UES - Escuela de Matemática

Apuntes de Clases • Programación con Python

Abril de 2022 Luisantos Bonilla Mejía

El presente documento está formado por mis apuntes de clase de la materia de Programación I, las cuales fueron impartidas en la Escuela de Matemática, de la Universidad de El Salvador.

La primera unidad está formada por el contenido impartido en un minicurso de introducción a la programación, desarrollado en tres semanas en el mes de agosto del 2020.

El resto de unidades están formadas por el contenido impartido en el ciclo I-2020, para las carreras de Licenciatura en Matemática y Licenciatura en Estadística.

Índice

1. Introducción a la programación

1.1 Conceptos básicos

El beneficio que nos da la programación es la de eliminar tareas repetitivas, por ejemplo, calcular el área de 50 triángulos, dadas las longitudes de la base y la altura de dichos triángulos. Tareas similares se volverían demasiadas tediosas (no imposibles) de realizar en determinados lapsos de tiempo, por ello la programación nos da una salida muy fácil para solventar estos problemas.

Hoy en día existen muchos algoritmos de programación para realizar diferentes tareas en especifico, unas mejores que otras de acuerdo a la necesidad del programador. Debido a esto los software los podemos clasificar:

Lenguaje Máquina: Es el que entienden directamente las computadoras y este consiste en programar usando el alfabeto binario, es decir, ceros y unos (0 y 1). Con este alfabeto formamos cadenas de ceros y unos que la máquina entiende usando el microprocesador como nuestras instrucciones.

Dicho lenguaje de programación fue el primero en usarse para programar, sin embargo debido a la facilidad de cometer errores se ha dejado de utilizar.

Lenguajes de programación de bajo nivel: Es el lenguaje de programación que se caracteriza porque influye directamente en el hardware, para poder transmitir las instrucciones a la máquina.

La desventaja de usar este tipo de lenguaje es que depende bastante de los componentes físicos de la máquina, sin embargo su implementación es más fácil para la creación de programas.

El lenguaje ensamblador fue el primer lenguaje de programación que trató de sustituir el lenguaje máquina por otro lenguaje que fuese más parecido al de los seres humanos.

Lenguajes de programación de alto nivel: Son aquellos cuya característica principal, consiste en una estructura sintáctica y semántica legible, acorde a las capacidades cognitivas humanas.

A diferencia de los lenguajes de bajo nivel, son independientes de la arquitectura del hardware, lo que significa que tienen mayor portabilidad.

En la actualidad, por su facilidad de lectura para el ser humano, existen muchos programas de alto nivel, por ello, es necesario clasificarlos. La clasifica-

ción que mostraremos a continuación se ha hecho pensando en la utilidad para el cual fue creado el lenguaje de programación.

1. Lenguajes de programación imperativos: entre ellos tenemos COBOL, PASCAL, C y ADA.
2. Lenguajes de programación declarativos: LISP y PROLOG.
3. Lenguajes de programación orientados a objetos: SMALLTALK y C++.
4. Lenguajes de programación orientados al problema: Son aquellos lenguajes específicos para gestión.
5. Lenguajes de programación naturales: Son los nuevos lenguajes que pretender aproximar el diseño y la construcción de programas al lenguaje de las personas.

Notemos que los lenguajes de programación de bajo nivel son difíciles de aprender; ya que al depender del hardware, requiere que el programa sea reescrito desde el inicio cuando se quiera utilizar en una máquina distinta a la que se utilizó para crear el programa, ya que depende de la codificación de los procesadores de cada máquina. Por otro lado los lenguajes de programación de alto nivel son más fáciles de aprender porque se usan palabras o comandos del lenguaje natural, generalmente del inglés.

Existe otra forma de clasificar los lenguajes de programación de acuerdo al desarrollo de las computadoras, dicha clasificación es la siguiente.

- Lenguajes de programación de primera generación: Lenguaje de máquina y de bajo nivel.
- Lenguajes de programación de segunda generación: Los primeros lenguajes de programación de alto nivel imperativo (FROTRAN, COBOL).
- Lenguajes de programación de tercera generación: Son lenguajes de programación de alto nivel imperativo (ALGOL 8, PL/I, PASCAL, MODULA).
- Lenguajes de programación de cuarta generación: Usados en aplicaciones de gestión y manejo de bases de dados (NATURAL, SQL).
- Lenguajes de programación de quinta generación: Creados para la inteligencia artificial y para el procesamiento de lenguajes naturales (LISP, PROLOG).

Durante el desarrollo de este curso nos ocuparemos de un lenguaje de **alto nivel**, para ser especifico nos concentraremos en aprender el lenguaje de programación llamado Python (Python 2.7 y Python 3). Por otra parte, de manera introductoria tengamos en cuenta el siguiente glosario.

Glosario	
Términos	**Definición**
Lenguaje informático	Es un idioma artificial, utilizado por ordenadores, cuyo fin es transmitir información de algo a alguien. Los lenguajes informáticos, pueden clasificarse en: a) lenguajes de programación (Python, PHP, Pearl, C, etc.); b) lenguajes de especificación (UML); c) lenguajes de consulta (SQL); d) lenguajes de marcas (HTML, XML); e) lenguajes de transformación (XSLT); f) protocolos de comunicaciones (HTTP, FTP); entre otros.
Lenguaje de programación	Es un lenguaje informático, diseñado para expresar órdenes e instrucciones precisas, que deben ser llevadas a cabo por una computadora. El mismo puede utilizarse para crear programas que controlen el comportamiento físico o lógico de un ordenador. Está compuesto por una serie de símbolos, reglas sintácticas y semánticas que definen la estructura del lenguaje.
Lenguajes interpretados	A diferencia de los lenguajes compilados, no requieren de un compilador para ser ejecutados sino de un intérprete. Un intérprete, actúa de manera casi idéntica a un compilador, con la salvedad de que ejecuta el programa directamente, sin necesidad de generar previamente un ejecutable. Ejemplo de lenguajes de programación interpretado son Python, PHP, Ruby, Lisp, entre otros.
Tipado dinámico	Un lenguaje de tipado dinámico es aquel cuyas variables, no requieren ser definidas asignando su tipo de datos, sino que éste, se auto-asigna en tiempo de ejecución, según el valor declarado.
Multiplataforma	Significa que puede ser interpretado en diversos Sistemas Operativos como GNU/Linux, Windows, Mac OS, Solaris, entre otros.
Multiparadigma	Acepta diferentes paradigmas (técnicas) de programación, tales como la orientación a objetos, aspectos, la programación imperativa y funcional.
Código fuente	Es un conjunto de instrucciones y órdenes lógicas, compuestos de algoritmos que se encuentran escritos en un determinado lenguaje de programación, las cuales deben ser interpretadas o compiladas, para permitir la ejecución del programa informático.

Investiga y responde cada una de las siguientes interrogantes:

1. Escribe otras desventajas que poseen los lenguajes de bajo nivel.
2. ¿Cuáles términos se pueden aplicar a Python del glosario anterior?

   Nota: Por lo descrito en la tabla sabemos que Python es un lenguaje informático, interpretado, así que la pregunta está dirigida al resto de términos.
3. Cuál es la definición de **lenguaje de programación compilado**. Además cite algunos ejemplos.

1.2 Representación de datos en las computadoras

La computadora solo entiende dos estados, encendido o apagado, que fácilmente podemos relacionar con las respuesta de sí o no de nuestro lenguaje cotidiano. Más aún, estos dos estados o dos respuestas las podemos representar por 0 y 1, a esto se le conoce como **bit** o **dígito binario** (por el ingles Binary Digit).

Básicamente, las computadoras representan la información mediante bits, de esta forma el lenguaje de la computadora es enteramente binario. El bit es la unidad más pequeña en que la computadora puede almacenar información.

Está claro que con un bit no es suficiente para poder guardar información compleja, por esta razón formamos secuencias de bits, que nos permitirán representar la información. A las secuencias de longitud 8 (cadenas de 8 bits) son llamados **Byte**. A continuación mostramos una tabla de unidades de información.

Unidades de información		
Unidad	**Símbolo**	**Equivalencia**
Bit	b	0 y 1
Byte	B	8 b
Kbyte	KB	1024 B
Mbyte	MB	1024 KB
Gbyte	GB	1024 MB
Tbyte	TB	1024 GB

Observación: Con los Bytes tenemos 256 combinaciones de 0 y 1, las suficientes para representar nuestro alfabeto y caracteres especiales.

Teniendo claro cómo la computadora entiende la información, podemos asociar a cada secuencia de bits un valor, y de esta forma tener un esquema de codificación de información, esto permitirá poder interactuar con la computadora de una manera más natural. Algunos esquemas de codificación son:

1. **ASCII:** El código ASCII (por sus siglas en ingles American Standard Code for Information Interchange) es el código estándar para el intercambio de información (se pronuncia Aski).

2. **EBCDIC:** El código (por sus siglas en ingles EBCDIC Extended Binary Coded Decimal Interchange Code) es un código binario que representa caracteres alfanuméricos, controles y signos de puntuación. Cada carácter está compuesto por 8 bits.

3. **Unicode:** Es un código diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de numerosos idiomas y disciplinas técnicas, además de textos clásicos de lenguas muertas. El término Unicode proviene de los tres objetivos: universalidad, uniformidad y unicidad. Este código puede tratar secuencias de 8, 16 o 32 bits.

