriben solo dos argumentos, Python le asigna a `s` el valor 1. Es decir `range(m, n)` es lo mismo que `range(m, n, 1)`.

Si se escribe solo un argumento, Python le asigna a `m` el valor 0 y a `s` el valor 1. Es decir, `range(n)` es lo mismo que `range(0, n, 1)`.

El tipo `range()` solo admite argumentos enteros. Si se utilizan argumentos decimales, se produce un error.

5.2.3. Ejemplo cálculo de promedio de números

Se requiere un algoritmo para obtener el promedio de un conjunto de números. Realiza el pseudocódigo y el programa en Python para representarlo, utilizando los dos tipos de estructuras de ciclo.

La tabla 5.2 muestra las variables que se van a utilizar para la solución del problema, sin importar qué estructura de ciclo se utilice; por consiguiente, es la misma para los dos tipos de ciclo para los que se dará la solución.

Nombre de la variable	**Descripción**	**Tipo**
con	Contador	Entero, Int
n	Cantidad de números	Entero, Int
num	Valor de un elemento	Entero, Int
suma	Suma de los números	Entero, Int
promedio	Valor promedio	Real, Float

Tabla 5.2: Variables del ejemplo promedio

La solución de este problema mediante el ciclo `Mientras` se puede establecer mediante el pseudocódigo 17:

Algoritmo 17: Obtiene el promedio

Entradas: n (entero), num (entero)
Salidas: promedio (real)

```
1: Inicio
2: con=1
3: suma=0
4: Leer n
5: mientras con<=n hacer
6:     Leer num
7:     suma=suma+num
8:     con=con+1
9: fin_mientras
10: promedio=suma/n
11: Escribir promedio
12: Fin
```

El código en Python para este ejemplo se muestra a continuación (ver programa 5.3):

```
con=1
suma=0
n=int(input("Introduce la cantidad de elementos a promediar: "))

while con<=n:
    num=int(input("Introduce un elemento: "))
    suma=suma+num
    con=con+1

promedio=suma/n
print(f"La suma total es: {promedio}")
```

Programa 5.3: Promedio con Mientras

El programa 5.3 tiene como salida:

```
Introduce la cantidad de elementos a promediar: 5
Introduce un elemento: 10
Introduce un elemento: 9
Introduce un elemento: 8
Introduce un elemento: 9
Introduce un elemento: 7
La suma total es: 8.6
```

La solución de este problema mediante el ciclo `Desde` se puede establecer mediante el pseudocódigo 18:

Algoritmo 18: Ejemplo Obtiene promedio con "Desde"

Entradas: n (entero), num (entero)
Salidas: promedio (real)

```
1: Inicio
2: suma=0
3: Leer n
4: desde con=1 hasta n hacer
5:     Leer num
6:     suma=suma+num
7: fin_desde
8: promedio=suma/n
9: Escribir promedio
10: Fin
```

El código en Python para este ejemplo utilizando la estructura `Desde` es el siguiente (ver código 5.4):

```
suma=0
n=int(input("Introduce la cantidad de elementos a promediar: "))

for con in range(1,n+1):
    num=int(input("Introduce un elemento: "))
    suma=suma+num

```

```
promedio=suma/n
print(f"La suma total es: {promedio}")
```

Programa 5.4: Promedio con Desde

El programa 5.4 tiene como salida:

```
Introduce la cantidad de elementos a promediar: 5
Introduce un elemento: 10
Introduce un elemento: 9
Introduce un elemento: 8
Introduce un elemento: 9
Introduce un elemento: 7
La suma total es: 8.6
```

5.2.4. Ejemplo cálculo de promedio sin conocer la cantidad de números

Se requiere un algoritmo para obtener el promedio de un conjunto de números, cuyo número de elementos se desconoce, el ciclo debe efectuarse siempre y cuando los elementos del conjunto de números sean mayores que cero. Realiza el pseudocódigo y el programa en Python para representarlo, utilizando el ciclo apropiado.

Como se puede ver, para resolver este problema no se puede utilizar el ciclo `Desde`, ya que no se tiene el número de elementos exacto, que es lo que determinaría el número de veces que se ejecuta el proceso que se encuentra dentro del ciclo.

Algoritmo 19: Obtiene el promedio sin número exacto de elementos

Entradas: elemento (real)
Salidas: promedio (real)

```
1: Inicio
2: con=0
3: suma=0
4: Leer elemento
5: mientras elemento>0 hacer
6:    suma=suma+elemento
7:    con=con+1
8:    Leer elemento
9: fin_mientras
10: promedio=suma/con
11: Escribir promedio
12: Fin
```

El ciclo apropiado para la solución de este problema es `Mientras`, ya que este ciclo se realiza siempre y cuando se cuente con un elemento mayor a cero, de una manera natural sin forzar el proceso en ningún momento, y en caso de que no se tenga elemento registrado el promedio es cero, y se debe indicar que no existe ningún elemento registrado.

La tabla 5.3 muestra las variables que se van a utilizar para la solución de este problema.

Nombre de la variable	Descripción	Tipo
con	Contador de elementos	Entero, Int
elemento	Valor de un elemento	Real, Float
suma	Suma de los elementos	Real, Float
promedio	Valor promedio	Real, Float

Tabla 5.3: Variables del ejemplo promedio sin número exacto de elementos

La representación del algoritmo para este problema se presenta mediante el pseudocódigo 19 y el programa en Python 5.5, en los cuales se utiliza el ciclo Mientras.

```
con=0
suma=0
elemento=float(input("Introduce un elemento a promediar: "))

while elemento>0:
    suma=suma+elemento
    con = con + 1
    elemento = float(input("Introduce un elemento a promediar: "))

promedio=suma/con
print(f"El promedio de los números es: {promedio}")
```

Programa 5.5: Variante del promedio

El programa 5.5 tiene como salida:

```
Introduce un elemento a promediar: 10
Introduce un elemento a promediar: 9
Introduce un elemento a promediar: 8
Introduce un elemento a promediar: 9
Introduce un elemento a promediar: 7
Introduce un elemento a promediar: 0
El promedio de los números es: 8.6
```

5.2.5. Ejemplo cálculo de ahorro en un año

Se requiere un algoritmo para determinar cuánto ahorrará una persona en un año, si al final de cada mes deposita cantidades variables de dinero; además, se requiere saber cuánto lleva ahorrado cada mes. Realiza el pseudocódigo y el programa en Python para representarlo, utilizando un ciclo apropiado.

La tabla 5.4 muestra las variables que se requieren para plantear la solución del problema.

Nombre de la variable	Descripción	Tipo
ahorro_mes	Ahorro mensual	Real, Float
con_mes	Contador de meses	Entero, Int
cantidad	Cantidad que se va a ahorrar	Real, Float

Tabla 5.4: Variables del ejemplo ahorro

Este problema se puede resolver mediante la utilización de cualquiera de los ciclos, dado que se conoce el número de veces que se debe efectuar el ciclo.

Algoritmo 20: Ejemplo Cálculo de ahorro con "Desde"

Entradas: cantidad (real)
Salidas: ahorro_mes (real)

```
1: Inicio
2: con_mes=1
3: ahorro_mes=0
4: desde con_mes=1 hasta 12 hacer
5:     Leer cantidad
6:     ahorro_mes=ahorro_mes+cantidad
7:     Escribir ahorro_mes
8: fin_desde
9: Escribir ahorro_mes
10: Fin
```

La solución para este problema utilizando el ciclo `Desde` se plantea mediante el pseudocódigo 20 y el código 5.6.

```
ahorro_mes=0

for con_mes in range(1,13):
    cantidad=float(input("Introduce la cantidad a ahorrar: "))
    ahorro_mes=ahorro_mes+cantidad
    print(f"El ahorro del mes {con_mes} es {ahorro_mes}")

print(f"El total ahorrado en el año es: {ahorro_mes}")
```

Programa 5.6: Ahorro anual

El programa 5.6 tiene como salida:

```
Introduce la cantidad a ahorrar: 100
El ahorro del mes 1 es 100.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 2 es 200.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 3 es 300.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 4 es 400.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 5 es 500.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 6 es 600.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 7 es 700.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 8 es 800.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 9 es 900.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 10 es 1000.0
Introduce la cantidad a ahorrar: 100
El ahorro del mes 11 es 1100.0
Introduce la cantidad a ahorrar: 100
```

```
El ahorro del mes 12 es 1200.0
El total ahorrado en el año es: 1200.0
```

5.2.6. Ejemplo serie Fibonacci

Elabora un algoritmo para generar un número de elementos de la sucesión de Fibonacci (1, 1, 2, 3, 5, 8, 13,...). Realiza el pseudocódigo y el programa en Python para representarlo, utilizando el ciclo apropiado.

El planteamiento del algoritmo correspondiente se hace a partir del análisis de la sucesión, en la que se puede observar que un tercer valor de la serie está dado por la suma de los dos valores previos, de aquí que se asignan los dos valores para sumar (1, 1), que dan la base para obtener el siguiente elemento que se busca, además, implica que el ciclo se efectué dos veces menos.

Algoritmo 21: Obtiene la serie Fibonacci

Entradas: N (entero)
Salidas: suma (entero)

```
1: Inicio
2: p1=1
3: p2=1
4: con=3
5: Leer N
6: Escribir P1, P2
7: mientras con<=N hacer
8:    suma=p1+p2
9:    Escribir suma
10:   p1=p2
11:   p2=suma
12:   con=con+1
13: fin_mientras
14: Fin
```

Las variables que se requieren para la solución de este problema se muestran en la tabla 5.5. En lo que respecta a qué tipo de ciclo se debe utilizar, es indistinto.

Nombre de la variable	Descripción	Tipo
p1	Puntero uno	Entero, Int
p2	Puntero dos	Entero, Int
suma	Suma punteros	Entero, Int
con	Contador	Entero, Int
N	Elementos de la serie	Entero, Int

Tabla 5.5: Variables del ejemplo serie Fibonacci

La solución para este problema utilizando el ciclo `Desde` se plantea mediante el

pseudocódigo 21 y el código 5.7.

```
p1=1
p2=1
con=3
N=int(input("Introduce la cantidad de elementos de la serie: "))
print(p1)
print(p2)

while con<=N:
    suma=p1+p2
    print(suma)
    p1=p2
    p2=suma
    con=con+1
```

Programa 5.7: Serie Fibonacci

El programa 5.7 tiene como salida:

```
Introduce la cantidad de elementos de la serie: 6
1
1
2
3
5
8
```

5.2.7. Ejemplo número primo

Un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. Por ejemplo, el número 4 no es un número primo, porque lo divide el 1, 2 y el mismo 4. Sin embargo, el número 5 si es primo, ya que únicamente es divisible entre 1 y 5.

El reto en esta ocasión es desarrollar un algoritmo que diga si un número proporcionado por el usuario es primo o no lo es. Represéntalo mediante pseudocódigo y código en Python.

La idea es este algoritmo es analizar todos los números desde 1 hasta el número que queremos determinar si es primo o no, es decir, `N`. Dicho análisis consistirá en ver si alguno de los valores desde 1 hasta `N` divide a `N`, si es así se cuenta el divisor. Si es primo, solo debe haber dos divisores (1 y N), en caso contrario, el número no es primo.

La tabla 5.6 muestra las variables que se necesitan para resolver este ejercicio.

Nombre de la variable	**Descripción**	**Tipo**
N	Número a verificar si es primo	Entero, Int
con	Contador	Entero, Int
conDivisores	Cuenta divisores	Entero, Int
R	Resultado	Cadena, String

Tabla 5.6: Variables del ejemplo número primo

El pseudocódigo que representa el algoritmo que determina si un número es primo es el siguiente (ver pseudocódigo 22):

Algoritmo 22: Determina si un número es primo o no

Entradas: N (entero)
Salidas: R (cadena)

```
1: Inicio
2: con=1
3: conDivisores=0
4: Leer N
5: mientras con<=N hacer
6:    si N%con==0 entonces
7:       conDivisores=conDivisores+1
8:    fin_si
9:    con=con+1
10: fin_mientras
11: si conDivisores==2 entonces
12:    R="Es primo"
13: de_lo_contrario
14:    R="No es primo"
15: fin_si
16: Escribir R
17: Fin
```

La solución para este problema utilizando el ciclo `Mientras` se plantea mediante el código 5.8.

```
con=1
conDivisores=0

N=int(input("Introduce el número a verificar: "))

while con<=N:
    if N%con==0:
        conDivisores+=1
    con+=1

if conDivisores==2:
    R="Es primo"
else:
    R="No es primo"

print(R)
```

Programa 5.8: Número primo

El programa 5.8 tiene como salida:

```
Introduce el número a verificar: 11
Es primo
```

5.2.8. Ejemplo máximo común divisor

En este ejemplo, se utiliza el algoritmo de Euclides (ver 1.2.3) para calcular el MCD de dos números dados. Este algoritmo está basado en divisiones sucesivas, las cuales finalizan cuando el residuo es cero. El MCD será entonces el último residuo distinto de cero.

El pseudocódigo que representa el algoritmo que determina el MCD es el siguiente (ver pseudocódigo 23):

Algoritmo 23: Determina el MCD de dos números

Entradas: a, b (enteros)
Salidas: numerador (entero)

```
1: Inicio
2: Leer a, b
3: si a>b entonces
4:    numerador=a
5:    denominador=b
6: de_lo_contrario
7:    numerador=b
8:    denominador=a
9: fin_si
10: res=-1
11: mientras res!=0 hacer
12:    res=numerador % denominador
13:    numerador = denominador
14:    denominador=res
15: fin_mientras
16: Escribir numerador
17: Fin
```

Cómo se desconoce el número de interacciones necesarias para llegar a un residuo igual a cero dados dos números cualesquiera, entonces la estructura mientras es la más adecuada para este ejemplo.

La tabla 5.7 muestra las variables que se necesitan para resolver este ejercicio.

Nombre de la variable	Descripción	Tipo
a	Primer número a calcular el MCD	Entero, Int
b	Segundo número a calcular el MCD	Entero, Int
numerador	El número que se va a dividir	Entero, Int
denominador	El número que divide	Entero, Int
res	El residuo de la división	Entero, Int

Tabla 5.7: Variables del ejemplo MCD

La solución para este problema utilizando el ciclo `Mientras` se plantea mediante el código 5.9.

```
#Leer números a calcular su MCD
a=int(input("Escribe el primer número: "))
b=int(input("Escribe el segundo número: "))
#De los números leídos, determinar el numerador y el denominador
if a > b:
    numerador=a
    denominador=b
else:
    numerador=b
    denominador=a

res = -1
#Se aplica el algoritmo de Euclides
while res!=0:
    res = numerador % denominador
    numerador=denominador
    denominador=res

print(f"El MCD es: {numerador}")
```

Programa 5.9: MCD

El programa 5.9 tiene como salida:

```
Escribe el primer número: 100
Escribe el segundo número: 50
El MCD es: 50
```

5.2.9. Ejemplo mínimo común múltiplo

En este ejemplo, se utiliza el algoritmo visto en la sub-sección 1.2.4 para calcular el mcm de dos números dados.

La tabla 5.8 muestra las variables que se necesitan para resolver este ejercicio.

Nombre de la variable	**Descripción**	**Tipo**
a	Primer número a calcular el MCD	Entero, Int
b	Segundo número a calcular el MCD	Entero, Int
numerador	El número que se va a dividir	Entero, Int
denominador	El número que divide	Entero, Int
res	El resto de la división	Entero, Int
mcm	Almacena el m.c.m	Real, Float

Tabla 5.8: Variables del ejemplo m.c.m.

El pseudocódigo que representa el algoritmo que determina el m.c.m. es el siguiente (ver pseudocódigo 24):

Algoritmo 24: Determina el m.c.m. de dos números

Entradas: a, b (enteros)
Salidas: numerador (entero)

```
1: Inicio
2: Leer a, b
3: si a>b entonces
4:    numerador=a
5:    denominador=b
6: de_lo_contrario
7:    numerador=b
8:    denominador=a
9: fin_si
10: res=-1
11: mientras res!=0 hacer
12:    res=numerador % denominador
13:    numerador = denominador
14:    denominador=res
15: fin_mientras
16: mcm=(a*b)/numerador
17: Escribir mcm
18: Fin
```

La solución para este problema utilizando el ciclo `Mientras` se plantea mediante el código 5.10.

```
#Leer números a calcular su MCD
a=int(input("Escribe el primer número: "))
b=int(input("Escribe el segundo número: "))
#De los números leídos, determinar el numerador y el denominador
if a > b:
    numerador=a
    denominador=b
else:
    numerador=b
    denominador=a

res = -1
#Se aplica el algoritmo de Euclides
while res!=0:
    res = numerador % denominador
    numerador=denominador
    denominador=res

#Se calcula el m.c.m.
mcm=(a*b)/numerador

```

```
22 print(f"El m.c.m. es: {mcm}")
```

Programa 5.10: m.c.m.