Nota: ASCII tiene 255 caracteres y EBCDIC tiene 256 caracteres.

Ejemplo: Tabla del código ASCII.

Caracteres ASCII de control

00	NULL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	HT	(tab horizontal)
10	LF	(nueva línea)
11	VT	(tab vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc.vínculo datos)
17	DC1	(control disp. 1)
18	DC2	(control disp. 2)
19	DC3	(control disp. 3)
20	DC4	(control disp. 4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sínc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin del medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles

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

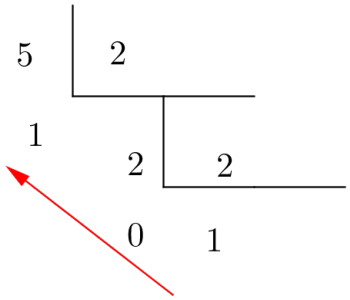
ASCII extendido

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

1.2.1 Representación binaria de los números naturales

Hablando un poco sobre cómo la computador entiende la información, conozcamos de manera mecánica, cómo codificar y decodificar números enteros no negativos.

Primero conozcamos el mecanismo de codificación, para ello veamos cómo codificar 5 en el código binario.



Como vemos tenemos que ir dividiendo entre dos el cociente en cada nuevo paso, luego para saber el código binario del número 5, tomamos el último cociente y tomamos los residuos de forma ascendente como indica la flecha roja. Así tenemos que 5 en código binario es 101.

Ahora para recuperar la información a partir del código binario, hacemos una multiplicación de derecha a izquierda, multiplicando el dígito binario por una potencia de dos, donde el exponente de dos es la posición del dígito binario. Las posiciones comienzan desde cero. Implementemos lo anterior para la secuencia 101.

$$(1 * 2^2) + (0 * 2^1) + (1 * 2^0) = 4 + 0 + 1 = 5$$

Nota: Este proceso es llamado **decodificación**.

Ejercicios:

1. Convertir a código binario los números: 2, 3, 8, 15 y 27
2. Decodificar las siguientes secuencias: 10, 11, 1000, 1111 y 11011

Una pregunta que puede surgir es ¿podemos hacer lo mismo pero con un número diferente de dos?

La respuesta a la pregunta anterior es sí, veamos un caso usando el número entero 16 para realizar la codificación y decodificación.

Codificación:

$$\begin{array}{r|l} 19 & 16 \\ \hline 3 & 1 \end{array}$$

Decodificación:

$$(1 * 16^1) + (3 * 16^0) = 16 + 3 = 19$$

Observación: Notemos que son los mismos pasos, salvo que se ha cambiado el 2 por 16.

Teniendo esto en cuenta, tenemos la siguiente tabla de equivalencia entre un código binario y un código hexadecimal.

$\mathbb{Z}_0^+$	**Binario**	**Hexadecimal**
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	*A*
11	1011	*B*
12	1100	*C*
13	1101	*D*
14	1110	*E*
15	1111	*F*

Utilizando está tabla podemos codificar 27 en sistema hexadecimal.

Codificación:

$$\begin{array}{r|l} 27 & 16 \\ \hline 11 & 1 \end{array}$$

Por lo tanto la codificación de 27 es $1B$, donde 11 se representa por la letra B. Para verificar hacemos una decodificación.

Decodificación:

$$(1 * 16^1) + (B * 16^0) = 16 + 11 = 27$$

1.2.2 Cambiar de un sistema a otro

Para cambiar de un número hexadecimal a un binario y viceversa, lo que hacemos es agrupar en grupos de 4 lo dígitos binarios, de derecha a izquierda, si el último bloque tiene menos de 4 dígitos, estos se llenan con ceros hasta completar los 4. Para entender esto veamos el siguiente ejemplo:

$$11011 = (1)(1011) = (0001)(1011) = 1B$$

1.3 Operaciones básicas con números binarios

1. **Sumar.** Para sumar números binarios lo hacemos de manera usual, pero teniendo en cuenta las siguientes operaciones.

   - $0 + 0 = 0$
   - $1 + 0 = 1$
   - $0 + 1 = 1$
   - $1 + 1 = 0$, y se lleva 1 a la siguiente columna (para sumar).

   Ejemplo:

$$\begin{array}{rcl} 101+ & \Leftrightarrow & 5+ \\ 11 & \Leftrightarrow & 3 \\ \hline 1000 & \Leftrightarrow & 8 \end{array}$$

2. **Restar.** Para restar números binarios lo hacemos de manera usual, pero teniendo en cuenta las siguientes operaciones.

   - $0 - 0 = 0$
   - $1 - 0 = 1$
   - $1 - 1 = 0$
   - $0 - 1 = 1$, y se lleva 1 a la siguiente columna (para restar).

Ejemplo:

$$\begin{array}{rcl} 101- & \Leftrightarrow & 5- \\ 11 & \Leftrightarrow & 3 \\ \hline 010 & \Leftrightarrow & 2 \end{array}$$

3. **Multiplicar.** Para multiplicar números binarios lo hacemos de manera usual, pero usando las reglas de la suma binaria.

   Ejemplo:

$$\begin{array}{rcl} 101 \times 11 & \Leftrightarrow & 5 \times 3 \\ 101 & & 15 \\ 101 & & \\ \hline 1111 & \Leftrightarrow & 15 \end{array}$$

4. **Dividir.** Para dividir números binarios lo hacemos de manera usual, pero usando las operaciones anteriores, y teniendo en cuenta que solo podemos colocar en el cociente ceros y unos.

   Ejemplo:

$$\begin{array}{r|l} 101 & 11 \\ \hline 11 & 001 \\ \hline 010 & \end{array} \Leftrightarrow \begin{array}{r|l} 5 & 3 \\ \hline 3 & 1 \\ \hline 2 & \end{array}$$

Ejercicios:

1. Realizar las siguientes operaciones:
   - $111 + 11101 + 10110$
   - $11011 - 1101$
   - 11011×1101

- $11111 \div 11$

2. Investigar las reglas para realizas operaciones sobre los números reales codificados en el sistema binario.

3. Investigar la manera de hacer las mismas operaciones en el sistema hexadecimal.

1.4 Pseudocódigo

Un **pseudocódigo**[1] es la manera informal de presentar el principio o idea operativa de un programa. Hay muchas maneras de presentar un pseudocódigo, la más fácil es mediante el uso de palabras claves de los lenguajes de programación, estos son: **imprimir**, **si**, **sino**, **para** y **mientras**.

Nota: La idea del pseudocódigo es poder presentar los pasos que un programa debe realizar, pero de tal manera que un ser humano lo entienda, sin necesidad que conozca o no un lenguaje de programación.

En términos generales, un pseudocódigo tiene la siguiente estructura:

```
Proceso Nombre_del_Programa
    acción 1
    acción 2
    ...
    acción n
FinProceso
```

Observación: La forma de escribir el pseudocódigo debe ser lo más apegado al lenguaje de programación que se desea implementar o enseñar.

Una de las ventajas de usar un pseudocódigo es que será más fácil comprender las acciones principales del programa, y por lo tanto, será más fácil su implementación. Sin embargo, un pseudocódigo no podrá ser entendido por una computadora, ya que en general, no estará escrito en el lenguaje de programación que deseemos utilizar.

Veamos un ejemplo de un pseudocódigo para el lenguaje de programación **Python**.

Ejemplo: El siguiente pseudocódigo representa un programa que puede decidir cuándo un entero positivo n es par o impar.

[1] Revisando las partes que conforman la palabra **pseudocódigo** podemos definirlo como *falso lenguaje*.

```
Proceso Numero_par
    n #un numero entero positivo.
    si el residuo de n al ser dividido por 2 es igual a cero:
        imprimir n es par
    sino:
        imprimir n es impar
FinProceso
```

Observación: En **Python** con # se puede introducir comentarios, para que el usuario del código pueda entender mejor ciertas instrucciones. La instrucción `imprimir` indica que en pantalla el usuario verá `n es par` o `n es impar`.

Ejercicios:

1. Escribir un pseudocódigo de un programa que:
   - Verifique si un número entero no negativo sea múltiplo de 5.
   - Realice la suma de dos enteros positivos, si ambos son múltiplos de 3.
   - Dibuje en pantalla la siguiente figura:

```
********
* Hola *
* Mundo*
********
```

Nota: En los lenguajes de programación, si escribimos varios imprimir, cada imprimir escribirá en una nueva línea en la pantalla. Por ejemplo:

```
imprimir Hola
imprimir Un gusto

>>> En pantalla veremos:
Hola
Un gusto
```

1.5 Variables en Python

Las **variables** son contenedores de información. La información puede ser del tipo numérico, cadena de texto o de cualquier otro tipo que **Python** pueda manejar. Al ser contenedores, requiere un edificador (nombre) para poder acceder a la

información que guardan. La sintaxis[2] es: `mi_variable = valor`. El nombres de las variables válidos en **Python** debe cumplir las siguientes tres reglas:

1. Son secuencias arbitrariamente largas de letras y dígitos.
2. La secuencia debe empezar con una letra.
3. El guión bajo (_) es considerado una letra.

Para que **Python** presente en pantalla la información de una variable necesitaremos la instrucción `print()`. Además, podemos saber el tipo de valor que guarda la variable usando el comando `type()`.

Ejemplos:

```
1  a = 5
2  print(a)
3  print(type(a))
```

```
5
<class 'int'>
```

```
1  b = 5.5
2  print(b)
3  print(type(b))
```

```
5.5
<class 'float'>
```

```
1  c = "Hola Mundo"
2  print(c)
3  print(type(c))
```

```
Hola Mundo
<class 'str'>
```

```
1  d = [1,1.1,"Hola",["xD"]]
2  print(d)
3  print(type(d))
```

```
[1, 1.1, 'Hola', ['xD']]
<class 'list'>
```

Observación: En los ejemplos anteriores, se han presentado los cuatro tipos básicos de información que puede contener una variable (entero, real, cadena de texto y lista).

[2]La sintaxis que se presenta, crea la variable al mismo tiempo que le asigna el valor, en dicha asignación queda determinado su tipo.

Además en **Python** podemos cambiar la información de la variable una vez creada, esto se hace reasignando el valor, veamos un ejemplos de esto.

Ejemplo:

```
1  b = 5.5
2  print(b)
3  print(type(b))
```

```
5.5
<class 'float'>
```

```
1  b = "Hola de nuevo"
2  print(b)
3  print(type(b))
```

```
Hola de nuevo
<class 'str'>
```

Observación: Notemos que basta con usar la sintaxis: `mi_variable = valor`, para cambiar el valor de una variable o para crearla.

Veamos algunas características básicas de las listas en **Python**.

1.5.1 Listas en Python

Lista: es un tipo de dato que almacena enteros (`int`), reales (`float`) y cadenas de texto (`str`), internamente cada posición puede ser un tipo de datos distinto. También dentro de pueden almacenar listas (`list`).

Ejemplo:

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(mi_lista)
```

```
["cadena de texto", 15, 2.8, "otro dato", 25]
```

Las listas en **Python** son:

- **Heterogéneas:** pueden estar conformadas por elementos de distintos tipo, incluidos otras listas.
- **Mutables:** sus elementos pueden modificarse.

Las listas pueden pensarse como una secuencia de datos ordenados por su posición dentro de la lista; esto quiere decir que cada elemento dentro de la lista tiene una posición, estas posiciones comienzan a enumerarse de izquierda a derecha comenzando desde cero y termina una unidad menos a la cantidad de

elementos que posea la lista (en programas como **Matlab** u **Octave** la numeración comienza desde uno hasta el número total de elementos que posea la lista).

A la **posición** de cada elemento se les llamará **índice**, siendo el índice cero (0) el perteneciente al primer elemento.

Ejemplo:

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(mi_lista[0])
3  print(mi_lista[3])
4  print(mi_lista[4])
```

```
» cadena de texto
otro dato
25
```

En los ejemplos anteriores, podemos observar que 4 es el índice del último elemento en la lista; es así, ya que la lista tiene 5 elementos, y la numeración en este caso seria desde 0 hasta 4 (desde cero hasta una unidad menos a la cantidad de elementos que posea la lista).

También, podemos extraer una parte de la lista usando los índices. Para extraer una parte de la lista, es necesario especificar el índice desde el cuál deseamos iniciar la extracción, hasta una unidad más del índice que queremos finalizar la extracción. Veamos a continuación dos ejemplos, el primero extraemos desde el segundo elemento hasta el cuarto elemento, y en el segundo desde el tercer elemento hasta el último (recuerde el último elemento tiene índice 4).

Ejemplo:

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(mi_lista[1:4])
3  print(mi_lista[2:5])
```

```
» [15, 2.8, "otro dato"]
[2.8, "otro dato", 25]
```

Además, a través del índice, podemos cambiar los elementos de una lista en el lugar que especifica dicho índice.

Ejemplos:

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(mi_lista[0])
```

```
» cadena de texto
```

```
mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
mi_lista[0] = 2 #Cambiamos "cadena de texto" por 2.
print(mi_lista[0])
```

```
2
```

```
mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
print(mi_lista[2])

mi_lista[2]= "¡Hola mundo!" #Cambiamos 2.8 por "¡Hola mundo!".
print(mi_lista[2])
```

```
2.8
¡Hola mundo!
```

Como vemos en los ejemplos anteriores, podemos incluso cambiar elementos por otros totalmente diferentes, incluso si son de diferentes tipos (en este caso, cambiamos una cadena de texto por un entero y un real por una cadena de texto).

Por último, podemos agregar nuevos elementos a una lista, lo que significa, hacer más grande la lista (que tenga más elementos), esto lo podemos hacer con la función `append()`, la sintaxis es: `lista.append(elemento)`.

Ejemplo:

```
print(mi_lista)

mi_lista.append("Nuevo Dato")

print(mi_lista)
```

```
[2, 15, "¡Hola mundo!", "otro dato", 25]
[2, 15, "¡Hola mundo!", "otro dato", 25, "Nuevo Dato"]
```

Observación: `append()` agrega el nuevo elemento al final de la lista, además, podemos agregar elementos de cualquier tipo, no solo cadenas de texto.

1.5.2 Cadenas de texto en Python

Las **cadenas de texto** son secuencias de caracteres inmutables[3] encerrado entre comillas. Al ser secuencias, la misma sintaxis de las listas se utiliza para acceder a la información de las cadenas de texto.

Ejemplos:

```
texto = "Hola Mundo"

print(texto[3])
```

[3]Son como listas, solo que no se puede cambiar sus elementos una vez creada.

```
5       print(texto[1:5])
6
7       print(len(texto))
```

```
a
ola
ola M
10
```

Observación: En las cadenas de texto, el espacio es considerado una letra. Además, `len()` es una función que nos permite saber cuántos elementos tiene una lista o una cadena de caracteres.

También podemos agregar más caracteres a una lista, usando el operador +.

Ejemplo:

```
1       texto = "Hola Mundo"
2       nuevo_texto = "Hola otra vez"
3       print(texto + nuevo_texto)
```

```
Hola MundoHola otra vez
```

```
1       texto = "Hola Mundo"
2       nuevo_texto = "Hola otra vez"
3       print(texto + " " + nuevo_texto)
```

```
Hola Mundo Hola otra vez
```

Observación: Al final de `texto` se agrega el contenido de `nuevo_texto`. Podemos agregar un espacio entre cada cadena de texto, colocando una cadena de texto con un espacio o directamente al inicio del contenido de la variable `nuevo_texto`. La cadena de texto formada por la unión de ambas variables puede ser guardada en una nueva variable.

1.5.3 Constantes en Python

En **Python** hay valores que son únicos y que se pueden pensar como constantes, los cuales son:

1. `None`: Este valor significa literalmente **nada**. Es muy útil para crear variables que no queramos utilizar aún.
2. `True`: Este valor significa literalmente **verdadero**. Es de gran utilidad para denotar que cierta condición es verdadera.
3. `False`: Este valor significa literalmente **Falso**. Es de gran utilidad para denotar que cierta condición es falsa.

Observación: Para acceder a esta información basta escribir `None`, `True` o `False`. Por esa razón son constantes, porque tienen identificador (nombre).

1.5.4 Comparando valores en Python

Para comparar valores en **Python** se utiliza el doble igual (==), esta instrucción devuelve `True`, si ambas expresiones son equivalentes o `False`, si no son equivalente.

Ejemplos:

```
1 print(2==2)
```

```
True
```

```
1 print(2==5)
```

```
False
```

Observación: Si queremos saber si dos expresiones **no** son equivalentes, se utiliza la instrucción `!=` (ejemplo: `2!=5`, devuelve `True`).

Nota: De igual forma podemos utilizar <, <=, > y >=, y se entienden tal cual como en matemática.

Ejercicios: Escriba en un archivo **Python** las siguientes instrucciones y concluya sobre los valores que se obtienen en pantalla.

1. `print(2<3)`
2. `print(2<=3)`
3. `print(2>3)`
4. `print(2>=3)`

1.6 Condicionales en Python

Para agregar una condición, usamos la palabra `if` seguido de una proposición lógica y cerrando con dos puntos (`:`). La sintaxis es: `if condicion_logica:`

Luego de escribir la condición, en una nueva línea se debe colocar las instrucciones que se ejecutarán, si se cumple la condición. Estas instrucciones deben ir indentadas (una Sangría) para indicar a **Python** que pertenecen al condicional.

Ejemplo:

```
if n%2==0:
    print("Es par")
```

Observación: La instrucción `n%2` devuelve el residuo que deja *n* al ser dividido por 2. La condición del ejemplo imprimirá en pantalla `Es par`, si el residuo es cero al dividir por 2.

Si queremos que se ejecute cierta instrucción cuando la condición lógica no se cumpla, debemos usar `else:` en una nueva línea.

Ejemplo:

```
if n%2==0:
    print("Es par")
else:
    print("Es impar")
```

Observación: Si la condición no se cumple, automáticamente **Python** se dirige a ejecutar las instrucciones del `else`.

Nota: En **Python** la instrucción `else` no puede ser escrita sin que antes exista la instrucción `if`.

Además, si tenemos varias condiciones que se excluyen una a otra, luego de usar `if` en la primera condición, en las siguientes debe escribirse `elif`.

Ejemplo:

```
if n<3==True:
    print("Es menor")
elif n>3==True:
    print("Es mayor")
```

Observación: También podemos colocar `else` para agregar instrucciones por si las demás condiciones no se cumplen.