El programa 5.10 tiene como salida:

```
Escribe el primer número: 30
Escribe el segundo número: 40
El m.c.m. es: 120.0
```

5.2.10. Ejemplo dibuja triángulo numérico

Hasta ahora, los ejemplos que hemos visto solo emplean un bucle o ciclo. Sin embargo, en la práctica muchos programas requieren ciclos anidados, es decir, ciclos dentro de ciclos. Tal como se muestra en el siguiente ejemplo, en donde el usuario introduce un número y el programa debe generar un triángulo de números:

```
Introduce un número: 10
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Algoritmo 25: Ejemplo Dibuja triángulo

Entradas: n (entero)
Salidas:

```
1: Inicio
2: Leer n
3: desde filas=1 hasta n hacer
4:    desde columnas=1 hasta filas hacer
5:       Escribir columnas y un espacio o tabulador
6:    fin_desde
7:    Saltar línea
8: fin_desde
9: Fin
```

La tabla 5.9 muestra las variables que se necesitan para resolver este ejercicio.

Nombre de la variable	Descripción	Tipo
filas	Número de filas	Entero, Int
columnas	Número de columnas	Entero, Int
n	Tamaño del triángulo	Entero, Int

Tabla 5.9: Variables del ejemplo dibuja triángulo

Cualquiera de los bucles que hemos visto se pueden utilizar; sin embargo, por elegancia, trabajaremos este ejercicio con un bucle `Desde`. El pseudocódigo 25 muestra la representación del algoritmo.

La solución para este problema utilizando el ciclo `Desde` se plantea mediante el código 5.11.

```
n=int(input("Introduce un número: "))

for filas in range(1,n+1):
    for columnas in range(1,filas+1):
        print(f"{columnas}",end="\t")
    print()
```

Programa 5.11: Dibuja triángulo

5.3. Conclusiones

Las estructuras repetitivas vistas en este capítulo son: mientras y para. Dichas estructuras se han representado tanto en pseudocódigo, como el lenguaje python. A través de los ejemplos presentados se muestra cuándo es conveniente utilizar mientras y cuando para.

5.4. Ejercicios

1. Se requiere un algoritmo para determinar, de `n` cantidades, cuántas son menores o iguales a cero y cuántas mayores a cero. Realiza el pseudocódigo y el programa en Python para representarlo, utilizando el ciclo apropiado.

2. Realiza un algoritmo para generar e imprimir los números pares que se encuentran entre 100 y 200. Realiza el pseudocódigo y programa en Python para representarlo, utilizando el ciclo apropiado.

3. Realiza un programa que genere la figura siguiente:

```
Introduce el tamaño del triángulo: 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

4. Un profesor tiene un salario inicial de $ 2600, y recibe un incremento de 11 % anual. ¿Cuál es su salario al cabo de 7 años? ¿Qué salario ha recibido en cada uno de los

7 años? Realiza el algoritmo y representa la solución mediante el pseudocódigo y el programa en Python, utilizando el ciclo apropiado.

Capítulo 6

Cadenas y colecciones

6.1. Introducción

Las cadenas y las colecciones son objetos que pueden ser caracterizados por su organización y las operaciones que se pueden hacer sobre ellos. Aunque en Python el concepto de "objetos" tiene un significado particular, por ahora debemos entenderlo como cualquier entidad dentro del algoritmo y código fuente, como variable, constante, función, etc.

Una colección permite agrupar varios objetos que tienen en común cierta característica. Por ejemplo, si necesitamos almacenar los nombres de alumnos de un curso de programación, sería más conveniente ubicarlos a todos dentro de una misma colección de nombre alumnos.

6.2. Cadenas

6.2.1. Índices y slices

Una cadena está formada por caracteres (es decir, letras, números y símbolos), a los cuales se puede acceder a través de sus índices. En el índice 0 se encuentra el primer carácter y el último carácter de la cadena se encuentra en la posición tamaño (longitud) de la cadena menos uno.

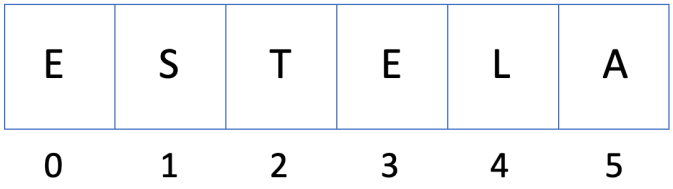


Figura 6.1: Ejemplo cadena

Veamos un ejemplo, tenemos la cadena "ESTELA", la cual mediante una asignación es almacenada en la variable `cad` (ver línea 1 de programa 6.1). Si queremos el primer caracter de la cadena, basta con escribir `cad[0]` en un `print` y se muestra el caracter "E" (ver línea 3 de programa 6.1).

```
1 cad="ESTELA"
2
3 print(cad[0])
```

Programa 6.1: Ejemplo cadena 1

El programa 6.1 tiene como salida:

```
E
```

Python, a diferencia de otros lenguajes de programación, permite trabajar con índices negativos. Un índice negativo sirve para iniciar a partir del final de la cadena, por ejemplo, en la misma cadena "ESTELA" si queremos el último carácter, podemos tener acceso mediante el índice -1, el cual devuelve el valor "A" (ver programa 6.2 y su salida). La lógica nos dice que si quisiéramos el penúltimo caracter de la cadena tendríamos que escribir `cad[-2]`, lo cual es complemente correcto, y así para los demás elementos de la cadena.

```
1 cad="ESTELA"
2
3 print(cad[-1])
```

Programa 6.2: Ejemplo cadena 2

El programa 6.2 tiene como salida:

```
A
```

Teniendo claro el concepto de índices, podemos pasar a algo un poco más avanzado como son los **slicing**, los cuales permiten obtener una subcadena de la cadena principal.

En resumen, éstas son sus principales opciones:

- `cadena[inicio:final]` desde el índice 'inicio' hasta el índice 'final'-1
- `cadena[inicio:]` desde el índice 'inicio' hasta el final de la cadena
- `cadena[:final]` desde el índice 0 hasta elemento 'final'-1
- `cadena[:]` todos los elementos de la cadena

Además de estos cuatro casos que son los más comunes, también puedes utilizar un tercer valor opcional llamado `step` o `salto`:

- `cadena[inicio:final:salto]` desde el elemento 'inicio' hasta 'final' pero saltando el número de elementos indicado por 'salto'

Otra de las opciones más interesantes del `slicing` es que el principio, el final y el salto pueden ser números negativos. Esto indica que se empieza a contar desde el final de la cadena.

En el mismo ejemplo de la cadena “ESTELA", si quisiéramos los caracteres desde el segundo elemento hasta el cuarto, tendríamos que realizar lo siguiente:

```
1 cad="ESTELA"
2
3 print(cad[1:5])
```

Programa 6.3: Ejemplo cadena 3

El programa 6.3 tiene como salida:

```
STEL
```

Nota que el segundo carácter se encuentra a partir del índice 1, y el quinto elemento se encuentra en el índice 4, el **slicing** va desde 1 hasta 5 ([1:5]); sin embargo, se detiene una posición anterior a la que se le indica. Es decir, si el final es 5 se detiene en la posición 4. Por eso, obtenemos la subcadena “STEL”.

A continuación, un último ejemplo del uso de **slicing**, en donde partiendo de la cadena “ESTELA”, queremos obtener la subcadena “STELA”. Para ello realizamos lo siguiente:

```
1 cad="ESTELA"
2
3 print(cad[1:])
```

Programa 6.4: Ejemplo cadena 4

El programa 6.4 tiene como salida:

```
STELA
```

En este punto es importante mencionar que las cadenas son **inmutables**, es decir, no se pueden modificar, al menos no directamente. Veamos el siguiente ejemplo, en el cual se intenta modificar el primer carácter de la cadena. Al ejecutarse este programa, produce una excepción (un tipo de error).

```
1 cad="ESTELA"
2
3 cad[0]="e"
4
5 print(cad)
```

Programa 6.5: Ejemplo cadena 5

El programa 6.5 tiene como salida:

```
Traceback (most recent call last):
  File "/Users/cadenaEstela5.py", line 3, in <module>
    cad[0]="e"
TypeError: 'str' object does not support item assignment
```

En el ámbito de la computación, la concatenación es una operación que consiste en la unión de dos o más caracteres para desarrollar una cadena de caracteres. Esta cadena

es una secuencia finita y ordenada de elementos que forman parte de un determinado lenguaje formal. La concatenación puede llevarse a cabo incluso uniendo dos cadenas de caracteres o enlazando un carácter a otra cadena. El operador + permite concatenar (unir) cadenas.

Si quisiéramos modificar la cadena en la posición 0, tendríamos que hacer algo como lo siguiente:

```
cad="ESTELA"

cad="e"+cad[1:]

print(cad)
```

Programa 6.6: Ejemplo cadena 5

El programa 6.6 tiene como salida:

```
eSTELA
```

6.2.2. Longitud de una cadena

La función `len()` devuelve la longitud de una cadena de caracteres. El argumento de la función `len()` es la cadena que queremos "medir".

```
fruta="manzana"

print(len(fruta))
```

Programa 6.7: Funcíon `len`

El programa 6.7 tiene como salida:

```
7
```

Como podemos observar en el programa 6.7 la palabra manzana (almacenada en la variable `fruta`), tiene siete caracteres. Al pasar la variable `fruta` a la función `len` se tiene como resultado el valor 7.

Para obtener la última letra de una cadena, podríamos sentirnos tentados a probar algo como esto:

```
fruta="manzana"
longitud = len(fruta)
ultima = fruta[longitud]
print(ultima)
```

Programa 6.8: Función len última letra de cadena incorrecto

El código del programa 6.8 provoca un error en tiempo de ejecución `IndexError: string index out of range`. La razón es que no hay una siete-ésima letra en "manzana". Como empezamos a contar desde cero, las siete letras están numeradas del 0 al 6. Para obtener el último carácter tenemos que restar 1 a la `longitud`:

```
fruta="manzana"
longitud = len(fruta)
ultima = fruta[longitud-1]
print(ultima)
```

Programa 6.9: Función len última letra de cadena correcto

6.2.3. Métodos para cadenas

Los métodos son funcionalidades que, en este caso de las cadenas, ya vienen bien definidos en Python. Podemos decir que un método recibe datos de entrada y devuelve salidas. A los datos que recibe como entrada, se les conoce como argumentos. Ademas podemos decir que un método tiene un nombre bien definido, lo que no se sabe es cómo implementa su funcionalidad.

Son muchos los métodos que Python tiene para el manejo de cadenas; sin embargo, en esta subsección nos centraremos en los que consideramos los mas importantes.

Método upper

Este método permite cambiar una cadena a mayúsculas. Su uso es muy simple:

```
‘‘cadena a convertir’’.upper()
```

Ejemplo

```
cadena="hola mundo".upper()

print(cadena)
```

Programa 6.10: Ejemplo cadena a mayúsculas

El programa 6.10 tiene como salida:

```
HOLA MUNDO
```

En este caso, la cadena “hola mundo” invoca al método `upper`, con lo que se convierte a mayúsculas y se almacena en la variable `cadena`, la cual se muestra con la función `print`.

Método lower

Este método sirve para cambiar una cadena a minúsculas. Su función es igual de sencilla y tiene el siguiente formato:

```
‘‘cadena a convertir’’.lower()
```

Ejemplo:

```
cadena="HOLA MUNDO".lower()

print(cadena)
```

Programa 6.11: Ejemplo cadena a minúsculas

El programa 6.11 tiene como salida:

```
hola mundo
```

En este ejemplo se emplea la cadena "HOLA MUNDO", en mayúsculas. Dicha cadena invoca al método `lower()` que la convierte a minúsculas. La cadena convertida en minúsculas se almacena en la variable `cadena`. Esta variable se muestra a través de `print` en pantalla.

Método count

Este método sirve para saber cuántas veces aparece un carácter en una cadena. Su formato es:

```
cadena.count(caracter)
```

Donde `cadena` es una cadena de caracteres o una variable que tenga almacenada una cadena; y `caracter` es el argumento que se quiere determinar cuántas veces aparece en la cadena. Este método devuelve el número de apariciones del carácter en la cadena.

Veamos su uso a través de un ejemplo:

```
cadena="hola mundo".count('o')

print(cadena)
```

Programa 6.12: Ejemplo count

El programa 6.12 tiene como salida:

```
2
```

En este ejemplo se determina el número de veces que aparece el carácter "o" en la cadena "hola mundo". Como podemos ver, dicho carácter aparece dos veces, que es la salida que nos devuelve Python.

Método find

Sirve para conocer el índice de la cadena donde aparece una palabra o carácter determinado. Veamos su funcionamiento a través de un ejemplo:

```
cadena="hola hola hola hola mundo".find('mundo')

print(cadena)
```

Programa 6.13: Ejemplo find

El programa 6.13 tiene como salida:

```
20
```

En este ejemplo tenemos la cadena "hola hola hola hola mundo", y a través del método `find` queremos determinar en qué índice de dicha cadena aparece la palabra "mundo" (pasada como argumento a find). Si contamos de izquierda a derecha, empezando desde 0, llegamos a la posición 20, que es donde se encuentra el inicio de la palabra "mundo". En caso de que la palabra o carácter que se busca no se encuentre, el método `find` devuelve -1.

Método islower

Este método devuelve un valor `True` si la cadena que lo invoca está completamente en minúsculas. En caso contrario, devuelve `False`. Ejemplo:

```
1 cadena="hola mundo".islower()
2
3 print(cadena)
```

Programa 6.14: Ejemplo islower

El programa 6.14 tiene como salida:

```
True
```

Método startswith

Este método recibe como argumento una palabra o carácter y devuelve `True` si en la cadena que invoca al método, dicha palabra o carácter aparece al inicio. Veamos el ejemplo:

```
1 cadena="hola mundo".startswith('hola')
2
3 print(cadena)
```

Programa 6.15: Ejemplo startswith

El programa 6.15 tiene como salida:

```
True
```

La cadena "hola mundo" llama al método `startswith`, al cual se le pasa como argumento la palabra "hola". En este caso, dicho método devuelve `True`, ya que la cadena "hola mundo" inicia con la palabra "hola".

Método endswith

Este método sirve para verificar si la cadena termina con una palabra o carácter que recibe como argumento. En caso de que sí termine con dicha palabra o carácter, devuelve `True`, si no, devuelve `False`. Veamos el ejemplo:

```
cadena="hola mundo".endswith('mundo')

print(cadena)
```

Programa 6.16: Ejemplo endswith

El programa 6.16 tiene como salida:

```
True
```

La cadena "hola mundo" llama al método `endswith` y le pasa como argumento la palabra "mundo". Como efectivamente dicha cadena termina con la palabra "mundo", este método devuelve `True`.

Método split

Este método separa los elementos de una cadena cada vez que encuentra un carácter que se le pase como argumento. Devuelve una lista con los elementos separados, por ejemplo:

```
cadena="hola mundo".split()

print(cadena)
```

Programa 6.17: Ejemplo split

El programa 6.17 tiene como salida:

```
['hola', 'mundo']
```

La cadena "hola mundo" invoca al método `split` sin pasarle ningún argumento; lo cual quiere decir que separará la cadena por espacios en blanco que encuentre. Como podemos ver en la salida, devuelve una lista (un concepto que veremos más adelante en este capítulo) con los elementos separados.

Para dejar más claro el método `split` veamos otro ejemplo:

```
cadena="hola-hola-mundo".split('-')

print(cadena)
```

Programa 6.18: Ejemplo split 2

El programa 6.18 tiene como salida:

```
['hola', 'hola', 'mundo']
```

Ahora la cadena "hola-hola-mundo" invoca al método `split`, al cual se le pasa como argumento el caracter "-", que significa que cuando encuentre este carácter en la cadena separará la palabra y la almacenará en una lista, tal cual se muestra en la salida del programa.

Método join

Retorna una cadena resultante de concatenar la cadena que recibe como argumento, separada por la cadena (S) sobre la que se llama al método.