1.6.1 Solicitando información al usuario

Para solicitar información al usuario, podemos usar la instrucción: `input()`. La información ingresada se guarda en formato cadena de texto.

Nota: Guardar la información ingresada por el usuario en una variable.

Ejemplo:

```
>>> n = input()
```

Observación: En el ejemplo vemos la sintaxis de cómo pedir información al usuario, y a la vez guardarla en la variable *n*.

Entre los paréntesis de `input()` podemos agregar una cadena de texto, a modo que el usuario entienda que se le está solicitando información.

Ejemplo:

```
1    n = input("Ingrese un entero: ")
```

```
»    Ingrese un entero: _
```

Como toda la información que recolecta `input()` es tomada como cadena de texto[4], nosotros podemos hacer que **Python** mantenga el tipo de dato ingresado, al usar la instrucción `eval()`.

Ejemplo:

```
1    n = eval(input("Ingrese un entero: "))
```

```
»    Ingrese un entero: _
```

Observación: `eval()` permite que cuando se ingrese el número entero, se mantenga como información del tipo entero en **Python**.

2. Ciclos, listas y funciones

2.1 Ciclos `For` y `While`

Ciclo `While`: este ciclo evalúa una condición e ingresa al ciclo mientras ésta sea verdadera. Caso contrario, finaliza las iteraciones y continúa con el resto del programa.

Por ejemplo:

```
1    count = 0
2    while count < 5:
3      print(count, " es menor que 5")
4      count = count + 1
5    print(count, " no es menor que 5")
```

```
»    0  es menor que 5
     1  es menor que 5
     2  es menor que 5
     3  es menor que 5
     4  es menor que 5
```

[4] En **Python 2** la instrucción **input**() mantiene el tipo de información ingresada, y existe la instrucción **raw_input**() que guarda todo como cadena de texto.

```
»   5  no es menor que 5
```

Una característica de los ciclos while es que la acción a repetirse puede hacerse desde 0 hasta un número indefinido de veces. Es posible que la cantidad máxima de iteraciones no pueda conocerse una vez iniciada la ejecución del programa.

Ciclo `For`: este ciclo asigna a una variable contador cada uno de los datos incluidos en una secuencia. A diferencia del ciclo while, no evalúa una condición, este ciclo finaliza cuando la variable contador llega al final de la secuencia.

Por ejemplo:

```
1       for count in [0,1,2,3,4]:
2         print(count, " es menor que 5")
3       print(count, " no es menor que 5")
```

```
»   0  es menor que 5
    1  es menor que 5
    2  es menor que 5
    3  es menor que 5
    4  es menor que 5
    4  no es menor que 5
```

Una característica de los ciclos for es que la cantidad de iteraciones puede ir desde 0 hasta un número determinado. Éste número puede conocerse durante la ejecución del programa y antes de ingresar al ciclo. Es por esta razón que es posible resolver el problema utilizando una secuencia de datos.

Una **secuencia** en python es un tipo de dato que permite almacenar múltiples valores en una misma variable. Esta característica de las secuencias es que la hace posible iterar sobre las mismas, recorriendo uno a uno sus elementos. Una secuencia puede estar vacía o tener un sólo elemento.

Una secuencia con un solo elemento sigue siendo iterable, pero admite una sola iteración. No ha de confundirse con una variable atómica que sólo toma un valor.

2.2 Función `range`

`range([inicio], fin, [salto])`: Esta función permite crear secuencias que contienen progresiones aritméticas. Los argumentos deben ser enteros. La lista inicia exactamente en "inicio" y termina antes de llegar a "fin". La función range genera una secuencia de tipo lista en Python 2 y en Python 3 es una clase del tipo *range*. En todo el curso lo consideraremos en Python 3.

Se consideran las siguientes situaciones:

1. Si *salto* es omitido, asume el valor por defecto de 1.

2. Si el *inicio* es omitido, asume el valor por defecto 0.

3. Si el *salto* es negativo, entonces la lista mantendrá un crecimiento descendente.

4. Si se colocan dos de los tres argumentos, se asume que son *inicio* y *fin*.

Por ejemplo:

```
1       for count in range(10):
2           print(count,end=",")
```

```
0,1,2,3,4,5,6,7,8,9,
```

```
1       for count in range(1,11):
2           print(count,end=",")
```

```
1,2,3,4,5,6,7,8,9,10,
```

```
1       for count in range(0,30,5):
2           print(count,end=",")
```

```
0,5,10,15,20,25,
```

```
1       for count in range(0):
2           print(count,end=",")
```

```

```

```
1       for count in range(0,-8,-1):
2           print(count,end=",")
```

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

```
1       for count in range(0,10,-1):
2           print(count,end=",")
```

```

```

```
1       for count in range(10,2):
2           print(count,end=",")
```

```

```

2.3 Sentencias `pass`, `break` y `continue`, Cláusula `else`

`pass`**:** Esta sentencia hace nada. Se utiliza cuando una sentencia se requiere sintácticamente, pero no funcionalmente.

Por ejemplo:

```
if x>=0:
  pass
else:
  x=-x
```

`continue`**:** Esta sentencia finaliza la iteración del ciclo en la que se encuentra y continúa con la siguiente (si la hubiere).

Por ejemplo:

```
for num in range(2, 10):
  if num % 2 == 0:
    print("número par", num)
    continue
  print("número impar", num)
```

`break`**:** Esta sentencia finaliza abruptamente el ciclo en el que se encuentre. El flujo evita toda instrucción asociada al ciclo roto.

Por ejemplo:

```
from random import randint
oculto = randint(10,100)
while True:
  numero = int(input("\nadvina un número entre 10 y 100 -->"))
  if numero == oculto:
    print("¡Felicidades, adivinaste!")
    break
  print("sigue intentanto, mejor suerte a la próxima")
```

`else`**:** Las estructuras iterativas también pueden contener una cláusula else. Se ejecuta cuando el ciclo termina al agotar la lista (`for`) o cuando la condición se evalúa falsa (`while`).

Por ejemplo: Generar los números primos menores de un dígito.

```
1       for n in range(2, 10):
2         for x in range(2, n):
3           if n % x == 0:
4             break
5         else: # el ciclo finalizó sin recurrir a break
6           print(n, 'es primo')
```

```
2 es primo
3 es primo
5 es primo
7 es primo
```

2.4 Listas

Un tipo de dato especial dentro de **Python**es la **lista**, este tipo de dato puede almacenar colecciones de datos de diversos tipos (incluyendo ellos mismos).

Lista: son variables que almacenan enteros (`int`), reales (`float`) y cadenas de texto (`str`), internamente cada posición puede ser un tipo de datos distinto. También dentro de pueden almacenar listas (`list`).

Ejemplo:

```
1 mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2 print(mi_lista)
```

```
["cadena de texto", 15, 2.8, "otro dato", 25]
```

Las listas en **Python**son:

- **Heterogéneas:** pueden estar conformadas por elementos de distintos tipo, incluidos otras listas.
- **Mutables:** sus elementos pueden modificarse.

Las listas pueden pensarse como una secuencia de datos ordenados por su posición dentro de la lista; esto quiere decir que cada elemento dentro de la lista tiene una posición, estas posiciones comienzan a enumerarse de izquierda a derecha comenzando desde cero y termina una unidad menos a la cantidad de elementos que posea la lista (en programas como **Matlab** u **Octave** la numeración comienza desde uno hasta el número total de elementos que posea la lista).

A la **posición** de cada elemento se les llamará **índice**, siendo el índice cero (0) el perteneciente al primer elemento.

Ejemplo:

```
1 mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2 print(mi_lista[0])
3 print(mi_lista[3])
4 print(mi_lista[4])
```

```
"cadena de texto"
"otro dato"
25
```

En los ejemplos anteriores, podemos observar que 4 es el índice del último elemento en la lista; es así, ya que la lista tiene 5 elementos, y la numeración en este caso seria desde 0 hasta 4 (desde cero hasta una unidad menos a la cantidad de elementos que posea la lista).

También, podemos extraer una parte de la lista usando los índices. Para extraer una parte de la lista, es necesario especificar el índice desde el cuál deseamos iniciar la extracción, hasta una unidad más del índice que queremos finalizar la extracción. Veamos a continuación dos ejemplos, el primero extraemos desde el segundo elemento hasta el cuarto elemento, y en el segundo desde el tercer elemento hasta el último (recuerde el último elemento tiene índice 4).

Ejemplo:

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(mi_lista[1:4])
3  print(mi_lista[2:5])
```

```
» [15, 2.8, "otro dato"]
  [2.8, "otro dato", 25]
```

Además, a través del índice, podemos cambiar los elementos de una lista en el lugar que especifica dicho índice.

Ejemplo:

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(mi_lista[0])
3  mi_lista[0] = 2 #Cambiamos "cadena de texto" por 2.
4  print(mi_lista[0])
5
6  print(mi_lista[2])
7  mi_lista[2] = "¡Hola mundo!" #Cambiamos 2.8 por "¡Hola mundo
       !".
8  print(mi_lista[2])
```

```
» "cadena de texto"
  2

  2.8
  "¡Hola mundo!"
```

Como vemos en los ejemplos anteriores, podemos incluso cambiar elementos por otros totalmente diferentes, incluso si son de diferentes tipos (en este caso, cambiamos una cadena de texto por un entero y un real por una cadena de texto).