`S.JOIN(secuencia)`

```
cadena=",".join("roberto")

print(cadena)
```

Programa 6.19: Ejemplo join

En este ejemplo, a la cadena "roberto" (que se le pasa como argumento al método `join`) se le concatena a cada elemento de la cadena el carácter ",". Lo cual genera la salida siguiente:

```
r,o,b,e,r,t,o
```

Método replace

Este método reemplaza un elemento por otro en una cadena de caracteres (la que invoca al método `replace`). Tanto el elemento que se va a reemplazar como el elemento reemplazado son pasados como argumentos. Este método devuelve una cadena con el elemento reemplazado. Ejemplo:

```
cadena="hola mundo".replace('o','9')

print(cadena)
```

Programa 6.20: Ejemplo replace

En la cadena "hola mundo" se reemplaza el caracter "o" por el "9". Finalmente, se tiene la salida que se muestra a continuación:

```
h9la mund9
```

6.2.4. Recorrido de una cadena

Muchos cálculos implican procesar una cadena carácter por carácter. A menudo se empieza por el principio, se selecciona cada carácter por turno, se hace algo con él y se sigue hasta el final. Este patrón de proceso se llama recorrido. Una forma de codificar un **recorrido** es con una sentencia `while`:

```
fruta="manzana"
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice += 1
```

Programa 6.21: Recorrido de una cadena con while

Este bucle recorre la cadena y muestra cada letra en una línea distinta. La condición del bucle 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 bucle. El último carácter al que se accede es el que tiene el índice `len(fruta)-1`, que es el último carácter de la cadena.

Usar un índice para recorrer un conjunto de valores es tan común que Python proporciona una sintaxis alternativa más simple — el bucle `for`:

```
fruta="manzana"

for carac in fruta:
    print(carac)
```

Programa 6.22: Recorrido de una cadena con for

Cada vez que recorremos el bucle, se asigna a la variable `carac` el siguiente carácter de la cadena. El bucle continúa hasta que no quedan más caracteres.

Para los recorridos con `while` y con `for` la salida es la siguiente:

```
m
a
n
z
a
n
a
```

6.3. Listas

Las listas en Python son una secuencia de valores que pueden ser de cualquier tipo: cadenas, enteros, flotantes, contenido mixto o cualquier otro. En esta sección hablaremos sobre los métodos de listas de Python y cómo crear, agregar elementos, agregar al final, invertir y muchas otras funciones de listas de Python.

6.3.1. Crear listas

Las listas pueden ser creadas con datos y sin datos. Veamos un ejemplo donde crearemos una lista vacía:

```
lista=[]

print(lista)
```

Programa 6.23: Ejemplo lista vacía

La salida del programa 6.23 es:

```
[]
```

Como podemos observar en el ejemplo anterior, para crear una lista se deben usar los corchetes [ ]. Al imprimir la lista anterior, ésta nos muestra únicamente los corchetes, los cuales son distintivos de una lista.

También podemos agregar datos a la lista al momento de crearla, por ejemplo:

```
lista=["Lunes","Martes","Miércoles","Jueves","Viernes"]

print(lista)
```

Programa 6.24: Ejemplo lista con datos

La salida del programa 6.24 es:

```
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes']
```

6.3.2. Mostrar elementos de la lista

Podemos mostrar un elemento de la lista a través de sus índices. En el índice 0 se encuentra el primer elemento de la lista. A diferencia de las cadenas, los elementos de las listas no son solo caracteres, como ya se mencionó anteriormente, estos elementos pueden ser cadenas, números u otras listas, por ejemplo. Veamos cómo acceder al primer elemento de una lista que tiene los días de la semana:

```
lista=["Lunes","Martes","Miércoles","Jueves","Viernes"]

print(lista[0])
```

Programa 6.25: Ejemplo primer elemento de la lista

La salida del programa 6.25 es:

```
Lunes
```

A partir de una lista, se pueden obtener sublistas utilizando el concepto de `slicing` que ya habíamos abordado en el tema de cadenas. Por ejemplo, en la lista que tiene los días de la semana, si quisiéramos únicamente los primeros tres días de la semana, podemos hacer lo siguiente:

```
lista=["Lunes","Martes","Miércoles","Jueves","Viernes"]

print(lista[0:3])
```

Programa 6.26: Ejemplo sublista

La salida del programa 6.26 es:

```
['Lunes', 'Martes', 'Miércoles']
```

Donde 0 es el inicio y 3 es el fin -1 de la sublista.

Veamos otro ejemplo, ahora se desea obtener una sublista que contenga los días desde el martes hasta el jueves. Entonces, el código quedaría de la manera siguiente:

```
lista=["Lunes","Martes","Miércoles","Jueves","Viernes"]

print(lista[1:4])
```

Programa 6.27: Ejemplo sublista dos

La salida del programa 6.27 es:

```
['Martes', 'Miércoles', 'Jueves']
```

6.3.3. Listas con elementos de diferentes tipos de datos

Como ya se mencionó al inicio de este sección, las listas pueden contener elementos de cualquier tipo de datos. Veamos un ejemplo para que quede más claro:

```
1 lista=["Lunes","Martes","Miércoles","Jueves","Viernes", 40, 5.67,
      [1,2,3]]
2
3 print(lista)
```

Programa 6.28: Ejemplo listas con elementos de diferentes tipos de datos

La salida del programa 6.28 es:

```
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 40, 5.67, [1, 2, 3]]
```

Como podemos observar, esta lista contiene cadenas, números enteros, números con parte fraccionaria e incluso una lista.

6.3.4. Determinar la cantidad de elementos en una lista

Para determinar la cantidad de elementos que tiene una lista, se puede usar la función `len`. Ésta recibe como argumento la lista y devuelve la cantidad de elementos que contiene dicha lista. Veamos un ejemplo:

```
1 lista=["Lunes","Martes","Miércoles","Jueves","Viernes", 40, 5.67,
      [1,2,3]]
2
3 print(len(lista))
```

Programa 6.29: Ejemplo len

La salida del programa 6.29 es:

```
8
```

En total hay 8 elementos en la lista, incluso, la lista contenida dentro de la lista cuenta como un solo elemento.

6.3.5. Insertar nuevos elementos en la lista

Podemos insertar nuevos elementos al final de la lista, para ello usaremos el método `append`, el cual es invocado por la lista en la que se agregará el nuevo elemento. Este método recibe como argumento el elemento por insertar en la lista. Veamos un ejemplo:

```
1 lista=[1,2,3,4,5]
2
3 lista.append(6)
4
5 print(lista)
```

Programa 6.30: Ejemplo append

La salida del programa 6.30 es:

```
[1, 2, 3, 4, 5, 6]
```

Inicialmente, se tiene una lista con elementos enteros que van desde el 1 al 5. Posteriormente, mediante el método `append` se agrega un nuevo elemento al final de dicha lista como podemos ver en la salida del programa.

Puede darse el caso en que se necesita agregar un elemento en la lista, pero no al final, si no en una posición específica. El método `insert` soluciona este problema. Dicho método es invocado por la lista en la que se va a insertar un elemento, recibe como argumentos la posición en la que se debe insertar el elemento, así como el elemento que se va a agregar. Ejemplo:

```
1 lista=[1,2,4,5]
2
3 lista.insert(2,3)
4
5 print(lista)
```

Programa 6.31: Ejemplo insert

La salida del programa 6.31 es:

```
[1, 2, 3, 4, 5]
```

Inicialmente se tiene una lista con cuatro elementos [1,2,4,5], el método `insert` agrega el elemento 3 en la posición 2 (son datos que `insert` recibe como argumentos). El primer argumento es la posición en la que se va a insertar el elemento, y el segundo argumento es el elemento que queremos insertar.

Para agregar varios elementos simultáneamente al final de la lista usaremos el método `extend`. Este método recibe como argumento una lista con los elementos que se quieren insertar en la lista que lo invoca. Su uso se ejemplifica a continuación:

```
1 lista=[1,2,3,4,5]
2
3 lista.extend([6,7,8])
4
5 print(lista)
```

Programa 6.32: Ejemplo extend

Inicialmente se tiene una lista con elementos enteros del 1 al 5. Mediante el método `extend` se agregan tres elementos más a la lista, los cuales son: 6, 7 y 8. La salida nos presenta la lista con todos los elementos integrados.

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

El operador + permite unir, en una sola lista, los elementos de las listas a las que se aplica dicho operador. Veamos el siguiente ejemplo:

```
1 lista1=[1,2,3,4,5]
2 lista2=[6,7,8]
3
4 lista3=lista1+lista2
```

```
5
6 print(lista3)
```

Programa 6.33: Ejemplo suma listas

Inicialmente, se tienen dos listas (`lista1` y `lista2`), en la `lista1` hay cinco elementos y en la `lista2` hay tres elementos. Se aplica el operador más (+) a ambas listas, y se almacena el resultado en `lista3`, al mostrarse observamos que ya integra tanto los elementos de `lista1` como de `lista2`.

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

6.3.6. Buscar elementos en la lista

La palabra reservada `in` nos permite saber si un elemento se encuentra en una lista. Devuelve `True` en caso de que el elemento se encuentre en la lista y `False` si no se encuentra. Ejemplo:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 print(3 in lista)
```

Programa 6.34: Ejemplo in

Se tiene una lista con 6 elementos, cinco números enteros y una cadena. Posteriormente, en un `print` se pregunta si el elemento 3 se encuentra en la lista (`3 in lista`). Como efectivamente el elemento 3 se encuentra en la lista, se imprime `True`. Cabe aclarar que también se pudo almacenar en una variable el resultado de evaluar `3 in lista`. La salida del programa 6.34 es:

```
True
```

El método `index` permite conocer la posición en la que se encuentra un elemento en la lista. Para ello es necesario pasarle como argumento el elemento que queremos conocer su posición. Si el elemento se encuentra, `index` devuelve la posición, en caso contrario, manda un error. Veamos el siguiente ejemplo:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 print(lista.index("Juan"))
```

Programa 6.35: Ejemplo index

La salida del programa 6.35 es:

```
5
```

En la misma lista con 6 elementos, lista invoca al método `index` y le pasa como argumento el elemento "Juan". Como efectivamente el elemento se encuentra en la lista, `index` devuelve la posición 5 que es donde se encuentra dicho elemento.

6.3.7. Contar cuantas veces aparece un elemento en la lista

Las listas permiten tener elementos repetidos, por lo que en algunas ocasiones nos encontraremos en la necesidad de conocer cuántas veces aparece un elemento en una lista. Para este fin utilizamos el método `count`, el cual recibe como argumento el elemento que queremos saber cuántas veces se encuentra repetido en la lista; `count` devuelve las veces que se encontró el elemento en la lista. Veamos el siguiente ejemplo:

```
1 lista=[1,2,3,4,5,"Juan",1,2,3,4,1,1]
2
3 print(lista.count(1))
```

Programa 6.36: Ejemplo count

La salida del programa 6.36 es:

```
4
```

Vemos una lista con varios elementos, dicha lista invoca al método `count` pasándole como argumento el elemento 1. Como dicha invocación se encuentra en un `print`, se imprime 4, que es el número de veces que aparece el elemento 1 en la lista.

6.3.8. Eliminar elementos de la lista

El método `pop`, si no recibe argumentos, tiene la peculiaridad de eliminar el último elemento de la lista. Veamos el siguiente ejemplo:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 lista.pop()
4
5 print(lista)
```

Programa 6.37: Ejemplo pop

La salida del programa 6.37 es:

```
[1, 2, 3, 4, 5]
```

Se tiene una lista con seis elementos, esta lista invoca al método `pop`, el cual no recibe argumentos. Por lo tanto, se elimina el último elemento de la lista ("Juan"). Al imprimir la lista, muestra únicamente los elementos que quedaron (ver salida del programa 6.37).

El método `pop` también puede recibir como argumento una posición, en la cual se encuentra el elemento que se quiere eliminar. Veamos el siguiente ejemplo:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 lista.pop(3)
4
5 print(lista)
```

Programa 6.38: Ejemplo pop con argumento

La salida del programa 6.38 es:

```
[1, 2, 3, 5, 'Juan']
```

En este caso se tiene una lista con seis elementos. Al método `pop` que es invocado por la lista se le pasa como argumento la posición 3, con lo que se le indica a `pop` que elimine de la lista el elemento que se encuentra en esa posición. Recordemos que el primer elemento de la lista se encuentra en la posición 0.

Para eliminar un elemento de la lista pasando como argumento el elemento, disponemos del método `remove`, el cual a diferencia de `pop`, no requiere conocer la posición del elemento. Veamos un ejemplo:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 lista.remove("Juan")
4
5 print(lista)
```

Programa 6.39: Ejemplo remove

Tenemos la lista con seis elementos y queremos eliminar el elemento "Juan". Entonces, la lista invoca a `remove`, el cual recibe como argumento a dicho elemento. Finalmente, al mostrar la lista notamos que el elemento "Juan" ya no se encuentra en la lista.

```
[1, 2, 3, 4, 5]
```

En algunas ocasiones será necesario vaciar por completo la lista. Para este fin disponemos del método `clear`, el cual al ser invocado realiza este vaciado. El método `clear` no recibe argumentos. Veamos el siguiente ejemplo:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 lista.clear()
4
5 print(lista)
```

Programa 6.40: Ejemplo clear

Para este ejemplo, utilizamos la lista de los ejemplos anteriores. Dicha lista invoca al método `clear`, el cual vacía por completo la lista.

```
[]
```

6.3.9. Invertir la lista

El método `reverse` permite invertir por completo una lista, es decir, el elemento 0 pasa a la ultima posición, y el que se encuentra en la última posición pasa a la posición 0. El mismo proceso aplica para los demás elementos. Veamos su uso:

```
1 lista=[1,2,3,4,5,"Juan"]
2
3 lista.reverse()
```

```
4
5 print(lista)
```

Programa 6.41: Ejemplo reverse

La salida del programa 6.41 es:

```
['Juan', 5, 4, 3, 2, 1]
```

6.3.10. Ordenar elementos de la lista

Otra operación interesante que podemos realizar con los elementos de una lista es ordenarlos ascendentemente, es decir, de menor a mayor. Para llevar a cabo dicho ordenamiento, utilizaremos el método `sort`, el cual es invocado por la lista. Veamos un ejemplo:

```
1 lista=[5,4,-7,9,0,1,3]
2
3 lista.sort()
4
5 print(lista)
```

Programa 6.42: Ejemplo sort

La salida del programa 6.42 es:

```
[-7, 0, 1, 3, 4, 5, 9]
```

Inicialmente tenemos una lista completamente desordenada, la cual invoca al método `sort`. No es necesario almacenar el resultado de ordenar la lista en una variable, ya que `sort` modifica directamente la lista que lo invocó (ver salida del programa 6.42).

En algunas ocasiones será necesario ordenar la lista de manera descendente, es decir, de mayor a menor. Para este fin, utilizamos de nueva cuenta el método `sort`, pero pasando la instrucción `reverse=True` como argumento. Veamos el ejemplo:

```
1 lista=[5,4,-7,9,0,1,3]
2
3 lista.sort(reverse=True)
4
5 print(lista)
```

Programa 6.43: Ejemplo sort 2

La salida del programa 6.43 es:

```
[9, 5, 4, 3, 1, 0, -7]
```

6.3.11. Recorrer listas con for

Supongamos que tenemos una lista sobre la que tenemos que realizar algún tipo de acción que implique acceder secuencialmente a cada uno de sus elementos. Sin ir más lejos, ¿cómo haríamos para imprimir, uno por uno, cada elemento que compone la lista?

El bucle `for` nos permite solucionar esto de un modo elegante. Obsérvalo en acción:

```
semana = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado
    ', 'Domingo']

for dia in semana:
    print(dia)
```

Programa 6.44: Recorre listas

El programa 6.44 tiene como salida:

```
Lunes
Martes
Miércoles
Jueves
Viernes
Sábado
Domingo
```

En python, es posible definir variables justo en el momento de ser utilizadas, por ejemplo la variable `dia` en el bucle. Los posibles valores para esta variable son asignados de los valores de la lista `semana`, tomados uno a uno desde la izquierda.

6.4. Tuplas

Las tuplas son secuencias muy similares a las listas, sin embargo, son inmutables; es decir, no se pueden modificar. No se podrá añadir, eliminar, ni modificar los elementos. Veamos entonces qué podemos hacer con las tuplas y qué ventajas tiene el utilizarlas.

6.4.1. Crear tuplas

En las listas se utilizan los corchetes [ ] para crearlas, ahora en las tuplas utilizaremos los paréntesis ( ). Para crear una tupla vacía simplemente le asignamos a una variable los paréntesis tanto de apertura como de cierre. Veamos el ejemplo:

```
tupla=()

print(tupla)
```

Programa 6.45: Ejemplo tupla vacía

La salida del programa 6.45 es:

```
()
```

Al igual que en las listas, en una tupla puede haber elementos de diferentes tipos de datos. Veamos el siguiente ejemplo:

```
tupla=(4,"Hola",6.78,[1,2,3,4])

print(tupla)
```

Programa 6.46: Ejemplo tupla con diferentes elementos

La salida del programa 6.46 es:

```
(4, 'Hola', 6.78, [1, 2, 3, 4])
```

En el ejemplo, se crea una tupla con elementos de diferentes tipos de datos, entre ellos: números, cadenas y listas.

Como ya habíamos mencionado, no se pueden agregar nuevos elementos a las tuplas. Por ejemplo, si intentamos usar el método `append` después de crear la tupla, tendríamos como salida un error. Tampoco se pueden modificar, ni eliminar los elementos existentes, cualquier intento de realizar alguna de estas operaciones en la tupla genera un error.

Entonces ¿qué podemos hacer con las tuplas?

6.4.2. Mostrar elementos de una tupla

A través de sus índices, se pueden mostrar los elementos de las tuplas. Como en las listas, hay que tener presente que el índice 0 sirve para referenciar al primer elemento de la tupla. Veamos el ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4])
2
3 print(tupla[1])
```

Programa 6.47: Mostrar elementos de la tupla

La salida del programa 6.47 es:

```
Hola
```

También podemos utilizar los **slicing** para desplegar simultáneamente más de un elemento de la tupla, ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4])
2
3 print(tupla[1:])
```

Programa 6.48: Mostrar simultáneamente elementos de la tupla

La salida del programa 6.48 es:

```
('Hola', 6.78, [1, 2, 3, 4])
```

6.4.3. Buscar elementos en la tupla

A través del operador `in` podemos saber si un elemento se encuentra en la tupla. Su uso es similar al que se manejó en las listas, es decir, `elemento in tupla`. En donde `elemento` es el elemento que se buscará y `tupla` es en donde lo va a buscar. Si el elemento se encuentra en la tupla, la expresión devuelve `True`. Veamos el ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4])
2
3 print(4 in tupla)
```

Programa 6.49: Operador in

La salida del programa 6.49 es:

```
True
```

Para conocer qué posición ocupa un elemento en la tupla disponemos del método `index`, el cual al ser invocado por la tupla devuelve dicha posición. Este método recibe como argumento el elemento que queremos saber su posición. Veamos el ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4])
2
3 print(tupla.index("Hola"))
```

Programa 6.50: Método index

La salida del programa 6.50 es:

```
1
```

6.4.4. Contar la aparición de un elemento en la tupla

En algunas ocasiones será necesario saber cuántas veces se repite un elemento en la tupla. Para poder contar el número de repeticiones Python ofrece el método `count`, el cual al ser invocado por la tupla recibe como argumento el elemento por contar, y devuelve las veces que dicho elemento aparece en la tupla, ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4],4)
2
3 print(tupla.count(4))
```

Programa 6.51: método count

La salida del programa 6.51 es:

```
2
```

6.4.5. Conocer la longitud de la tupla

La función `len` permite conocer la cantidad de elementos que se encuentran en una tupla. Dicha función recibe como argumento la tupla y devuelve la cantidad de elementos que ésta tiene. Veamos un ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4],4)
2
3 print(len(tupla))
```

Programa 6.52: función len

La salida del programa 6.52 es:

```
5
```

6.4.6. Transformar tuplas en lista

En algún momento podemos requerir que nuestra tupla se convierta en una lista, para este fin podemos utilizar la función `list`, la cual recibe como argumento la colección (en este caso una tupla) que queremos convertir a lista. Veamos el ejemplo:

```
1 tupla=(4,"Hola",6.78,[1,2,3,4],4)
2
3 lista=list(tupla)
4
5 print(lista)
```

Programa 6.53: función list

La salida del programa 6.53 es:

```
[4, 'Hola', 6.78, [1, 2, 3, 4], 4]
```

En este ejemplo, se crea la tupla con cinco elementos de todo tipo de datos. Luego a la función `list` se le pasa como argumento dicha tupla y se almacena la lista que nos devuelve `list` en la variable `lista`.

6.4.7. Transformar listas en tuplas

Podemos transformar una lista en una tupla mediante la función `tuple`. Su uso es muy sencillo, únicamente le pasamos la lista que queremos convertir a tupla y esta función devuelve la lista convertida en tupla. Veamos el ejemplo:

```
1 lista=[4,"Hola",6.78,[1,2,3,4],4]
2
3 tupla=tuple(lista)
4
5 print(tupla)
```

Programa 6.54: función tuple

La salida del programa 6.54 es:

```
(4, 'Hola', 6.78, [1, 2, 3, 4], 4)
```

6.4.8. Recorrer tuplas

Una característica de las tuplas es que son objetos iterables; es decir, con un sencillo bucle `for` podemos recorrer fácilmente todos los elementos.

```
1 t = ("a","cadena de texto",20,5.2)
2
3 for i in t:
4     print(i)
```

Programa 6.55: Recorrido de la tupla t

El programa 6.55 tiene como salida:

```
a
cadena de texto
20
5.2
```

La variable `i`, en el propio bucle, va a tomar, **uno a uno**, todos los elementos de la tupla `t`.

6.4.9. Ventajas de utilizar tuplas

Se mencionan a continuación algunas ventajas de usar tuplas:

1. Las tuplas son más rápidas que las listas y consumen menos memoria.

2. Hace que el código sea más seguro ya que "protege contra escritura" los datos que no necesitan ser cambiados. Usar un tupla en lugar de una lista es como tener una declaración de afirmación implícita de que estos datos son constantes y que se requiere un pensamiento especial (y una función específica) para anular eso.

3. Algunas tuplas se pueden usar como claves de diccionario (específicamente, tuplas que contienen valores inmutables como cadenas, números y otras tuplas).

6.5. Diccionarios

Los diccionarios en Python, al igual que las listas y las tuplas, nos permiten almacenar diferentes tipos de datos: cadenas, enteros, flotantes, booleanos, tuplas, listas e inclusive otros diccionarios.

Los diccionario son mutables, es decir, es posible modificarlos, agregar o quitar elementos de él.

A diferencias de las listas y de las tuplas, los diccionarios no se rigen por la regla de los índices. En este caso, todos los valores que se almacenen en el diccionario no corresponderán a un índice, sino a una clave.

Todos los valores necesitan tener una clave y cada clave necesita tener un valor.

Algo interesante para mencionar y a tener muy en cuenta es que una clave podrá ser cualquier objeto inmutable en Python, ya sea una cadena, un entero, un flotante o una tupla, etc.

Podemos decir entonces que un diccionario tiene dos elementos por posición: `clave` y `valor`. Y es importante mencionar que las claves no pueden ser duplicadas.

6.5.1. Crear un diccionario

Los diccionarios son creados con las llaves {}, a continuación, se muestra cómo crear un diccionario vacío.

```
diccionario={}
```

Para agregar elementos por default (es decir, elementos que no necesitan de un método en particular para ser agregados) a un diccionario, es necesario indicar la clave y el valor. Veamos un ejemplo en el que se requiere traducir un color de español a inglés.

```
1 diccionario={"azul":"blue", "rojo":"red", "verde":"green"}
2
3 print(diccionario)
```

Programa 6.56: Diccionario

La salida del programa 6.56 es:

```
{'azul': 'blue', 'rojo': 'red', 'verde': 'green'}
```

En este ejemplo se crea un diccionario (que se pudo llamar de cualquier forma) con tres elementos. El primer elemento tiene como `clave` la cadena "azul" y como `valor` la cadena "blue". La `clave` y el `valor` de un elemento son separados por dos puntos (:). Al imprimir el diccionario vemos los elementos que éste contiene.

6.5.2. Acceder al valor de un elemento del diccionario

Una vez creado el diccionario y que éste tenga elementos agregados, se puede acceder al valor de un elemento en particular a través de su `clave`. Para ello veamos el siguiente ejemplo:

```
1 diccionario={"azul":"blue", "rojo":"red", "verde":"green"}
2
3 print(diccionario["azul"])
```

Programa 6.57: Acceso a un elemento del diccionario

La salida del programa 6.57 es:

```
blue
```

El ejemplo presenta el diccionario con tres elementos que traducen un color de español a inglés. En dicho ejemplo se muestra como se dice "azul" en inglés. El acceso al `valor` del elemento se lleva a cabo utilizando [ ] corchetes y dentro de ellos la `clave`.

6.5.3. Agregar nuevos elementos al diccionario

En muchas ocasiones será necesario agregar elementos al diccionario posterior a su creación. Esta operación de agregar un nuevo elemento es muy sencilla, veamos un ejemplo:

```
1 diccionario={"azul":"blue", "rojo":"red", "verde":"green"}
2
3 diccionario["amarillo"]="yellow"
4
5 print(diccionario)
```

Programa 6.58: Agregar un elemento al diccionario

La salida del programa 6.58 es:

```
{'azul': 'blue', 'rojo': 'red', 'verde': 'green', 'amarillo': 'yellow'}
```

En el ejemplo inicialmente se tiene un diccionario con tres elementos. Posteriormente, se anexa en el diccionario el color amarillo con su traducción al inglés (ver línea 3 de programa 6.58). Finalmente se muestra el diccionario, el cual ya incluye al nuevo elemento.

Nota que para agregar una nueva `clave` (con su respectivo `valor`), se utilizan los corchetes "[ ]".

6.5.4. Modificar un elemento

Una herramienta muy útil en los diccionarios es poder modificar su `valor`. Dicha modificación es muy fácil, únicamente accedemos al elemento mediante su `clave` y agregamos el nuevo `valor`. Veamos el ejemplo:

```
1 diccionario={"azul":"blue", "rojo":"red", "verde":"green"}
2
3 diccionario["azul"]="BLUE"
4
5 print(diccionario)
```

Programa 6.59: Modificar un elemento del diccionario

La salida del programa 6.59 es:

```
{'azul': 'BLUE', 'rojo': 'red', 'verde': 'green'}
```

En este ejemplo, se modifica el `valor` del elemento que tiene como `clave` la cadena "azul". Dicha modificación consiste en cambiar el color de "blue" a "BLUE" (ver línea 3 de programa 6.59). Finalmente, se muestra el diccionario con modificación realizada.

6.5.5. Eliminar un elemento

Otra operación que podemos hacer con un diccionario es eliminar un elemento mientras el programa se ejecuta. Para realizar dicha eliminación, utilizamos la función `del`, la cual recibe como argumento el elemento por eliminar. Veamos un ejemplo:

```
1 diccionario={"azul":"blue", "rojo":"red", "verde":"green"}
2
3 del(diccionario["verde"])
4
5 print(diccionario)
```

Programa 6.60: Eliminar un elemento del diccionario

La salida del programa 6.60 es:

```
{'azul': 'blue', 'rojo': 'red'}
```

Inicialmente se tiene un diccionario con tres elementos, del cual se quiere eliminar el elemento que tiene como `clave` la cadena "verde". Únicamente pasamos este elemento

como argumento a la función `del` y listo. Imprimimos el diccionario y observamos que el elemento ya no se encuentra.

6.5.6. Cuando la clave no existe

Puede darse el caso de que se accede a un elemento que no se encuentra en el diccionario. Al realizar dicho acceso Python devuelve una excepción, veamos el ejemplo:

```
1 diccionario={"alejandro":{"edad":22,"estatura":1.73},"jose":[21,1.75],
    "maria":[22,1.67]}
2
3 print(diccionario["Luis"])
```

Programa 6.61: Acceso a un elemento que no existe en el diccionario

La salida del programa 6.61 es:

```
Traceback (most recent call last):
  File "/diccionario6.py", line 3, in <module>
    print(diccionario["Luis"])
KeyError: 'Luis'
```

En el ejemplo, se tiene un diccionario con tres elementos. Posteriormente, se intenta acceder al elemento que tiene como `clave` la cadena "Luis". Como dicho elemento no se encuentra en el diccionario, Python manda una Excepción `KeyError`.

Podemos manejar el error que manda Python de una manera muy sencilla y más elegante cuando el elemento no se encuentra en el diccionario. Veamos el ejemplo:

```
1 diccionario={"alejandro":{"edad":22,"estatura":1.73},"jose":[21,1.75],
    "maria":[22,1.67]}
2
3 print(diccionario.get("Luis","No existe esa persona"))
```

Programa 6.62: Acceso a un elemento que no existe en el diccionario 2

La salida del programa 6.62 es:

```
No existe esa persona
```

Como ya habíamos visto anteriormente, el elemento con `clave` "Luis" no existe, entonces podemos manejar esa excepción con el método `get`, el cual recibe como argumentos la clave del elemento que queremos obtener y un mensaje que le dé alguna indicación al usuario en caso de que dicho elemento no se encuentre.

La salida del programa ya no manda la excepción `KeyError`, en su lugar muestra la cadena "No existe esa persona".

6.5.7. Búsqueda directa

Podemos hacer una búsqueda para determinar si un elemento se encuentra en el diccionario. Para realizar dicha búsqueda empleamos el operador `in`. Veamos el ejemplo:

```
1 diccionario={"alejandro":{"edad":22,"estatura":1.73},"jose":[21,1.75],
    "maria":[22,1.67]}
```

```
2
3 print("alejandro" in diccionario)
```

Programa 6.63: Operador in en diccionarios

La salida del programa 6.63 es:

```
True
```

Como podemos ver en el ejemplo, se opera con `in` indicándole la `clave` que queremos buscar en el diccionario. Si dicha `clave` se encuentra, esta expresión devuelve `True`, en caso contrario, devuelve `False`.

6.5.8. Mostrar los valores del diccionario

Para mostrar los valores de los elementos de un diccionario disponemos del método `values`, el cual es invocado por el diccionario. Este método no requiere argumentos, veamos el ejemplo:

```
1 diccionario={"alejandro":{"edad":22,"estatura":1.73},"jose":[21,1.75],
    "maria":[22,1.67]}
2
3 print(diccionario.values())
```

Programa 6.64: Método values

La salida del programa 6.64 es:

```
dict_values([{'edad': 22, 'estatura': 1.73}, [21, 1.75], [22, 1.67]])
```

6.5.9. Cantidad de elementos del diccionario

La función `len` determina la cantidad de elementos del diccionario, su uso es muy simple. Esta función recibe como argumento el diccionario. Veamos el ejemplo:

```
1 diccionario={"alejandro":{"edad":22,"estatura":1.73},"jose":[21,1.75],
    "maria":[22,1.67]}
2
3 print(len(diccionario))
```

Programa 6.65: Función len en diccionarios

La salida del programa 6.65 es:

```
3
```

6.5.10. Vaciar el diccionario

Podemos eliminar todos los elementos del diccionario utilizando el método `clear`, el cual es invocado por el diccionario que queremos vaciar. Dicho método no recibe argumentos, veamos el ejemplo:

```
1 diccionario={"alejandro":{"edad":22,"estatura":1.73},"jose":[21,1.75],
    "maria":[22,1.67]}
2
3 diccionario.clear()
4
5 print(diccionario)
```

Programa 6.66: Método clear en diccionarios

La salida del programa 6.66 es:

```
{}
```

6.5.11. Recorrer un diccionario con for

En numerosas ocasiones necesitaremos iterar a través de un diccionario mediante un bucle `for`.

El código 6.67 muestra como iterar sobre las claves de un diccionario:

```
1 diccionario = {'plátano':'amarillo', 'fresa':'roja', 'manzana':'verde'
    }
2
3 frutas=diccionario.keys()
4
5 for fruta in frutas:
6     print(fruta)
```

Programa 6.67: Recorrido a través de las claves

La salida del programa 6.67 es:

```
plátano
fresa
manzana
```

Primero se crea el `diccionario` con algunos datos de frutas (ver línea 1 de código 6.67). En seguida, se almacenan las claves del diccionario en la variable `frutas` (llevar a cabo este almacenamiento es posible gracias al método `keys`). Finalmente, mediante el bucle `for` se van mostrando una por una las claves del diccionario.

Otra forma de iterar en un diccionario es a través de sus valores. El código 6.68 muestra como realizar dicha iteración.

```
1 diccionario = {'plátano':'amarillo', 'fresa':'roja', 'manzana':'verde'
    }
2
3 frutas=diccionario.values()
4
5 for fruta in frutas:
6     print(fruta)
```

Programa 6.68: Recorrido a través de los valores

La salida del programa 6.68 es:

```
amarillo
roja
verde
```

Como puedes notar los dos últimos códigos son similares, lo único que cambia es que para iterar a través de los valores del diccionario se emplea el método `values`.

Finalmente, si necesitamos iterar sobre las claves y sus correspondientes valores simultáneamente utilizamos el método `items`:

```
diccionario = {'plátano':'amarillo', 'fresa':'roja', 'manzana':'verde'
    }

elementos=diccionario.items()

for fruta, color in elementos:
    print(fruta,"->",color)
```

Programa 6.69: Recorrido a través de los elementos del diccionario

La salida del programa 6.69 es:

```
plátano -> amarillo
fresa -> roja
manzana -> verde
```

6.6. Conjuntos

Los conjuntos son grupos de elementos desordenados, donde su principal característica es que no puede haber elemento duplicados.

6.6.1. Crear un conjunto

Para crear un conjunto podemos hacerlo de la siguiente manera:

```
conjunto=set()
```

Al igual que en los diccionarios, los conjuntos utilizan las llaves "{}" para contener elementos; sin embargo, para poder diferenciarlos de los diccionarios es necesario utilizar la función `set` al momento de crearlos.