Por último, podemos agregar nuevos elementos a una lista, lo que significa, hacer más grande la lista (que tenga más elementos), esto lo podemos hacer con la función `append()`, la sintaxis es: `lista.append(elemento)`.

Ejemplo:

```
1  mi_lista = [2, 15, "¡Hola mundo!", "otro dato", 25]
2  print(mi_lista)
3  mi_lista.append("Nuevo Dato")
4  print(mi_lista)
```

```
[2, 15, "¡Hola mundo!", "otro dato", 25]
[2, 15, "¡Hola mundo!", "otro dato", 25, "Nuevo Dato"]
```

Observación: `append()` agrega el nuevo elemento al final de la lista, además, podemos agregar elementos de cualquier tipo, no solo cadenas de texto.

`len()`: esta función devuelve la longitud de la lista (su cantidad de elementos). La sintaxis para usar esta función es: `len(lista)`.

Ejemplos:

```
1  print(len([2, 1, "hola"])) #La lista tiene 3 elementos (2, 1 y↘
       "hola").
```

```
3
```

```
1  mi_lista = ["cadena de texto", 15, 2.8, "otro dato", 25]
2  print(len(mi_lista))
```

```
5
```

Concatenar listas: es la acción de unir o enlazar listas. Las listas se pueden concatenar con el símbolo de la suma (+). La sintaxis es: `lista_1 + lista_2`.

Ejemplo:

```
1  lista1 = [1,2,3]
2  lista2 = ["hola", "saludos"]
3  print(lista1 + lista2) #Concatenamos la lista1 con la lista2.
```

```
[1, 2, 3, "hola", "saludos"]
```

Observación: al concatenar dos listas, el resultado es una nueva lista que contiene en las primeras posiciones los elementos de la lista a la izquierda del símbolo +, y luego van los elementos de la segunda lista a la derecha del símbolo +.

También, podemos usar la concatenación para agregar un elemento de cualquier tipo a una lista. La sintaxis es la siguiente: `lista += [elemento_nuevo]` o `lista = lista + [elemento_nuevo]`.

Nota: el operador suma (+) necesita que los dos operandos sean listas.

Ejemplo:

```
1  lista1 = [1,2,3]
2  lista1 += ["Nuevo dato"]
3  print(lista1)
```

```
[1, 2, 3, "Nuevo dato"]
```

Observación: el nuevo elemento a agregar debe estar siempre entre corchetes, esto es así, ya que + necesita dos elementos del tipo lista. También, notemos que el resultado es el mismo si usáramos `append()` en la lista1.

2.5 Funciones en Python

Una **función** es una agrupación de expresiones y sentencias (algoritmos) que realizan determinadas acciones, pero que éstas, solo se ejecutan cuando son llamadas. Es decir, si se coloca un algoritmo dentro de una función y luego éste se corre, el algoritmo no será ejecutado, dado que no se ha hecho una referencia a la función que lo contiene.

El uso de funciones es un componente muy importante del paradigma de la programación llamada *estructurada*[5], y tiene varias ventajas:

- **Modularización:** permite segmentar un programa complejo en una serie de partes o módulos más simples, facilitando así la programación y el depurado.
- **Reutilización:** permite reutilizar una misma función en distintos programas.

Python dispone de una serie de funciones integradas al lenguaje (ejemplos: `range()` y `len()`), y también permite crear funciones definidas por el usuario para ser usadas en su propios programas. En esta sección, nos ocuparemos de cómo crear tus propias funciones.

2.5.1 Definiendo funciones

En **Python**, la definición de funciones se realiza mediante la instrucción `def` más un nombre de función descriptivo (se recomienda que el nombre de la función haga alusión a la acción que realiza) seguido de paréntesis de apertura y cierre. Como toda estructura de control en **Python**, la definición de la función finaliza con dos puntos (`:`) y el algoritmo que la compone irá indentado:

[5] La programación estructurada es un paradigma de programación basado en utilizar funciones o subprogramas, y únicamente tres estructuras de control: **secuencia** (sentencias), **selección o condicional** (**if**) e **iteración** (ciclo o bucle)

La sintaxis para crear una función en **Python** es:

```
def mi_funcion():
    instrucciones
```

Donde:

- `mi_funcion`: es el nombre de la función.
- `instrucciones`: es el bloque de sentencias en lenguaje **Python** que realizar cierta operación dada.

Una vez creada la función, podemos usarla simplemente escribiendo el nombre de la función, que en este caso sería: `mi_funcion()`

Ejemplo:

```
1  def texto():
2      print("¡Hola mundo!")
3  texto()
```

```
¡Hola mundo!
```

2.5.2 Parámetros

Un **parámetro** es un valor que la función espera recibir cuando sea llamada, a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán **separados por una coma**, entre los paréntesis que van seguidos después del nombre de la función) o ninguno[6].

La sintaxis para crear una función con parámetros en **Python** es:

```
def mi_funcion(lista_de_parametros):
    instrucciones
```

Donde:

- `mi_funcion`: es el nombre de la función.
- `lista_de_parametros`: es la lista de parámetros que puede recibir una función.
- `instrucciones`: es el bloque de sentencias en lenguaje **Python** que realizar cierta operación dada.

[6]Como vimos en la sección anterior, para crear una función no es necesario que tenga parámetros.

La manera para llamar una función con parámetros, es usando el nombre de la función seguido de la lista de parámetros encerrados entre paréntesis. Para este caso sería: `mi_funcion(lista_de_parametros)`.

Nota: Al definir una función, los valores que se recibe se denominan **parámetros**, pero durante la llamada los valores que se envían se denominan **argumentos**.

Ejemplo:

```
1  def mi_funcion(a,b):
2      print(a-b) #Esta función resta al primer parámetro, el segundo parámetro.
3  mi_funcion(5,3)
```

```
2
```

Observación: La función del ejemplo anterior tiene dos argumentos, y estos deben ser del tipo entero o real, entonces, si esta misma función es llamada con parámetros de otro tipo para los cuales el operador $-$ no esté definido, generará un error en su ejecución (lo mismo sucederá, si se mezclan diferentes tipos de variable, por ejemplo `int` con `str`).

Además, si el operador soportara otros tipos de variable, el resultado ya no sería lo que esperamos (la resta de dos números), por esto será necesario validar cada una de las variables, antes de la ejecución de las instrucciones que definirán la finalidad de la función.

2.5.2.1. Parámetros por defecto u omisión

Si una función fue creada con parámetros y es llamada (o invocada) sin argumentos, generará un error de ejecución. Para solucionar este problema, podemos crear los parámetros con ciertos valores para que la función se ejecute con los valores establecidos en caso que sea llamada sin argumentos.

La sintaxis para los parámetros sería:

```
mi_funcion(a_1 = valor_1, a_2 = valor_2, ... , a_n = valor_n):
```

Ejemplo:

```
1  def mi_funcion(a = 1,b = 3):
2      print(a-b)
3
4  mi_funcion(5,3)
5  mi_funcion()
6  mi_funcion(6)
```

```
2
-2
```

```
»  3
```

Como podemos observar, al parámetro *a* se le ha asignado 1 y a *b*, 2, en caso que la función sea llamada sin argumentos, por esta razón cuándo ejecutamos `mi_funcion()`, el resultado[7] en pantalla es -2.

Además, note se que en el caso de la instrucción `mi_funcion(6)`, la función toma como valor para *a* el número 6 (en el caso de a ver más parámetros y menos argumentos, **Python** irá asignando en orden de izquierda a derecha, argumento con parámetro).

Veamos otro ejemplo donde la función es definida con parámetros de diferentes tipos.

Ejemplo:

```
1  def funcion_2(numero,texto):
2      print(2**numero)
3      print("El texto ingresado es: ", texto)
4
5  funcion_2(3,"Hola")
```

```
»  8
   El texto ingresado es: Hola
```

En el ejemplo anterior, notese que el primer argumento debe ser un número, y el segundo una cadena de texto. Así, si ingresáramos argumentos de otro tipo, por ejemplo, escribiéramos la instrucción `funcion_2("Hola",3)`, esto causaría un error de ejecución ya que 2 no se puede elevar a una cadena de texto, por lo tanto, el orden con que creamos los parámetros y el tipo que variable que le «asociemos» en las instrucciones de la función, deben ser respetados por los argumentos a la hora de llamar a la función.

Nota: Tenga en cuenta que al momento de llamar una función que fue definida con parámetros, los siguientes casos pueden generar un error de ejecución:

1. Si tiene más argumentos que parámetros.
2. Si el número de argumentos es igual al de parámetros, pero no se colocan correctamente (en el orden que fueron creados).

2.5.3 Argumentos por nombre

Una forma de evadir el segundo problema de ejecución, presentado al final del apartado anterior, es llamar a la función utilizando el nombre de cada parámetro

[7] $a - b = 1 - 3 = -2$

y especificando su valor.

Ejemplo:

```
1       def funcion_2(numero,texto):
2           print(2**numero)
3           print("El texto ingresado es: ", texto)
4
5       funcion_2(texto = "Hola", numero = 3)
```

```
»   8
    El texto ingresado es: Hola
```

Observación: El inconveniente de esta forma de llamar a una función, es que necesitamos saber cuál es el nombre de cada parámetro de la función.

2.5.4 `None`, `return` y `pass`

`None`: Es el único valor del tipo `NoneType`. `None` se usa con frecuencia para representar la ausencia de un valor, como cuando los argumentos predeterminados no se pasan a una función.

Como se mencionado anteriormente, si no deseamos colocar valores por defecto a los parámetros de una función, podemos usar `None`, y mostrar un mensaje en pantalla, indicando que no se colocaron argumentos.

Ejemplo:

```
1       def mi_funcion(a = None,b = None):
2           if a == None or b == None:
3               print("Error, se necesitan dos números como argumentos↘
                    para ejecutar la función")
4           else:
5               print(a-b)
6
7       mi_funcion()
8       mi_funcion(6)
9       mi_funcion(5,3)
```

```
»   Error, se necesitan dos números como argumentos para ejecutar la
        función
    Error, se necesitan dos números como argumentos para ejecutar la
        función
    2
```

`return`: Esta instrucción «devuelve» el resultado o valor de la expresión que le sigue a `return` en la misma línea de código. `return` es usado en funciones para finalizar, y devolver un valor o expresión en concreto, las instrucciones posteriores a `return` no se ejecutan. Si la sentencia `return` no tiene ninguna expresión, por defecto devolverá `None`.

Nota: `return` no se puede usar fuera de una función.

Ejemplo:

```
1   def mi_funcion(a = None,b = None):
2       if a == None or b == None:
3           print("Error, se necesitan dos números como argumentos
                para ejecutar la función")
4       else:
5           return(a-b)
6
7   mi_funcion(5,3)
8   print(mi_funcion(5,3))
9
10  x = mi_funcion(5,3)
11  print(x)
```

```
2
2
```

Observación: Como podemos observar en el ejemplo anterior, solo usar la instrucción: `mi_funcion(5,3)`, esta no muestra nada en pantalla, sin embargo, si la colocamos ahora dentro de un `print()`, podremos ver el valor que está devolviendo la función.

La ventaja más grande de usar `return`, es la de poder guardar los resultados de las funciones en variables y así poder reutilizar estos valores en otros programas o funciones (como en el ejemplo anterior, podemos observar que se le asigna a la variable *x*, el valor que devuelve la función, con la instrucción `x = mi_funcion(5,3)`).

Una característica muy importante de `return`, es la posibilidad de devolver valores múltiples separados por comas.

Nota: En el caso de devolver múltiples valores una función, `return` las devuelve en un tipo muy especial de elemento, conocido como **tuplas**, más adelante estudiaremos este tipo de elemento.

Ejemplo:

```
1   def mi_funcion(a = None,b = None):
2       if a == None or b == None:
3           print("Error, se necesitan dos números como argumentos
                para ejecutar la función")
4       else:
5           return(a-b,a,b,[a,b])
6
7   print(mi_funcion(5,3))
```

```
(2, 5, 3, [5, 3])
```

`pass`: es una operación nula, cuando se ejecuta, no sucede nada. Es útil como

marcador de posición cuando se requiere una declaración, pero no es necesario ejecutar ningún código. A veces en la elaboración de programas, tenemos claro desde un inicio cuántas funciones llevará, pero no así el código de ellas, así podemos usar pass para declarar las funciones y posteriormente crear sus algoritmos.

Ejemplo:

```
def funcion_3():
    pass #Esta función no hace nada.

funcion_3()
```

2.6 Alcance y variables globales

Las variables por defecto son locales en **Python**, esto quiere decir que las variables definidas dentro de una función no podrán ser utilizadas fuera de ellas. Para entender mejor lo descrito antes, veamos un ejemplo.

Ejemplo:

```
def suma(x,y):
    a=1
    return x+y

print(a)
```

```
NameError: name 'a' is not defined
```

Notemos que en el ejemplo anterior dentro de la función nos encontramos a la variable `a`, la cual tiene asignado el valor de 1. Podríamos pensar que luego de haber creado la función podemos reutilizar la variable `a`, pero no es así, ya que al intentar imprimir en pantalla la variable `a`, **Python** nos dice que no está definida.

Ahora, veamos un caso diferente, donde fuera de la función se define una variable y se intenta usar dentro de la función.

Ejemplo:

```
b=2
def suma(x,y):
    print(b)
    return x+y

print(suma(2,3))
```

```
2
5
```

En este caso, sí, es posible reutilizar la variable `b` dentro de la función (ya se

muestra el valor de *b* en pantalla). El carácter local o global de la variable dependerá desde qué estructura veamos la variable, ya que si tenemos un programa principal, y definimos variables (pero no dentro de una función u otra estructura que contenga el programa principal), entonces tendremos una **variable global**. La razón de porqué llamarla global, es porque se podrán usar en todas las estructuras que el programa principal contenga. Por otro lado una **variable local**, solo es posible usarla en la estructura donde fue definida y no fuera de ella.

Observación: La variable *b* recibe el nombre de **variable global**, porque podemos usarla dentro de la función, por otro lado a la variable *a* se le llamará **variable local**, porque no es posible usarla fuera de la función.

A pesar del carácter local de las variables definidas en una función, podemos cambiar el carácter local de la variable con `global`. La sintaxis para `global` seria: `global variable`, ahora veamos un ejemplo de esto.

Ejemplo:

```
1       def suma(x,y):
2           global a
3           a=1
4           return x+y
5
6       print(a)
```

```
»       1
```

Observación: Como vemos la variable *a* ha dejado de ser local, ya que ahora ya se puede usar fuera de la función. En este caso la instrucción que transforma la variable *a* a global es `global a`.

2.7 Errores y excepciones

En la medida que interactuamos con **Python**, nos podemos encontrar con algunos errores, estos los podemos clasificar como **errores de sintaxis** y **errores de excepción**. El primer tipo, es quizás el más conocido, ya que este aparece en pantalla cuando escribimos algo que no está dentro del lenguaje de **Python**.

Ejemplo:

```
1       if x
```

```
»       SyntaxError: invalid syntax
```

Observación: Como vemos no se ha escrito correctamente la sentencia del condicional, ni las instrucciones que deben ejecutarse de cumplirse `if`, por ende no lo entiende **Python**, y por lo tanto, genera un error de sintaxis.

El segundo tipo de errores que podemos encontrar en **Python** son los errores de excepción, este tipo son el resultado de un error lógico o de operación no válida, en otras palabras está bien escrito en el lenguaje **Python**, pero no tiene sentido. Veamos algunos ejemplos:

Ejemplos:

```
1    1/0
```

```
ZeroDivisionError: division by zero
```

```
1    "2"+[2]
```

```
TypeError: can only concatenate str (not "list") to str
```

Nota: Estos tipos de errores hacen que se detenga la ejecución del programa.

Como vemos, los errores producidos son errores donde no está definida la operación, sin embargo están correctamente bien definida en el lenguaje de **Python**.

Los errores de excepción podemos de alguna manera controlarlos, y hacer que nuestro programa no pare su ejecución, esto lo logramos con `try` y `except`. La partícula `try` nos permite buscar errores en un bloque de código. Por otro lado `except`, te permite manejar los errores encontrados, especificando en pantalla que ha sucedido un error, pero permitiendo que el programa continué su ejecución con normalidad.

Ejemplo:

```
1    try:
2        1/0
3    except:
4        print("No se puede dividir por cero")
```

```
No se puede dividir por cero
```

En los primeros ejemplos vimos como **Python** ya clasifica estos tipos[8] de errores, por ejemplo $1/0$, es un error del tipo `ZeroDivisionError`, entonces nosotros podemos ir buscando un tipo de error en específico en nuestros códigos por medio de `except`.

Ejemplo:

```
1    try:
2        1/0
3    except ZeroDivisionError:
```

[8]Una lista completa de los diferentes tipos de excepciones que podemos usar en Python 3, lo podemos encontrar en el enlace siguiente: `https://docs.python.org/3/library/exceptions.html#bltin-exceptions`

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

```
»   No se puede dividir por cero
```

Observación: En el ejemplo podemos ver que estamos buscando el error de dividir por cero, sin embargo es probable que nuestros códigos presenten otros tipos de error, por ello se sugiere agregar otro `except`, pero sin especificar el tipo de error.

Ejemplo:

```
1   try:
2       "2"+[2]
3   except ZeroDivisionError:
4       print("No se puede dividir por cero")
5   except:
6       print("Hay otro error")
```

```
»   Hay otro error
```

Observación: Como vemos podemos tener más de un `except`. Sin embargo no puede existir más de un `try` sin un `except`.

Normalmente `try` y `except` son utilizados para validar el funcionamiento de una función. Veamos un ejemplo de esto:

Ejemplo:

```
1   def suma(x,y):
2       try:
3           r = x+y
4           return r
5       except:
6           print("Operación incorrecta.")
7
8   print(suma(2,3))
```

```
»   5
```

2.8 Documentación de funciones

La documentación de una función es muy importante, ya que con esto dejamos claro qué valores espera recibir la función, así como el tipo que deben ser y qué hace la función con ellos. Además de especificar qué valor o valores devolverá, si es que devuelve algo la función. También, se colocan ejemplos de implementación de la función creada, y datos del creador si es necesario. Todo esto con el objetivo que el usuario sepa cómo trabajar con la función creada.