Como ya habíamos mencionado, se crea con `set` un conjunto vacío. Sin embargo, si al conjunto le agregas elementos desde su creación, ya no es necesario usar la función `set`.

A continuación, veremos cómo crear un conjunto vacío:

```
conjunto=set()

conjunto={}

print(conjunto)
```

Programa 6.70: Conjunto vacío

La salida del programa 6.70 es:

```
{}
```

En un conjunto se pueden agregar diferentes tipos de elementos (cadenas, enteros, flotantes):

```
1 conjunto={1,2,3,"Hola",345.5}
2
3 print(conjunto)
```

Programa 6.71: Conjunto con elementos

La salida del programa 6.71 es:

```
{1, 2, 3, 345.5, 'Hola'}
```

Sin embargo, en un conjunto no se pueden agregar otros tipos de colecciones como las listas.

En los conjuntos no puede haber elementos duplicados, si agregamos elementos duplicados, no se produce un error, pero el conjunto no los almacena.

```
1 conjunto={1,2,3,"Hola",345.5,1,2,3}
2
3 print(conjunto)
```

Programa 6.72: Conjunto con elementos duplicados

La salida del programa 6.72 es:

```
{1, 2, 3, 'Hola', 345.5}
```

Es importante notar que los elementos agregados al conjunto se muestran de forma desordenada.

Para agregar más elementos al conjunto se dispone del método `add`, el cual recibe como argumento el elemento que se va a agregar. Veamos el ejemplo:

```
1 conjunto={1,2,3,"Hola",345.5}
2
3 conjunto.add(5)
4
5 print(conjunto)
```

Programa 6.73: Método add

La salida del programa 6.73 es:

```
{1, 2, 3, 5, 'Hola', 345.5}
```

Nota que el conjunto no necesariamente se agregó al final, recordemos que el conjunto lo agrega de forma desordenada.

6.6.2. Eliminar elementos del conjunto

Para eliminar elementos del conjunto Python ofrece el método `discard`, el cual recibe como argumento el elemento que se eliminará.

```
conjunto={1,2,3,"Hola",345.5}

conjunto.discard(3)

print(conjunto)
```

Programa 6.74: Método discard

La salida del programa 6.74 es:

```
{1, 2, 345.5, 'Hola'}
```

Ahora bien, para vaciar el conjunto, es decir, eliminar todos los elementos, Python ofrece el método `clear`, sin argumentos. Veamos el ejemplo:

```
conjunto={1,2,3,"Hola",345.5}

conjunto.clear()

print(conjunto)
```

Programa 6.75: Método clear

La salida del programa 6.75 es:

```
set()
```

6.6.3. Buscar un elemento

Se puede saber si un elemento existe en el conjunto, esto es posible gracias al operador `in`. Su funcionamiento es tal cual lo hemos venido trabajando en otras colecciones, es decir, `elemento in conjunto`. Donde `elemento` es el elemento que queremos buscar, y `conjunto` el conjunto donde se realiza la búsqueda. Esta expresión devuelve `True` si el elemento se encuentra, en caso contrario, devuelve `False`. Veamos el ejemplo:

```
conjunto={1,2,3,"Hola",345.5}

print(3 in conjunto)
```

Programa 6.76: Operador in en conjuntos

La salida del programa 6.76 es:

```
True
```

La búsqueda también se puede hacer a la inversa, es decir, determinar si el elemento no se encuentra en el conjunto. Esta búsqueda es posible utilizando el operador `not in`, el cual devuelve `True` si el elemento no se encuentra, y `False` si el elemento se encuentra en el conjunto. Veamos el ejemplo:

```
conjunto={1,2,3,"Hola",345.5}

print(3 not in conjunto)
```

Programa 6.77: Operador not in en conjuntos

La salida del programa 6.77 es:

```
False
```

6.6.4. Igualdad de conjuntos

Se puede determinar si un conjunto es igual a otro conjunto mediante el operador ==. Veamos el ejemplo:

```
1 a={1,2,3}
2
3 b={3,4,5}
4
5 print(a == b)
```

Programa 6.78: Operador de igualdad

La salida del programa 6.78 es:

```
False
```

Se tienen dos conjuntos (a y b), con el mismo número de elementos pero con elementos diferentes. Al aplicar el operador ==, éste devuelve `False`, ya que son conjuntos diferentes.

6.6.5. Conocer la cantidad de elementos de un conjunto

Para conocer la cantidad de elementos de un conjunto Python ofrece la función `len`, la cual recibe como argumento el conjunto. Veamos el ejemplo:

```
1 a={1,2,3}
2
3 b={3,4,5}
4
5 print(len(a))
```

Programa 6.79: Función len en conjuntos

La salida del programa 6.79 es:

```
3
```

6.6.6. Unión de conjuntos

Como vimos en el capítulo 1, la **unión** de dos o más conjuntos es el conjunto formado por todos los elementos que pertenecen a ambos conjuntos. En Python, podemos realizar dicha **unión** empleando el operador barra (|). Veamos el ejemplo:

```
1 a={1,2,3}
2
3 b={3,4,5}
4
5 c=a|b
```

```
6
7 print(c)
```

Programa 6.80: Unión de conjuntos

La salida del programa 6.80 es:

```
{1, 2, 3, 4, 5}
```

En este ejemplo, se tienen inicialmente dos conjuntos con tres elementos cada uno (`a` y `b`), luego se crea el conjunto `c`, al cual se le asigna la unión de `a` y `b`. Finalmente, se muestra el conjunto `c` con los elementos correspondientes.

6.6.7. Intersección de conjuntos

Recordando, la **intersección** de dos o más conjuntos es el conjunto formado por lo elementos que tienen en común ambos conjuntos. En Python podemos realizar dicha intersección empleando el operador &. Veamos el ejemplo:

```
1 a={1,2,3}
2
3 b={3,4,5}
4
5 c = a & b
6
7 print(c)
```

Programa 6.81: intersección de conjuntos

La salida del programa 6.81 es:

```
{3}
```

6.6.8. Diferencia de conjuntos

Podemos decir que la diferencia de dos conjuntos (`a - b`), nos devuelve el conjunto con los elementos de `a` que no están en `b`. El operador resta (-) lleva a cabo dicha función. Veamos un ejemplo:

```
1 a={1,2,3}
2
3 b={3,4,5}
4
5 c = a - b
6
7 print(c)
```

Programa 6.82: Diferencia de conjuntos

La salida del programa 6.82 es:

```
{1, 2}
```

6.6.9. Diferencia simétrica

Son los elementos que están en `a` y en `b`, pero no en ambos. El operador ∧ permite realizar la diferencia simétrica. Veamos el ejemplo en Python:

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

b={3,4,5}

c = a ^ b

print(c)
```

Programa 6.83: Diferencia simétrica de conjuntos

La salida del programa 6.83 es:

```
{1, 2, 4, 5}
```

6.6.10. Determinar si un conjunto es subconjunto de otro

Para determinar si un conjunto es subconjunto de otro, disponemos del método `issubset`, el cual es invocado por el posible "subconjunto" y recibe como argumento el conjunto en el que se quiere buscar al subconjunto. Su función es de la siguiente manera:

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

b={3,4,5}

c={1,2,3,4,5}

print(a.issubset(c))
```

Programa 6.84: Método issubset

La salida del programa 6.84 es:

```
True
```

En el ejemplo tenemos el conjunto `a`, `b`, `c`; y se quiere determinar si `a` es subconjunto de `c`. Como efectivamente `a` es subconjunto de `c`, el método `issubset` devuelve `True`.

6.6.11. Recorrer un conjunto con for

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos `list` o `tuple`. Es por ello que no se puede acceder a los elementos a través de un índice. Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle `for`:

```
conjunto = {1, 3, 2, 9, 3, 1}

for e in conjunto:
```

```
4     print(e)
```

Programa 6.85: Recorrido de un conjunto

La salida del programa 6.85 es:

```
1
2
3
9
```

6.7. Conclusiones

En este capítulo se definieron las cadenas, listas, tuplas, diccionarios y conjuntos, así como las operaciones disponibles para trabajar con estas estructuras de datos.

6.8. Ejercicios cadenas y colecciones

1. Hacer un programa donde se deberá imprimir por la consola la palabra con más caracteres de dos palabras dadas. En el caso de que ambas palabras tengan la misma cantidad de caracteres, deberás mostrar el mensaje "Son iguales".

2. Hacer un programa en Python para detectar si una frase introducida por el usuario finaliza con un punto "." o no. Deberás imprimir por consola una de las siguientes opciones; "Termina con un punto" o por el contrario "No termina con un punto".

3. Hacer un programa que determine si una palabra o frase es palíndroma. Una cadena palíndroma se lee igual de izquierda a derecha que de derecha a izquierda. Ejemplos:

   - oso
   - reconocer
   - anita lava la tina

4. Hacer un programa donde se cuente cada una de las vocales en una cadena, mostrar el conteo de las apariciones de cada vocal.

5. Escribe un programa donde tenga una lista y que, a continuación, elimine los elementos repetidos, por último, mostrar la lista.

6. Escribe un programa que tenga dos listas y que, a continuación, cree las siguientes listas (en las que no debe haber repeticiones):

   - Lista de elementos que aparecen en las dos listas.
   - Lista de elementos que aparecen en la primera lista, pero no en la segunda.

- Lista de elementos que aparecen en la segunda lista, pero no en la primera.
- Lista de elementos que aparecen en ambas listas

Capítulo 7

Funciones

Una función es un bloque de código que se puede usar varias veces en varios puntos distintos de nuestro programa. Nos evita tener que repetir código y facilita la lectura y el mantenimiento del programa.

Para declarar una función usamos la palabra reservada `def` seguida del nombre de la función, un paréntesis para pasarle datos a la función (no en todos los casos requiere datos) y dos puntos. A continuación de esto se encuentra el código de la función (es importante dejar un espaciado donde inicia el código, ya que de esta manera Python identifica que dicho código pertenece a la función).

Los nombres de las funciones siguen las mismas reglas que los nombres de las variables. Es decir, deben comenzar por una letra o un guion bajo, no pueden contener espacios ni pueden comenzar por un número, pero sí contener números. Se recomienda usar nombres en minúsculas y, si constan de más de una palabra, éstas deberán estar separadas por guiones bajos.

Una función debe ser invocada para que se ejecute. Para invocar a la función se usa su nombre seguido de un paréntesis. Cada vez que se llama de este modo a la función, se ejecuta su código:

```
1 def lugar():
2     print("Tabasco", end=" ")
3
4 print("Yo nací en", end=" ")
5 lugar()
6 print("mi padre nació en", end=" ")
7 lugar()
8 print("mis amigos nacieron en", end=" ")
9 lugar()
```

Programa 7.1: Función lugar

Cuando ejecutamos el programa, tenemos como salida:

```
Yo nací en Tabasco mi padre nació en Tabasco mis amigos nacieron en Tabasco
```

En este ejemplo (ver programa 7.1), cada vez que invoquemos en nuestro programa a la función `lugar()`, se imprimirá “Tabasco” (ver salida de programa 7.1). Si hemos

de hacerlo varias veces como en el ejemplo, nos ahorramos escribir el `print` cada vez, lo que redunda además en un código más limpio, claro y legible.

7.1. Funciones con y sin retorno de valor

Existen dos tipos de funciones, aquellas que realizan una tarea y no devuelven ningún valor y aquellas que tras realizar la tarea devuelven un valor o resultado. En esta sección analizaremos primero las funciones que no retornan ningún valor y posteriormente las funciones que retornan un valor.

7.1.1. Funciones sin retorno de valor

Nada mejor que un ejemplo para poder ilustrar este tipo de funciones:

```
1 def saludar():
2     print("Hola amigo")
3
4 saludar()
```

Programa 7.2: Ejemplo función sin retorno de valor

El código anterior muestra una función llamada `saludar`, la cual no recibe ningún argumento o dato de entrada; y en su interior contiene una línea de código que imprime un saludo. Posteriormente esta función es invocada generando la salida siguiente:

```
Hola amigo
```

Algunas funciones que no retornan un valor también pueden recibir datos de entrada como argumentos. Estos datos de entrada son procesados en el interior de la función. Veamos el siguiente ejemplo:

```
1 def saludar(nombre):
2     print(f"Hola {nombre}")
3
4 saludar("Carlos")
```

Programa 7.3: Ejemplo función sin retorno de valor con argumento

La función `saludar` recibe como argumento una cadena, la cual es mostrada junto con la palabra "Hola". En este caso, la cadena que se pasa como argumento es la palabra "Carlos", que combinada con la palabra que se encuentra en la función genera la salida siguiente:

```
Hola Carlos
```

Otro ejemplo más práctico del uso de funciones sin retorno de valor y con argumento de entrada, lo visualizamos en la siguiente función, la cual tiene como finalidad mostrar la tabla de multiplicar de un número pasado como argumento.

```
1 def tabla_multiplicar(i):
2     for j in range(1,11):
3         print(f"{i} X {j} = {i*j}")
```

```
4
5 tabla_multiplicar(8)
```

Programa 7.4: Tabla de multiplicar

Cuando ejecutamos el programa, tenemos como salida:

```
8 X 1 = 8
8 X 2 = 16
8 X 3 = 24
8 X 4 = 32
8 X 5 = 40
8 X 6 = 48
8 X 7 = 56
8 X 8 = 64
8 X 9 = 72
8 X 10 = 80
```

7.1.2. Funciones con retorno de valor

A diferencia de las funciones que no retornan un valor, las funciones con retorno de valor hacen uso de la palabra reservada `return` para devolver un resultado, el cual es producto de la tarea realizada por la función.

A continuación, se presenta una función llamada `multiplicar`, que recibe como argumentos (datos de entrada) dos números, los cuales son multiplicados entre sí y el resultado es devuelto por la función.

```
1 def multiplicar(num1,num2):
2     mult=num1*num2
3     return mult
4
5 print(multiplicar(3,4))
```

Programa 7.5: Multiplica dos números

Cuando ejecutamos el programa, tenemos como salida:

```
12
```

Se debe notar que la función es invocada dentro de un `print`, con los valores 3 y 4. Lo que genera que se imprima un 12. Es importante hacer notar que también el valor devuelto por la función pudo haber sido almacenado en una variable en lugar de mostrarlo directamente con `print`.

Una función también puede retornar más de un valor (de diferentes tipos, por ejemplo, cadenas, enteros, listas, etc.), los cuales son separados por comas al momento de realizar el retorno (`return`). Ademas, al realizar dicha separación con comas, aunque no se utilicen los paréntesis, automáticamente los valores se integran en una tupla. Veamos el siguiente ejemplo:

```
1 def funcion():
2     return "Hola", 45, [1,2,3]
3
4 print(funcion())
```

Programa 7.6: Función retorna varios valores

Cuando ejecutamos el programa, tenemos como salida:

```
('Hola', 45, [1, 2, 3])
```

En el ejemplo anterior podemos ver la función `funcion`, la cual retorna varios elementos de diferentes tipos (es decir, una cadena, un entero y una lista). Al ser invocada dicha función, vemos los elementos integrados en una `tupla`.

Retomando el ejemplo anterior, en el cual la función retorna más de un valor, también se pueden almacenar estos valores (al momento de invocar la función) en variables. Es importante que haya tantas variables como valores retornados. Veamos el siguiente ejemplo:

```
1 def funcion():
2     return "Hola", 45, [1,2,3]
3
4 c,n,l=funcion()
5 print(c)
6 print(n)
7 print(l)
```

Programa 7.7: Función retorna varios valores 2

Cuando ejecutamos el programa, tenemos como salida:

```
Hola
45
[1, 2, 3]
```

7.2. Argumentos y parámetros

Hasta el momento no se ha marcado una diferencia entre argumento y parámetro, es decir, ambos términos se han manejado por igual. Sin embargo, existe una diferencia sutil para diferenciar estos conceptos. Dicha diferencia es la siguiente:

1. Cuando la función es invocada, los datos que se proporcionan a la función se les denomina argumentos.

2. En el prototipo de la función, es decir, en donde la función es creada, los valores que recibe de entrada esta función se les denomina parámetros.

Veamos el siguiente ejemplo:

```
1 def restar(num1,num2):
2     return num1-num2
3
4 print(restar(4,3))
```

Programa 7.8: Ejemplo argumentos y parámetros

Cuando ejecutamos el programa, tenemos como salida:

```
1
```

En el ejemplo se presenta la función `restar`, la cual se encarga de restar los valores proporcionamos como entradas. Cuando la función es invocada, se le proporcionan los valores 4 y 3, a los cuales en ese momento se les denomina argumentos. Por su parte, el prototipo de la función `resta` recibe como datos de entrada `num1` y `num2`. A estos datos se les denomina parámetros.