Ejemplo:

```
1       def suma(x,y):
2           """
3           Esta función suma dos números reales
4           Valores de entrada: x, y (deben ser números reales)
5           Valores de salida: r (este es el resultado de sumar x con ↘
                y)
6
7           Ejemplo:
8           >>> suma(2,3)
9           5
10
11          Creador: Luisantos Bonilla (09/05/2020)
12          """
13          r = x + y
14          return r
15
16      print(suma(3,4))
```

```
»   7
```

Nota: La documentación de una función debe colocarse siempre entre las triples comillas, o sea la sintaxis es `"""texto"""`.

La documentación nunca se despliega en pantalla mientras se ejecuta la función, ya que es un recurso de carácter informativo, así para que podamos ver dicha documentación es necesario usar `help`.

Ejemplo:

```
1       def suma(x,y):
2           """
3           Esta función suma dos números reales
4           Valores de entrada: x, y (deben ser números reales)
5           Valores de salida: r (este es el resultado de sumar x con ↘
                y)
6
7           Ejemplo:
8           >>> suma(2,3)
9           5
10
11          Creador: Luisantos Bonilla (09/05/2020)
12          """
13          r = x + y
14          return r
15
16      help(suma)
```

```
»   Help on function suma in module __main__:

    suma(x, y)
        Esta función suma dos números reales
        Valores de entrada: x, y (deben ser números reales)
        Valores de salida: r (este es el resultado de sumar x con y)
```

```
»
        Ejemplo:
        >>> suma(2,3)
        5

        Creador: Luisantos Bonilla (09/05/2020)
```

Observación: Notese que para usar `help`, solo es necesario colocar entre paréntesis el nombre de la función (ejemplo: `help(suma)`).

2.9 Ejercicios - 1

1. Escriba un programa que genere y guarde los primeros n números de Fibonacci en una lista.

2. Dadas dos listas, hacer un programa que imprima los elementos de la primera lista en orden normal (respecto a su índice), y la segunda en orden inverso, como por ejemplo:

```
>>> lista = [1, 24, 5]
1, 24, 5
>>> lista2 = [2, 4, 45, 6, 7, 89]
89, 7, 6, 45, 4, 2
```

3. Escribir un programa que sume dos listas, elemento por elemento (las listas solo deben tener elementos enteros o reales). Además, considere el hecho que pueden ser listas de diferentes tamaños.

4. Considere el polinomio $P(x) = x^3 - x + 1$ y la lista `[3, 6, 8, 9]`. Escribir un programa que evalúe cada elemento de la lista anterior en $P(x)$ y cada resultado se guarde en una nueva lista (no olvide mostrar la nueva lista en pantalla).

5. Escribe un programa que concatene dos listas, pero sin usar $+$.

6. Escribir un programa que tome dos listas y las concatene, pero que el resultado sea una lista con elementos alternados de las dos listas dadas, por ejemplo:

```
[1,2,3], ["a","b"] -> [1, "a", 2, "b", 3]
```

7. Escribe un programa que dado un número entero n positivo, genere en pantalla el triángulo de Pascal. Por ejemplo, si $n = 5$, el resultado que se espera es:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

2.10 Ejercicios - 2

Indicaciones: Crea un archivo `.py` para cada ejercicio. Además, valida los parámetros de cada función teniendo en cuenta el tipo de valor a usar. También, en el archivo debes implementar los ejemplos necesarios para corroborar el buen funcionamiento de la función.

1. Cree una función que devuelva el área de un triángulo dadas las coordenadas de sus vértices. **Nota:** El área de un triángulo se puede calcular con la siguiente formula:

$$A = \frac{1}{2}|x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1|$$

   Donde (x_1, y_1), (x_2, y_2) y (x_3, y_3) son los vértices del triángulo.

2. Cree un función que dada una lista de números devuelva la **moda** de la lista de números presentado.

3. Escribir una función que dado un número entero, devuelva otro con sus dígitos invertidos.

4. Escriba una función que devuelva el elemento más grande y el más pequeño en una lista..

5. Escribir una función que reciba un número entero y devuelva una cadena con una representación como esta: `1569 = 1*10^3+5*10^2+6*10^1+9*10^0`

6. Escribir una función que resuelva un sistema de ecuaciones lineales de dos incógnitas.

7. Diseña una función que reciba una lista de cadenas y devuelva el prefijo común más largo. Por ejemplo, la cadena `"pol"` es el prefijo común más largo de esta lista:

```
["poliedro", "policía", "polífona", "polinizar", "polaridad", "política"]
```

8. Escribir una función que calcule la cantidad de días que han transcurrido en un año hasta una fecha dada.

9. Diseña una función que devuelva el valor absoluto de la máxima diferencia entre cualquier par de elementos de una lista. Por ejemplo, el valor devuelto para la lista `[1, 10, 2, 6, 8, 2]` es 9, pues es la diferencia entre el valor 10 y el valor 1.

10. Escriba una función que rote una lista (sin hacer copias de la lista dada) *n* espacios a la izquierda cada elemento.

11. Escribir una función que tome una ecuación de primer grado como texto, y que calcule la solución.

3. MÓDULOS

3.1 Módulos en Python

Un módulo es una agrupación de funciones, constantes y clases[9] en un solo script **.py**, en otras palabras, un módulo es un programa donde se encuentran definidas funciones, constantes y clases.

Un ejemplo de módulo es el script **hermite.py** que podemos encontrar en la biblioteca o librería llamada **Numpy**[10], el módulo hermite tiene muchas funciones definidas para trabajar con polinomios, una de ellas **poly2herm** que convierte un polinomio a una serie de Hermite. Como se puede intuir del ejemplo anterior un módulo no es otra cosa que un archivo con extensión **.py**.

3.1.1 Usando Módulos

[9] Las clases son estructuras más complejas que las funciones en **Python**. las clases serán estudiadas en el curso de Programación II.

[10] Al inicio del curso se abusó del término módulo especial, sin embargo, debe quedar claro que Numpy es una librería. Las librerías serán tratadas como módulo, ya que la sintaxis para usar sus funciones es similar.

3.1.1.1. Import

Para usar un módulo en otro programa, debe usarse la sentencia `import`, al inicio de nuestro programa o antes de usar alguna función del módulo. La sintaxis es: `import nombre_del_modulo`

Una vez importado el módulo, la sintaxis para usar una función del módulo debe ser:

```
nombre_del_modulo.nombre_de_la_funcion(argumentos)
```

Lo mismo aplica si queremos usar las constantes del módulo, solo debemos cambiar el nombre de la función por el nombre de la constante.

Ejemplos:

```
1  import numpy
2  a=4
3  print(a)
4
5  print(numpy.cos(1))
```

```
4
0.5403023058681398
```

```
1  a=4
2  print(a)
3  import numpy
4  print(numpy.cos(1))
```

```
4
0.5403023058681398
```

Observación: Notemos que para usar las funciones que las librerías[11] contienen, se usa la misma sintaxis que para los módulos, por está razón los trataremos como módulos especiales.

3.1.1.2. From e Import

Otra forma de usar `import` es usándolo junto a `from`, esto permite usar las funciones y constantes de manera usual, solo usando sus nombres, sin hacer referencia al módulo al que pertenece. La sintaxis es: `from nombre_del_modulo import *`

El asterisco que aparece en la instrucción `import`, hace que todas las funciones y constantes del módulo estén disponibles para el programa.

[11] Las librerías son un conjunto de módulos que se interrelacionan entre sí.

Ejemplos:

```
1  from numpy import *
2  a=4
3  print(a)
4
5  print(cos(1))
```

```
4
0.5403023058681398
```

```
1  a=4
2  print(a)
3  from numpy import *
4  print(cos(1))
```

```
4
0.5403023058681398
```

A veces solo estamos interesados en usar ciertas funciones de un módulo, para esto solo debemos cambiar el asterisco por el nombre de la función que necesitamos. La sintaxis es:

```
from nombre_del_modulo import nombre_de_la_funcion
```

Nota: La instrucción anterior solo hará que el programa reconozca la función que hayamos especificado, así será imposible usar otra función distinta a las que hayamos importado del módulo.

Ejemplos:

```
1  from numpy import cos
2  a=4
3  print(a)
4
5  print(cos(1))
```

```
4
0.5403023058681398
```

```
1  a=4
2  print(a)
3  from numpy import cos
4  print(cos(1))
```

```
4
0.5403023058681398
```

Observación: En este ejemplo será imposible usar otra función del módulo especial numpy, porque se especificó que la función `cos` sea la única que se importe,

si quisiéramos otra función deberíamos agregarla después de `cos`, separándolos por comas, por ejemplo `import cos, sin`, así tendremos a disposición la función `cos` y `sin`.

3.2 Creación de módulos

A lo largo del curso hemos estado creando módulos inconscientemente (de alguna manera claro), ahora para crear un módulo estrictamente debemos crear un archivo con extensión **.py**, y dicho archivo debe contener solo funciones y constates (si fuese necesario). Nuestros módulos no deben contener pruebas de las funciones, porque serán cargadas por defecto a la hora de importar el módulo.

Ejemplo: Considere el módulo llamado **aritmética.py**, que contenga la siguientes funciones:

```
def sumar(x,y):
    return x+y

def restar(x,y):
    return x-y
```

Observación: Este archivo que solo contiene dos funciones es un módulo con nombre aritmética.