Es importante hacer notar que tanto argumentos como parámetros tienen un orden. En el ejemplo anterior, `num1` toma el valor de 4 y `num2` toma el valor de 3. En dicho ejemplo, si se hubiera querido restar 4 de 3, se hubiera invocado la función se la siguiente manera:

```
restar(3,4)
```

Lo que hubiera devuelto como resultado -1.

Podemos decir, entonces, que el primer argumento va a estar referenciado por el primer parámetro; y así para los demás elementos que reciba como entrada la función.

7.3. Argumentos por valor y por referencia

Algunos lenguajes de programación soportan el paso de argumentos por valor y por referencia. La idea detrás de estos conceptos es la siguiente:

1. **Paso por valor**: se pasa como argumento un dato o variable (por ejemplo, un entero, un flotante, una cadena, etc.), la cual no se modifica su valor al ejecutarse la función. Por ejemplo:

```
1 def cuadrado(n):
2     n=n**2
3
4 n=5
5 cuadrado(n)
6 print(n)
```

Programa 7.9: Paso valor

Cuando ejecutamos el programa, tenemos como salida:

```
5
```

En este ejemplo se pasa la variable `n` con un valor inicial de 5 a la función `cuadrado`. Cuando ese valor `n` entra en dicha función, se eleva al cuadrado su valor, es decir, si `n` valía 5, en la función vale 25. Sin embargo, al regresar a la ejecución después de salir de la función se imprime `n`, con el valor de 5. Esto quiere decir que el valor de `n` solo fue modificado al interior de la función, pero no por toda la ejecución del programa.

2. **Paso por referencia**: La idea detrás del paso por referencia es que al pasar un valor como argumento a una función y modificarlo, dicha modificación se

mantenga aun cuando se salga de la función. Este concepto no aplica a todos los tipos de datos en Python, por ejemplo, en los enteros si se quisiera modificar el valor y que este valor modificado se mantenga se tendría que hacer algo como lo siguiente:

```
def cuadrado(n):
    return n**2

n=5
n=cuadrado(n)
print(n)
```

Programa 7.10: Paso referencia con enteros

Cuando ejecutamos el programa, tenemos como salida:

```
25
```

Inicialmente `n` vale 5, este valor se pasa como argumento a la función, la cual en su interior retorna (`return`) el doble de 5, o sea, 10. Ese valor retornado se almacena en la variable `n`, con lo que `n` ya vale 10 al exterior de la función.

Las listas son estructuras que se pasan por referencia en su pleno concepto. Esto significa que, si sus valores cambian al interior de una función, sin necesidad de que los nuevos valores de la lista sean retornados, éstos se mantendrán al exterior de la función. Veamos el ejemplo siguiente:

```
def cuadrados(numeros):
    for i, n in enumerate(numeros):
        numeros[i]**=2

n=[5,10,15,20]
cuadrados(n)
print(n)
```

Programa 7.11: Paso referencia con listas

Cuando ejecutamos el programa, tenemos como salida:

```
[25, 100, 225, 400]
```

7.4. Funciones recursivas

Las funciones recursivas son funciones que se llaman a sí mismas durante su ejecución. Ellas funcionan de forma similar a las iteraciones, pero se debe de planificar el momento en que dejan de llamarse a sí mismas o tendrá una función recursiva infinita. El método tradicional de planificación consiste en establecer un caso base, que en código sería una estructura selectiva en la que se define la condición que se debe cumplir para detener la recursión.

Estas funciones se estilan utilizar para dividir una tarea en subtareas más simples de forma que sea más fácil abordar el problema y solucionarlo.

7.4.1. Función recursiva sin retorno

Un ejemplo de función recursiva sin retorno es el cálculo del Máximo Común Divisor de dos números, utilizando el algoritmo de Euclides.

Partamos de la bien conocida definición de MCD, que enuncia lo siguiente:

1. $MCD(a, b) = a$ si $b = 0$
2. $MCD(a, b) = MCD(b, a\,\%b)$ si $b > 0$

Ejemplo

A partir de la definición anterior, cálcular el MCD de 7 y 5, donde $a = 7$ y $b = 5$ (ver figura 7.1). Como b es distinto de cero se aplica la definición 2, a partir de la cual se invoca de nueva cuenta la función MCD, sin embargo, en esta nueva invocación el valor de $a = 5$ y $b = 2$ ($7\,\%5{=}2$). De nueva cuenta b es distinto de cero, por lo que se invoca la función MCD con $a = 2$ y $b = 1$ ($5\,\%2{=}1$). Como b sigue siendo distinto de cero, se incova de nuevo la función con $a = 1$ y $b = 0$ ($2\,\%1{=}0$), y termina la recursión, ya que b es cero, con lo que se aplica la definición 1 y se imprime 1 (es decir, a).

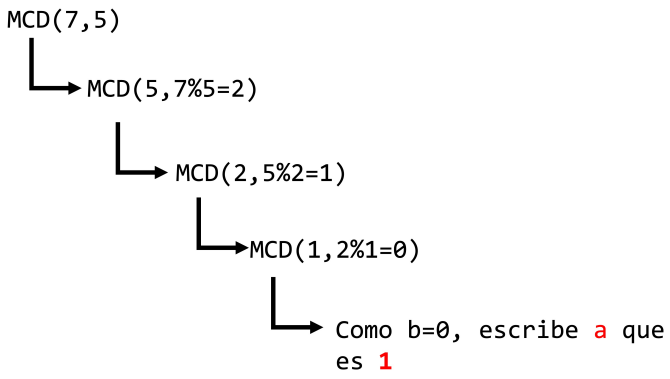


Figura 7.1: Llamadas recursivas para el cálculo del MCD de 7 y 5

El programa en Python, queda de la siguiente forma:

```
1 def MCD(a,b):
2     #caso base
3     if b==0:
4         print(a)
5     #caso recursivo
6     else:
7         MCD(b, a % b)
8 
9 MCD(7,5)
```

Programa 7.12: MCD recursivo

Cuando ejecutamos el programa, tenemos como salida:

```
1
```

7.4.2. Función recursiva con retorno

Un ejemplo de función recursiva con retorno es el cálculo del factorial de un número, el cual consiste en multiplicar desde n (el número del que se desea calcular) hasta 1. Por ejemplo, factorial de 4 (comunmente expresado como 4!) es 4*3*2*1=24. A partir de este ejemplo, se puede decir que factorial de un número n es $n * n - 1!$.

Partiendo del párrafo anterior se presenta la siguiente definición:

1. fac(n) devuelve 1 si $n = 0$ ó $n = 1$
2. fac(n) devuelve n*fac($n - 1$) si n>1

Ejemplo. Calcular el factorial de 4 con base en la definición anterior.

En la figura 7.2, se muestran las llamadas a la función `fac`, la primera llamada es `fac(4)`, como 4 es distinto de cero y de uno, se aplica la definición 2, es decir, `fac(4)` devuelve `4*fac(3)`. El valor 3 (pasado como argumento a la función `fac`) es distinto de cero y de uno, por lo que se llama de nueva cuenta a la función, es decir, devuelve `3*fac(2)`. El valor 2 (pasado como argumento a la función `fac`) es distinto de cero y de uno, por lo que se llama de nuevo a la función, es decir, devuelve `2*fac(1)`. Como el argumento proporcionado a la función `fac` es 1, se aplica la primera definición y se devuelve 1. Este 1 se multiplica por 2 (antepenúltima llamada), lo cual da 2, se devuelve 2 a la anteantepenúltima llamada y se múltiplica por 3, que es 6, y así sucesivamente. El proceso continúa hasta llegar a la primera llamada de `fac` y se obtiene 24.

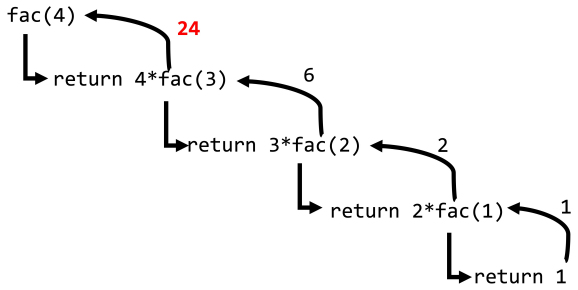


Figura 7.2: Llamadas recursivas para el cálculo del factorial de 4

El programa en Python para calcular el factorial es:

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

print(factorial(4))
```

Programa 7.13: Factorial recursivo

Cuando ejecutamos el programa, tenemos como salida:

```
24
```

7.5. Conclusiones

Las funciones son implementaciones que dividen un problema en problemas más pequeños, lo cual posibilita una mejor comprensión del código; así como también problemas más pequeños por resolver originan menos errores. A través de los ejemplos presentados en este capítulo podemos aprender a utilizar funciones en python, para resolver diversos problemas de la vida real.

7.6. Ejercicios

1. Crea una función llamada `mcm` que calcule el mínimo común múltiplo de dos números y lo retorne.

2. Se desea calcular independientemente la suma de los números pares comprendidos entre `n` y `m` para `m>n`.

   La estructura de la función es:

   ```
   def suma_pares(n,m):
   ```

   Devuelve la suma de los pares entre `n` y `m`

3. Crear una función que recibe un número `N` y escriba su cuadrado.

   La estructura de la función es:

   ```
   def cuadrado(N):
   ```

   Devuelve el cuadrado de `N`

4. Analiza y explica la lógica de la función que calcula el MCD de varios elementos:

```
def MCD_extendido(lista):
    lista.sort()
    a = lista[0]
    for i in range(1,len(lista)):
        res=-1
        b = lista[i]
        mayor=max(a,b)
        menor=min(a,b)
        while res!=0:
            res=mayor%menor
            mayor=menor
            menor=res

        a=mayor

```

```
16      return mayor
17
18  print(MCD_extendido([20,30,10,100]))
```

5. Crear una función que reciba como parámetro un número y diga si el número es primo o no es primo.

6. Hacer un programa que pida la anchura y altura de un rectángulo y con ayuda de una función lo dibuje con *.
   Ejemplo:
   ancho=7
   alto=3

```
* * * * * * *
* * * * * * *
* * * * * * *
```

Capítulo 8

Introducción a la complejidad computacional

8.1. Análisis de algoritmos

El análisis de algoritmos es una actividad que sirve para evaluar la calidad en términos de tiempo y memoria empleados en la ejecución del algoritmo. Se hace con el fin de comparar algoritmos de igual función pero diferente procedimiento.

Consideremos dos programas `P1` y `P2` para resolver el mismo problema, el cual consiste en calcular la suma de los primeros `n` números naturales. ¿Cuál de los dos es mejor?

Como primera aproximación podríamos implementar ambos programas y medir el tiempo que cada uno de ellos consume en resolver el problema para diversos valores de `n`.

El programa 8.1 implementa dos funciones (`suma_P1` y `suma_P2`), cada una de ellas resuelve el problema de sumar los primeros `n` números naturales.

```
1  import time
2
3  def suma_P1(n):
4      suma = (n*(n+1))/2
5      return suma
6
7  def suma_P2(n):
8      con=1
9      suma=0
10
11     while(con<=n):
12         suma=suma+con
13         con+=1
14
15     return suma
16
17 n=20000000
```

```
18 comienzo=time.time()
19 suma_P1(n)
20 final=time.time()
21 print(final-comienzo)
22
23 comienzo=time.time()
24 suma_P2(n)
25 final=time.time()
26 print(final-comienzo)
```

Programa 8.1: Suma de primeros n números

Veamos entonces cuánto tiempo tarda cada una de ellas en resolver el problema, para diversos valores de `n`.

Programa	**n=1000**	**n=10000**	**n=100000**	**n=1000000**
P1	$2.14e-06$	$3.09e-06$	$4.05e-06$	$4.05e-06$
P2	$9.39e-05$	0.0016	0.0190	0.1929

Tabla 8.1: Tiempos de ejecución de P1 y P2

De forma gráfica los resultados son los siguientes:

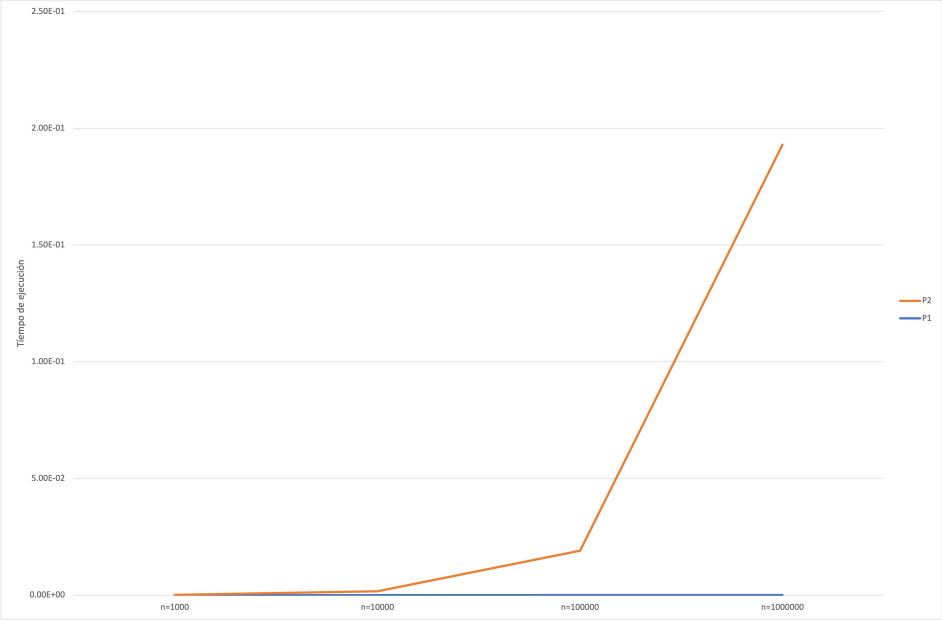


Figura 8.1: Tiempos de ejecución de P1 y P2

Características del equipo donde se realizó la prueba a los programas P1 y P2

- Macbook Air, con procesador Intel Core i5 de doble núcleo a 1.6 GHz y con 4 MB de caché L3 compartida.

- 8 GB de memoria SDRAM a 2133 MHz
- SSD basado en PCIe de 256 GB

De acuerdo con el análisis realizado en esta primera aproximación, `P1` es más eficiente que `P2`. Sin embargo, si utilizamos esta estrategia de solución para determinar la eficiencia de los algoritmos nos limitamos a depender de la computadora (tanto del hardware como del software) donde se prueben los programas. Así mismo, sería poco factible implementar todas las soluciones que puedan haber para el mismo problema con el fin de determinar la más eficiente.

Lo ideal sería entonces establecer una medida de la calidad de los algoritmos, que permita compararlos sin la necesidad de implementarlos y cronometrarlos. Esto implica asociar a cada algoritmo una función matemática que mida su eficiencia.

8.1.1. Tiempo de ejecución

Se define como el tiempo empleado por un algoritmo en procesar una entrada de tamaño `n` y producir una solución al problema. El ideal es encontrar una función matemática `T(n)` que describa de manera exacta dicho tiempo empleado.

Ejemplo: consideremos los programas `P1` y `P2` que suman los primeros `n` números naturales. Supongamos que fijamos `n=5` y que la evaluación de cada línea del programa tarda t microsegundos. En el caso del programa `P2` tendríamos lo siguiente:

Instrucción	**tiempo en** *ms*
`con=1`	*t1 ms*
`suma=0`	*t1 ms*
`1<=5`	*t2 ms*
`suma=suma+con`	*t3 ms*
`con+=1`	*t3 ms*
`2<=5`	*t2 ms*
`suma=suma+con`	*t3 ms*
`con+=1`	*t3 ms*
`3<=5`	*t2 ms*
`suma=suma+con`	*t3 ms*
`con+=1`	*t3 ms*
`4<=5`	*t2 ms*
`suma=suma+con`	*t3 ms*
`con+=1`	*t3 ms*
`5<=5`	*t2 ms*
`suma=suma+con`	*t3 ms*
`con+=1`	*t3 ms*
Total	$2t1\ ms + 5t2\ ms + 10t3\ ms = t4\ ms$

Ahora veamos qué pasa con el programa `P1`:

Instrucción	**tiempo en** *ms*
$(5 * 6)/2$	t *ms*
Total	$1t$ *ms*

A partir de este análisis podemos decir que para `P1` el tiempo de ejecución es `T(n)=1` (ya que solo es una instrucción la que se ejecuta, independientemente del tamaño de `n`) y para `P2` el tiempo de ejecución es `T(n)=3n+2` (ya que en la tabla anterior se observó que se realizan 17 instruciones con *t4 ms* cuando `n` es 5)

Aunque se conozca el tamaño de los datos de entrada, es imposible para muchos problemas determinar el tiempo de ejecución para cada una de las posibles entradas. Por esta razón se debe trabajar con el tiempo utilizado por el algoritmo en el peor de los casos. Con este antecedente se redefine `T(n)`.

`T(n)` = *Tiempo que se demora el algoritmo, para encontrar una solución a un problema de tamaño n.*

8.2. Complejidad

La idea detrás del concepto de complejidad es tratar de encontrar una función $f(n)$, fácil de calcular y conocida, que acote el crecimiento de la función de tiempo, para poder decir: "$T(n)$ crece aproximadamente como f" o más exacto "en ningún caso $T(n)$ se comportará peor que f al aumentar el tamaño del problema".

La **complejidad asintótica** consiste en el cálculo de la complejidad temporal de un algoritmo en función del tamaño de las entradas del problema, prescindiendo de factores constantes y suponiendo valores de `n` muy grandes. Esta complejidad no sirve para establecer el tiempo exacto de ejecución, sino que permite especificar una cota (inferior, superior o ambas) para el tiempo de ejecución del algoritmo.

La eficiencia de un algoritmo se mide mediante el número de operaciones elementales que se deben ejecutar.

8.2.1. Notación asintótica

Se le da el nombre de **notación asintótica** porque aborda el comportamiento de funciones para valores de entrada muy muy grandes. Para la notación asintótica, los valores de entrada pequeños no son interesantes de analizar. La notación **O Grande** establece una cota para el peor caso (el máximo costo de aplicar un algoritmo a un problema de tamaño `n`), es decir, asegura que conociendo dicha cota, ningun otro tiempo empleado en resolver el problema, será superior al de la cota.

La notación asintótica clasifica las funciones de tiempo de los algoritmos (ver fig. 8.2) para que puedan ser comparadas. A modo de ejemplo se pueden mencionar algunas funciones típicas de complejidad de algoritmos (dicho de otra forma que acotan superiormente el comportamiento del tiempo de ejecución). La línea de color rojo, que se encuentra casi pegada al *eje x*, representa a la función $f(n) = log(n)$. La línea naranja

(que sigue a la roja), representa a la función $f(n) = n$. Por su parte, la línea azul marino (posterior a la naranja), representa la función $f(n) = nlog(n)$. La función $f(n) = n^2$ está representada por la línea verde. Finalmente, la línea que se encuentra casi pegada al eje y de color azul turquesa representa la función $f(n) = 2^n$. El *eje x* representa los valores que toma `n`, y el *eje y* `f(n)`.

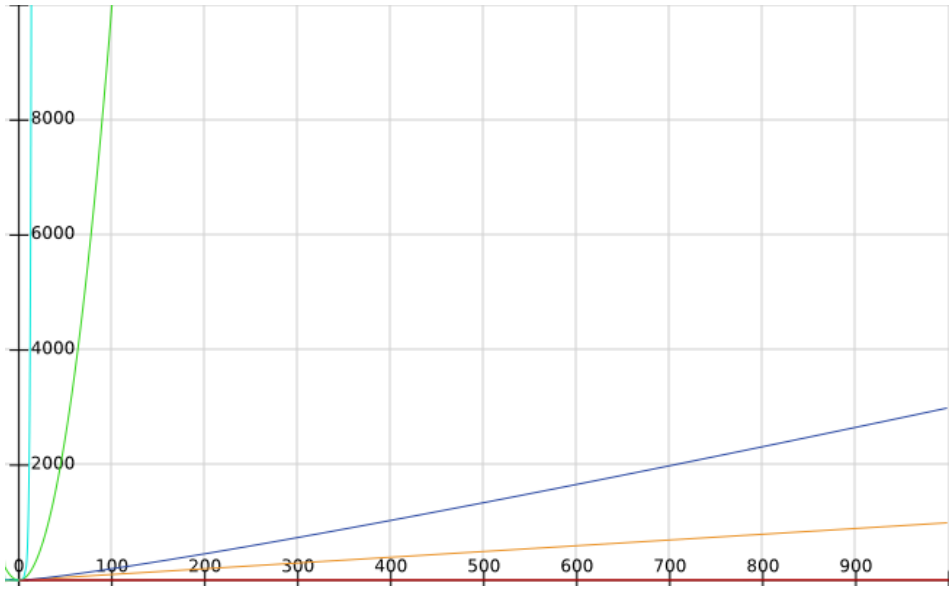


Figura 8.2: Funciones de tiempo de los algoritmos

Esta representación gráfica de las funciones da la pauta para saber cómo "transcurre el tiempo" conforme el valor de `n` aumenta. Evidentemente un algoritmo que implemente una solución exponencial no es adecuado para valores de `n` grandes.

Un problema se denomina **Tratable** si existe un algoritmo de complejidad polinomial para resolverlo. En caso contrario se denomina **Intratable**.

Esta clasificación es importante porque, cuando el tamaño del problema aumenta, los algoritmos de complejidad polinomial dejan de ser utilizables de manera gradual.

8.2.2. Reglas útiles

Consideremos las siguientes reglas aplicadas a la notación de complejidad **O Grande**:

- Regla de la suma:

  $$O(f) + O(g) = O(f + g) = O(max(f, g))$$

- Regla del producto:

  $$O(f)O(g) = O(fg)$$

Ejemplos regla de la suma:

```
def f(n):
    for i in range(n):
        print(i)

    for i in range(n):
        print(i)
```

Programa 8.2: Ejemplo regla de la suma

En el programa 8.2 podemos ver que existe una función f que recibe como argumento a n. Dentro de esta función hay dos ciclos independientes, el primero va de 0 hasta n-1 y el segundo ciclo también. La complejidad tanto del primer ciclo como del segundo es $O(n)$. Pero al estar independientes ambos ciclos podemos utilizar la regla de la suma, es decir:

$$O(n) + O(n) = O(n + n) = O(max(n, n)) = O(n)$$

Podemos decir entonces que la complejidad del código 8.2 es $O(n)$, la cual es lineal. Consideremos el siguiente fragmento de código:

```
def f(n):
    for i in range(n):
        print(i)

    for i in range(n*n):
        print(i)
```

Programa 8.3: Ejemplo regla de la suma 2

Podemos ver que la función f que recibe como argumento a n. En esta función existen dos ciclos independientes, el primero va desde 0 hasta $n-1$, el segundo va desde 0 hasta $n^2 - 1$. Aplicando la regla de la suma tenemos que:

$$O(n) + O(n * n) = O(n + n^2) = O(max(n, n^2)) = O(n^2)$$

Podemos decir entonces que la complejidad del código 7.3, es $O(n^2)$, la cual es polinomial.

Ejemplo regla del producto

Considera el siguiente fragmento de código:

```
def f(n):
    for i in range(n):
        for j in range(n):
            print(i,j)
```

Programa 8.4: Ejemplo regla del producto

La función f recibe como argumento un valor n. Dentro de esta función hay dos ciclos, de los cuales uno está anidado (es decir, el ciclo interno). Ambos van desde 0 hasta $n-1$; sin embargo, el ciclo interno repite este proceso n veces.

Aplicando la regla de la multiplicación tenemos que:

$$O(n) * O(n) = O(n * n) = O(n^2)$$

Concluimos entonces que la complejidad de este programa es $O(n^2)$, es decir, tiene una complejidad polinomial.

8.2.3. Complejidad exponencial y conclusiones del capítulo

Considera el siguiente fragmento de código:

```
1 def fibonacci(n):
2     if n==0 or n==1:
3         return 1
4 
5     return fibonacci(n-1)+fibonacci(n-2)
```

Programa 8.5: Ejemplo complejidad exponencial

En este ejemplo tenemos la función *fibonacci*, la cual recibe como argumento a n. Esta función realiza dos llamadas a sí misma, es decir, emplea una recursividad múltiple. Es importante hacer notar que cada llamada genera dos llamadas más a la función *fibonacci*, por lo que la complejidad de este código es $O(2^n)$. Si, por ejemplo, hubiera tres llamadas recursivas, la complejidad sería $O(3^n)$, pero, bueno, éste no es el caso. Estamos hablando entonces de una complejidad exponencial, y este tipo de algoritmos son los que debemos evitar para trabajar con valores de n grandes.

Retomando a los programas `P1` y `P2` del inicio del capítulo, podemos decir que `P1` tiene una complejidad constante, esto quiere decir que independientemente del tamaño de n únicamente se ejecutarán a lo más un par de instrucciones. En el caso de `P2`, podemos decir que tiene una complejidad lineal, ya que el ciclo que se encuentra en dicho programa se ejecuta n veces.

En conclusión, `P1` es más eficiente que `P2`, ya que una complejidad constante es más rápida que una complejidad lineal, principalmente para valores de n muy grandes.

8.3. Bonus

Siempre que sea posible, se debe buscar la solución más rápida para resolver un problema. Para este fin, te puedes guiar por la complejidad de los algoritmos.

Veamos cómo podemos pasar de una complejidad lineal a una complejidad constante mediante el siguiente ejercicio, el cual consiste en sumar los primeros `n` números pares, empezando del 2.

Como primera aproximación para resolver este ejercicio se nos puede ocurrir una implementación como la siguiente:

```
def suma_pares(n):
    con=2
    c=1
    suma=0

    while c<=n:
        suma+=con
        con+=2
        c+=1

    return suma

n=100
print(suma_pares(n))
```

Programa 8.6: Suma pares 1

Este algoritmo tiene una complejidad lineal, la cual convertiremos en una complejidad constante.

Partamos que la suma de los primeros `n` números naturales está definida por la fórmula bien conocida $(n * (n + 1))/2$.

Tomemos un valor de `n` pequeño, por ejemplo 3. Entonces la suma de los pares sería de:

$$2 + 4 + 6 = 12$$

También podemos ver la expresión anterior como:

$$2(1 + 2 + 3) \Rightarrow 2(n(n + 1)/2) = n(n + 1)$$

Podemos decir que la suma de los pares está definida por la fórmula:

$$n(n + 1)$$

Entonces, la implementación queda de la siguiente manera:

```
def suma_pares_mejorado(n):
    return n*(n+1)

n=100
print(suma_pares(n))
```

Programa 8.7: Suma pares 2

8.4. Conclusiones

La idea de este capítulo es introducir al lector para que dé importancia al diseño eficiente de algoritmos. Como bien sabemos, un problema puede ser resuelto de muchas formas, sin embargo hay formas más eficientes que otras.

8.5. Ejercicios

1. Implementa una solución con complejidad constante para calcular la suma de `n` números impares, partiendo de 1.

2. Implementa una solución con complejidad constante para calcular la suma de los primeros `n` múltiplos de 4, partiendo de 4.

3. Investigar la complejidad del algoritmo de búsqueda binaria e implementarlo.

Capítulo 9

Ejercicios de programación competitiva

9.1. Introducción

La palabra competitiva se lleva a un concepto de concurso, entonces, la programación competitiva es un deporte mental en donde los participantes deben de resolver la mayor cantidad de problemas con ciertas especificaciones en un lapso determinado. Ganará aquel equipo que resuelva más problemas en menos tiempo. También se evalúa, basándose en límites, el uso de la memoria y el tiempo de ejecución de un programa.

¿Para qué me sirve la programación competitiva? Esta pregunta continúa siendo un debate abierto, pero entre las principales ventajas de realizar programación competitiva están:

- Amplio conocimiento de algoritmos y estructuras de datos, esencial para aplicar a puestos de trabajo en Google, Facebook, Amazon o cualquier empresa grande de tecnología.
- Obtener conocimiento avanzado para nuestro ámbito con respecto a la resolución y abstracción de problemas lo cual puede ser empleado en áreas como inteligencia artificial, modelado matemático y simulación.
- Obtener la habilidad de poder trabajar bajo presión y con límites de tiempo.
- Generar redes de contactos para poder colaborar en proyectos futuros.
- Vivir experiencias y viajes únicos por el país y por el mundo.
- Disfrutar resolviendo problemas.

9.1.1. Anatomía de un problema

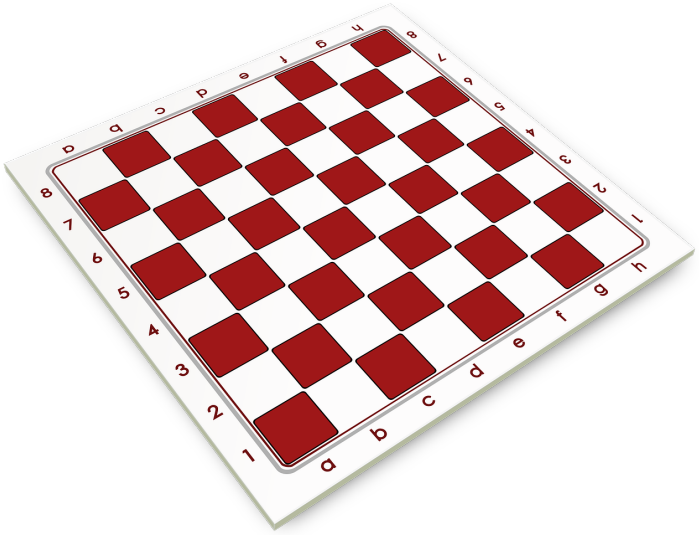
Un problema de un concurso de programación contiene normalmente los siguientes elementos(Halim y Halim, 2019)
Enunciado del problema. Los problemas más sencillos se suelen escribir de forma que engañen a los concursantes, aparentando ser difíciles, mediante la inclusión, por ejemplo, de "información extra" para crear una distracción. Los concursantes deberán ser capaces de filtrar los detalles sin importancia y centrarse en los esenciales.

Descripción de la entrada y la salida. En esta sección se proporcionan los detalles del formato que tendrán los datos de entrada y del que deberán tener los de salida.

Ejemplos de entrada y salida. Los autores de los problemas suelen proporcionar casos de prueba triviales a los concursantes. Los ejemplos de entrada y salida sirven para verificar que los concursantes han entendido el problema en lo más básico y para comprobar si el código es, al menos, capaz de procesar un caso mínimo, y proporcionar la salida correcta en el formato indicado.

Pistas o notas al pie. En algunos casos, los autores del problema pueden incluir pistas o notas al pie, para facilitar la comprensión del problema.

9.2. Tablero de ajedrez



Los tableros de ajedrez tradicionales son de 8 filas (cada fila tiene asignada una letra, de la `a` a la `h`) y 8 columnas (cada columna tiene asignado un número, empezando con el 1), en total 64 casillas que se alternan entre dos colores (regularmente, blanco y negro

aunque pueden variar los colores). A tráves de una columna (letra) y una fila (número) se identifica una casilla, por ejemplo, la casilla a1 es la primer casilla del tablero y además es de color negro.

Tu labor será construir un programa que dada la letra (de la `a` a la `h`) y un número (del 1 al 8), indique que color es la casilla.

Entrada
una letra y un número

Salida
el color de la casilla

Ejemplo entrada
a 1

Ejemplo salida
NEGRO

Solución
Si observamos el tablero notamos que cuando la fila es par (aunque estan identificadas por letras, supongamos que a la `a` se le asigna el número 1, a la `b` el 2 y así para las demás letras) y la columna es par el color de la casilla es negro. Si la fila es par y la columna es impar, la casilla es blanca. Si la fila es impar y la columna par tambien es blanco y si la fila y columna son impares es negro.

A partir del análisis realizado, una posible solución para este reto consiste en crear un diccionario que tenga como claves las filas del tablero (es decir, letras de la `a` a la `h`) y los valores de estas claves sean números (del 1 al 8). La idea del diccionario es trabajar con números en lugar de letras.

Creado el diccionario y leidos los datos de la casilla, mediante un **if-elif-else** se procede a evaluar los casos mencionados en el análisis.

```
1  letras={'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7, 'h':8}
2
3  datos=input().split()
4
5  l=datos[0]
6  d=int(datos[1])
7
8  if letras[l]%2==0 and d%2==0:
9      print("NEGRO")
10 elif letras[l]%2==0 and d%2==1:
11     print("BLANCO")
12 elif letras[l]%2==1 and d%2==0:
13     print("BLANCO")
14 elif letras[l]%2==1 and d%2==1:
15     print("NEGRO")
```

Programa 9.1: Tablero de ajedrez

9.3. Escribir al revés

Pedirás una cadena de caracteres y la desplegarás al revés.

Entrada
cadena

Salida
cadena desplegada al revés

Ejemplo entrada
Pedro

Ejemplo salida
ordeP

Solución

```
cadena=input()

cad=cadena[-1::-1]
print(cad)
```

Programa 9.2: Escribir al revés

9.4. Ordena a los alumnos

La maestra Lucía esta diseñando una lista de asistencia un tanto diferente a la tradicional (que se ordena alfabeticamente). Este nuevo diseño consiste en ordenar por edades de manera descendente, es decir, el alumno de mayor edad irá al inicio y el de menor edad irá al final.

Entrada
Primero hay que pedir la cantidad de alumnos y después sus edades

Salida
Mostrar las edades ordenadas del mayor al menor

Ejemplo entrada
4
11 13 9 10

Ejemplo salida
13 11 10 9

Solución

Se leen las edades de los alumnos, posteriormente mediante un `for`, estas edades se guardan como enteros en una lista llamada `aux`. En seguida, dicha lista invoca al método `sort` (para ordenar los elementos de la lista) con el argumento `reverse=True`, para ordenar los elementos de mayor a menor. Finalmente, en un último `for` se imprimen los valores (ya ordenados de mayor a menor) de la lista `aux`.

```
cantidad_alumnos=int(input())
edades=input().split()

aux=[]
for edad in edades:
    aux.append(int(edad))

aux.sort(reverse=True)

for i in aux:
    print(i,end=" ")
```

Programa 9.3: Ordena a los alumnos

9.5. 888

"Directo al grano, tu trabajo es encontrar el $N-esimo$ número cuyo cubo termina en los dígitos 888."(Manuel, 2017)

Entrada

Primero una línea con un número T la cantidad de casos de prueba. Las siguientes T líneas contienen un entero N.

Salida

Para cada línea de entrada, una línea con el $N-esimo$ número cuyo cubo termina en 888.

Ejemplo entrada

1
1

Ejemplo salida

192

Límites

$1 \leq N \leq 10^6$

Solución

En primera instancia podemos pensar en hacer una búsqueda exhaustiva para cada caso de prueba, desde 1 hasta el $N-esimo$ número, el cual su cubo tiene una terminación en 888. Sin embargo, dados los límites de N en este ejercicio esta solución propuesta no es la adecuada.

Mediante una hoja de cálculo se puede ir visualizando aquellos números que cumplen con el requisito propuesto en el ejercicio. Posteriormente, analizar los patrones que hay entre números elevados al cubo con terminación 888. A partir de este análisis se puede llegar a un patrón o fórmula. En particular una fórmula encontrada para resolver este ejercicio está definida de la siguiente manera:

```
formula=192+250*(N-1)
```

Se observó que cada 250 números a partir de 192 existe un número cuyo cubo termina en 888.

Entonces el programa 9.5 resuelve el ejercicio de la siguiente forma:

Se lee el número T de casos de prueba, seguidamente en un `while` que va desde 1 hasta T se lee el valor de N, el cual se procesa en la fórmula y se guarda en una lista. Al terminar el proceso en el `while`, en un `for` se van imprimiendo los valores que hay almacenados en la lista.

```
1  T=int(input())
2
3  con=1
4  lista=[]
5  while con<=T:
6      N=int(input())
7      formula=192+250*(N-1)
8      lista.append(formula)
9      con+=1
10
11 for i in lista:
12     print(i)
```

Programa 9.4: 888

9.6. Edades de los alumnos

Ahora la maestra Lucía quiere saber cuántos alumnos hay con la misma edad. Ayuda a la maestra en esta ardua labor.

Entrada:

La primera línea contendrá un número n número de alumnos ($1 \leq n \leq 50$). Las siguiente n entradas serán números enteros $a_1, a_2, a_3, ..., a_n (1 \leq a_i \leq 40)$, las edades de

los alumnos.

Salida
Imprime en orden ascendente por la edad y después la cantidad de alumnos correspondiente.

Ejemplo entrada
6
12 12 8 6 9 8

Ejemplo salida
6 1
8 2
9 1
12 2

Solución

En este ejercicio se utiliza una lista y un diccionario para poder darle solución. Primero se leen las edades de los alumnos. A continuación, se crea una lista y un diccionario vacíos. Posteriormente, mediante un `for` se van analizando las edades una por una, si una edad ya se encuentra en la lista `claves` entonces el `diccionario` (las claves en el diccionario son las edades y los valores son las apariciones de cada edad) se incrementa en una unidad en la posición que tenga como clave dicha edad; en caso contrario, la edad se agrega como clave al `diccionario` y se inicializa en 1, también se agrega dicha edad en la lista `claves`. En seguida, se ordena la lista `claves` mediante el método `sort` y finalmente, mediante un `for` se imprime la información contenida en el `diccionario` (es decir, `edad:apariciones`) ordenadamente.

```
n=int(input())
lista=input().split()

claves=[]
diccionario={}

for i in lista:
    valor=int(i)
    if valor in claves:
        diccionario[valor]=diccionario[valor]+1
    else:
        diccionario[valor]=1
        claves.append(valor)

claves.sort()

for i in claves:
```

```
18      print(f"{i} {diccionario[i]}")
```

Programa 9.5: Edades

9.7. Cuenta letras de la cadena

Escribe un programa que lea una cadena. Y el programa nos dirá cuántas veces aparece cada letra del alfabeto.

Entrada
Una cadena de caracteres del alfabeto en minúsculas, si acentos, sin espacios, sin la "ñ"

Salida
La salida imprimirá, en la primera línea, una lista de números que indican el número de veces que aparecen todas y cada una la letra del alfabeto (por orden alfabético). Después se imprimirá una línea por cada una de las letras que sí aparecen en la cadena (también por orden alfabético), de la siguiente forma: La letra x aparece n veces

Ejemplo entrada
Caso 1
universidad

Ejemplo salida
Caso 1
1 0 0 2 1 0 0 0 2 0 0 0 0 1 0 0 0 1 1 0 1 1 0 0 0 0
La letra a aparece 1 veces
La letra d aparece 2 veces
La letra e aparece 1 veces
La letra i aparece 2 veces
La letra n aparece 1 veces
La letra r aparece 1 veces
La letra s aparece 1 veces
La letra u aparece 1 veces
La letra v aparece 1 veces

Solución

Para resolver este ejercicio se necesita hacer uso de un diccionario, el cual tendrá como clave las letras del abecedario y como valor inicial 0. Posteriormente, se procede a leer la cadena que se quiere analizar.

En un primer `for` (que va desde el primer hasta el último carácter de la cadena leída), se va contando mediante el método `count` las veces que aparece cada carácter en la cadena; a su vez, este conteo se almacena en el diccionario en su correspondiente clave. En seguida, en otro `for` se imprimen los valores del diccionario, que son las veces

que apareció cada letra del abecedario en la cadena leída. Finalmente, en un último `for` se van imprimiendo solo las letras del abecedario que tuvieron por lo menos una ocurrencia en la cadena introducida al inicio.

```
diccionario={'a':0, 'b':0, 'c':0, 'd':0, 'e':0, 'f':0,
             'g':0, 'h':0, 'i':0, 'j':0, 'k':0, 'l':0,
             'm':0, 'n':0, 'o':0, 'p':0, 'q':0, 'r':0,
             's':0, 't':0, 'u':0, 'v':0, 'w':0, 'x':0,
             'y':0, 'z':0}

cadena=input()

for i in cadena:
    diccionario[i]=cadena.count(i)

for i in diccionario:
    print(diccionario[i],end=" ")

print()

for i in diccionario:
    if diccionario[i]!=0:
        print(f"La letra {i} aparece {diccionario[i]} veces")
```

Programa 9.6: Cuenta letras vector

9.8. Un anagrama sencillo

> Un anagrama es una palabra o frase que resulta de la transposición de letras de otra palabra o frase. Dicho de otra forma, una palabra es anagrama de otra si las dos tienen las mismas letras, con el mismo número de apariciones, pero en un orden diferente.
> Escribe un programa que lea dos palabras en minúsculas y que determine si las palabras tienen la misma longitud, si no tuvieran la misma longitud, cancelar el programa, de lo contrario, determinar si las palabras son anagrama una de otra.
>
> (Rochin, 2020)

Entrada
La primera línea de entrada contendrá la primera palabra (solo minúsculas, del alfabeto inglés (sin ñ) y a lo mucho contendrá 1000 letras). La segunda línea contendrá la segunda palabra (solo minúsculas, del alfabeto inglés (sin ñ) y a lo mucho contendrá 1000 letras).

Ejemplos entrada
Caso 1
imperdonablemente

imponderablemente

Caso 2
bulliciosamente
escabullimiento

Caso 3
roca
coro

Caso 4
enfriamiento
refinamientos

Ejemplos salida
Caso 1
SI es un ANAGRAMA!

Caso 2
SI es un ANAGRAMA!

Caso 3
NO es un Anagrama!

Caso 4
Las cadenas tienen longitud diferente.
Operacion Cancelada!

Solución

Se leen las dos cadenas por analizar y se compara la longitud de ambas (función `len`), si las cadenas no tienen la misma longitud, entonces se manda el mensaje "Operación cancelada". De lo contrario, si las cadenas tienen la misma longitud, se procede a analizarlas. Primero se crea una variable llamada `ban` (bandera) con el valor de 0. Posteriormente, mediante un `for` que recorre cada carácter de la primera cadena (`cadena1`) se va contando mediante el método `count`, el número de veces que aparece el carácter en ambas cadenas. Si el número de apariciones del carácter es diferente en ambas cadenas entonces, se establece `ban=1` y se sale del ciclo con el `break`. Finalmente, fuera del `for` se valida si la variable `ban` es igual a 0, lo que significa que se recorrieron todos los caracteres de la cadena sin ningún problema, y por lo tanto sí es un anagrama; por el contrario, si la variable `ban` es igual a 1, entonces se concluye que no es un anagrama.

```
1 cadena1=input()
2 cadena2=input()
3
```

```
4  if len(cadena1)!=len(cadena2):
5      print("Las cadenas tienen longitud diferente. \nOperacion
       Cancelada!")
6  else:
7      ban=0
8      for caracter in cadena1:
9          a=cadena2.count(caracter)
10         b=cadena1.count(caracter)
11         if a!=b:
12             ban=1
13             break
14
15     if ban==0:
16         print("SI es un ANAGRAMA!")
17     else:
18         print("NO es un Anagrama!")
```

Programa 9.7: Anagrama sencillo

9.9. Conclusiones

En este capítulo se presentaron diversos retos de programación competitiva, la cual es un deporte mental, con la finalidad de que el lector aplique los conocimientos obtenidos de esta obra.

9.10. Problemas propuestos

9.10.1. Cuántas mayúsculas y minúsculas

Crear un programa que te diga cuántas letras mayúsculas y cuántas minúsculas contiene una palabra.

Entrada
Una palabra.

Salida
En una primer línea el número de caracteres mayúsculas. En otra línea el número de caracteres minúsculas.

Ejemplos entrada
Caso 1
Juan

Caso 2
CucHillo

Ejemplos salida
Caso 1
1
3

Caso 2
2
6

Límites
Podrías limitar las palabras a 30 caracteres

9.10.2. Crucigrama

Escribe un programa que genere un crucigrama básico. Al programa le das 2 palabras: palabra A y palabra B. La palabra A la debe escribir horizontalmente, y la palabra B la debe escribir verticalmente; de tal manera que las palabras se puedan cruzar; esto es deben compartir exactamente una letra. La letra compartida debe ser la primera letra en A que aparece en B. Por ejemplo: dadas las palabras A = "ABBA" y B = "CCBBD", el programa debe generar, en este caso, las 5 líneas que se muestran en el ejemplo mas abajo.

Entrada
La primera y única línea de entrada contiene dos palabras: A y B; de no mas de 30 letras cada una, separadas por un solo espacio. Ambas palabras están escritas con letras mayúsculas del alfabeto español. En todos los casos habrá, al menos, una letra común en ambas palabras.

Salida
Si la longitud de la palabra A es N, y la longitud de la palabra B es M. La salida deben ser M líneas con N caracteres cada linea. La figura que se forme con la salida debe contener a las 2 palabras cruzadas, como se describe en la explicación del problema (por eso se llama crucigrama). El resto de los caracteres, en cada línea, deben ser puntos (el carácter punto).

Ejemplo entrada
ABBA CCBBD

Ejemplo salida
. C . .
. C . .
ABBA

. B . .
. D . .

Límites
$1 \leq longituddeA \leq 30$
$1 \leq longituddeB \leq 30$

9.10.3. Encuentra el tesoro

Pyke es un pirata muy desconfiado y posee mucho dinero, por lo cual el decidió crear varias matrices de 3*3 en diferentes islas,el problema es que no recuerda en cual matriz lo guardo, ayuda lo a ver si en la isla que esta es la correcta y encuentre su tesoro.

Entrada
Siempre sera una solo matriz de 3*3, debes revisar si su diagonal principal y secundaria son iguales.

Salida Si esto es verdad imprimir "Tesoro encontrado" y si es falso imprimir "Sigue buscando".

Ejemplos entrada
Caso 1
1 2 3
2 4 5
6 7 8

Caso 2
1 1 1
1 1 1
1 1 1

Caso 3
3 2 3
7 3 9
3 6 3

Ejemplos salida
Caso 1
Sigue buscando

Caso 2
Tesoro encontrado

Caso 3
Tesoro encontrado

Límites
$0 \leq x \leq 1000$

Referencias

Alberto, M., Schwer, I., Cámara, V., y Fumero, Y. (2005). Matemática discreta. *Con aplicaciones a*.

Cervantes, O., Báez, D., Arízaga, A., y Castillo, E. (2017). *Python con aplicaciones a las matemáticas, ingeniería y finanzas* (1.ª ed.). Alfaomega. Descargado de https://books.google.com.mx/books?id=bermvQEACAAJ

Elmer. (2015). *Fundamentos de la teoria de la computacion (isis).* Descargado de https://fundamentosteoriadelacomputacion.blogspot.com/

Espinosa Armenta, R. (2017). *Matemáticas discretas* (2.ª ed.). México: Alfaomega.

Gómez, C. B. (s.f.). *Álgebra booleana. aplicaciones tecnológicas*. Universidad de Caldas.

Grimaldi, R. P., y Mateos, M. L. (1998). *Matemáticas discreta y combinatoria: una introducción con aplicaciones* (n.º 510 G755M 1997.). Addison-Wesley Iberoamericana.

Halim, S., y Halim, F. (2019). *Programación competitiva manual para concursantes del icpc y la ioi* (1.ª ed.). OJBooks.

Hinojosa, Á. P. (2016). *Python paso a paso.* RA-MA. Descargado de https://books.google.com.mx/books?id=Uo6fDwAAQBAJ

Johnsonbaugh, R. (2005). *Matemáticas discretas.* Pearson Educación.

Ledama. (2016). *¿cuáles son las habilidades básicas del pensamiento?* Descargado de https://erikagh18.wixsite.com/ledama-y-asociados/single-post/2016/03/30/twitter-turns-10

Manuel, L. (2017). *888.* Descargado de https://omegaup.com/arena/problem/888/

Matematicas10. (2018). *Ejemplos de conjunto numerable.* Descargado de https://www.matematicas10.net/2018/03/ejemplos-de-conjunto-numerable.html

Peters, M. (2022). *Álgebra y trigonometría.* Reverté.

Peters, T. (2004). *The zen of python.* Descargado de https://peps.python.org/pep-0020/

Pinales, F. J., y Velázquez, C. E. (2014). *Problemario de algoritmos resueltos con diagramas de flujo y pseudocódigo* (1.ª ed.). Departamento Editorial de la Universidad Autónoma de Aguascalientes.

Rees, P. K. (1986). *Álgebra.* Reverté.

Rochin, F. (2020). *Un anagrama sencillo.* Descargado de https://omegaup.com/arena/problem/un_anagrama_sencillo/

Sués, J. (2015). *Los 100 mejores juegos de ingenio* (1.ª ed.). Paidós.

Torres, A. (2015). *Inteligencia lógico-matemática: ¿qué es y cómo podemos mejorar?* Descargado de https://psicologiaymente.com/inteligencia/inteligencia-logico-matematica

Wikipedia. (2019). *Aritmética.* Descargado de https://es.wikipedia.org/wiki/Aritmética

Wikipedia. (2022). *Sudoku.* Descargado de https://es.wikipedia.org/wiki/Sudoku

Índice alfabético

Acerca del autor

Roberto Enrique Alberto Lira. Egresado de la Licenciatura en Computación por parte de la **Universidad Juárez Autónoma de Tabasco**. Cuenta con una Maestría en Ciencias en Computación por parte del **Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional**.

Del año 2010 a la fecha, se ha desempeñado como profesor-investigador en instituciones de educación superior a nivel nacional, entre ellas, su alma máter la Universidad Juárez Autónoma de Tabasco.

Cuenta con publicaciones en áreas de las Ciencias en Computación. Entre ellas se desctaca la publicación del libro titulado "*Mecanismo de Remodelación Semi-Plástica Para Interfaces Colaborativas*". Entre sus áreas de interés se encuentran: Algoritmos y programación, Programación de dispositivos móviles, Programación Competitiva, etc.

Wilfrido Miguel Contreras Sánchez

Secretario de Investigación, Posgrado y Vinculación

Pablo Marín Olán

Director de Difusión, Divulgación Científica y Tecnológica

Francisco Cubas Jiménez

Jefe del Departamento Editorial de Publicaciones No Periódicas