Una pregunta valida sería: ¿cómo puedo hacer pruebas para saber si las funciones del módulo están bien hechas? Para ello debemos usar la siguiente instrucción: `if __name__=='__main__':` al final del archivo (módulo).

Ejemplo: Retomemos el módulo aritmética.

```
def sumar(x,y):
    return x+y

def restar(x,y):
    return x-y

if __name__=='__main__':

    print("Ejemplos:")

    print(sumar(3,4))

    print(resta(7,3))
```

Observación: La instrucción `if __name__=='__main__':`, permite que cuando se importe el módulo las pruebas no se ejecuten, porque literalmente pregunta si el archivo que se está ejecutando es el principal. En el caso que se esté importando el módulo esta instrucción es falsa, porque el nombre del archivo principal es otro, no es el del módulo, ya que este último se está usando por otro archivo.

Agregando `if __name__=='__main__':` a los módulos, permite que el módulo se comporte como un programa (cuando se desee actualizar o corregir algún error) y posteriormente como un módulo cuando se importe.

3.2.1 Usando As

A veces los nombres que colocamos a nuestro módulos son muy largos, y si estamos usando solamente `import` para impórtalos, resultaría un poco incómodo de escribir siempre el nombre del módulo cuando usamos sus funciones. Para una mayor comodidad, podemos usar la instrucción `as`, esta instrucción nos permite renombrar el nombre de los módulos en nuestro programa, para un mejor manejo de la escritura del programa donde estamos usándolos.

Ejemplo:

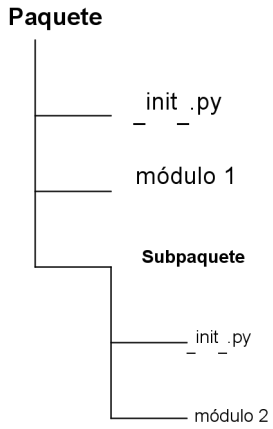
```
1  import numpy as npy
2  a=4
3  print(a)
4
5  print(npy.cos(1))
```

```
4
0.5403023058681398
```

3.2.1.1. Paquetes y módulos

Un paquete es una carpeta que contiene módulos, y posiblemente otros paquetes y un archivo `__init__.py`[12]. En la siguiente imagen mostramos la relación entre paquete y módulo:

[12]El archivo `__init__.py` es lo que permite que se considere una carpeta como paquete o no, ya que en un paquete, dicho archivo siempre está (puede incluso ser un archivo vacío).



Para poder usar un paquete en nuestros programas, debemos usar las sentencias `from` y `import` de la siguiente manera:

```
from nombre_del_paquete import nombre_del_modulo
```

Ejemplo: Consideremos que el módulo aritmética.py está en un paquete llamado matemática, así consideré el siguiente programa donde importamos el módulo aritmética del paquete matemática.

```
1  from matemática import aritmética
2  a=4
3  print(a)
4
5  print(aritmética.sumar(1,2))
```

```
4
3
```

Observación: Notemos que el módulo es llamado por la instrucción `import`, en ese caso será necesario escribir el nombre del módulo, seguido de un punto y finalizando con el nombre de la función que deseamos usar.

3.3 Ejercicios

Indicaciones: En cada uno de los siguiente problema debe crearse un programa modular[13] (que funcione como módulo y programa), que facilite la creación de un

[13]Este módulo debe contener todas las funciones auxiliares que se requieran para la creación del programa o programas que se solicitan en cada literal.

segundo programa (este **no** debe ser un módulo) importando el programa modular creado previamente.

1. Un programa que desarrolle $(x+y)^n$, donde el único parámetro es n.
2. Escribir un programa interactivo, que le permita al usuario convertir una temperatura de grados Fahrenheit a grado Celsius y viceversa.
3. Escribir un programa que devuelva la suma de los múltiplos de 3 y 5 que están entre 0 y n (parámetro).
4. Escribir un programa que imprima en pantalla los números primos entre 1 y n (parámetro).
5. Escribir un programa que verifique la velocidad de los conductores. Dada una velocidad (en kilómetros por hora) se verifique y aplique lo siguiente:
   - Si la velocidad es a lo sumo 60 km/h, imprimir en pantalla "ok".
   - Si la velocidad es mayor a 60, por cada 5 km por encima del límite de velocidad (60), se le asignará al conductor un punto de penalización e imprimir el número total de puntos de penalización. Por ejemplo, si la velocidad es 80, debería imprimir en pantalla: "Puntos de penalización: 4".
   - Si el conductor obtiene más de 12 puntos, la función debería imprimir: "Licencia suspendida".

4. Secuencias en Python

4.1 Tuplas y listas

Ya hemos estudiado al inicio del curso las listas, en este apartado no solo retomamos el estudio de las listas, sino también conoceremos un tipo de dato de **Python** muy parecido a las listas, las cuales llamaremos **tuplas**.

4.1.1 Tuplas

Tuplas: Las tuplas son secuencias al igual que las listas, con la diferencia que son inmutables una vez creadas. Significa que una vez hechas, ya no se podrán modificar de ninguna manera. La sintaxis de las tuplas es: `nombre_tupla =(elementos)`.

Ejemplo:

```
>>> mi_tupla = (1,2,"hola",[1,(2,3)],(5))
```

Observación: Al igual que las listas, las tuplas pueden contener una combinación de los diferentes tipos de datos, incluso puede contener otra tupla como elementos.

El tipo de datos para las tuplas es `tuple`, además por ser una secuencia, sus elementos están indexados, y su numeración es de cero hasta una unidad menos a la cantidad de elementos que contenga la tupla. Como podemos observar las tuplas son básicamente listas que no podemos modificar.

Ejemplos:

```
1  mi_tupla = (1,2,"hola",[1,(2,3)],(5))
2  print(mi_tupla[1])
3  print(mi_tupla[0])
4  print(mi_tupla[3])
5  print(len(mi_tupla))
```

```
2
1
[1, (2, 3)]
5
```

Observación: También podemos usar `len()` en las tuplas para saber cuántos elementos posee. No debe olvidarse que al igual que las listas, el índice del primer elemento es 0 y del último elemento es `len(mi_tupla)-1`.

4.2 Métodos

De manera informal los métodos[14] son funciones que son exclusivamente para tipos de datos que pueden contener otros elementos[15] dentro de **Python**. En este curso solo hablaremos de métodos para cadenas de texto, listas y tuplas.

La sintaxis para usar métodos es: `nombre_dato.nombre_metodo(argumentos)`. A continuación presentamos los métodos más usados para cadenas de texto, listas y tuplas.

4.2.1 Métodos para cadenas de texto.

1. `capitalize ()`: Devuelve una copia de la cadena con el primer carácter en mayúscula.

[14] La definición correcta será presentada en el curso de programación II.
[15] Estos elementos pueden ser de los tipos que conocemos, por ejemplo `int`, `float` o `list`.

Ejemplo:

```
1  mi_texto="hola mundo"
2  print(mi_texto.capitalize())
```

```
»  "Hola mundo"
```

2. `count(elemento,inicio,final)`: Devuelve cuántas veces aparece `elemento` en la cadena desde el índice `inicio` hasta el índice `final`.

   Nota: Los argumentos `inicio` y `final` son opcionales y se interpretan según la notación de corte, es decir, si solo se coloca uno, se entenderá que se busca desde el índice cero hasta el que se le ha escrito.

   Ejemplos:

```
1  mi_texto="hola mundo"
2  print(mi_texto.count("o",2,4))
3  print(mi_texto.count("o",2))
4  print(mi_texto.count("o"))
```

```
»  0
   1
   2
```

   Observación: En el último ejemplo, ya que no se indica el intervalo para buscar dentro de la lista, simplemente busca en toda la cadena de texto.

3. `find(elemento,inicio,final)`: Devuelve el menor índice de la cadena para el que `elemento` se encuentre, de tal modo que `elemento` quede contenido en el rango desde `inicio` hasta antes de `final`.

   Nota: Los argumentos `inicio` y `final` son opcionales y se interpretan exactamente igual que en el caso de `verb`. Devuelve -1 si no se halla `elemento`.

   Ejemplos:

```
1  mi_texto="hola mundo"
2  print(mi_texto.find("o",1,3))
3  print(mi_texto.find("o"))
4  print(mi_texto.find("w"))
```

```
»  1
   1
   -1
```

4. `isalnum()`: Devuelve verdadero si todos los caracteres de la cadena son alfanuméricos. En caso contrario, devuelve falso.

Ejemplos:

```
1 mi_texto="hola mundo"
2 print(mi_texto.isalnum())
```

```
False
```

```
1 texto2 = "HolaMundo"
2 print(texto2.isalnum())
```

```
True
```

```
1 texto3 = "HolaMundo2"
2 print(texto3.isalnum())
```

```
True
```

```
1 texto4 = "HolaMundo%"
2 print(texto4.isalnum())
```

```
False
```

5. `isalpha()`: Devuelve verdadero si todos los caracteres de la cadena son alfabéticos. En caso contrario, devuelve falso.

Ejemplos:

```
1 mi_texto="hola mundo"
2 print(mi_texto.isalpha())
```

```
False
```

```
1 texto2 = "HolaMundo"
2 print(texto2.isalpha())
```

```
True
```

```
1 texto3 = "HolaMundo2"
2 print(texto3.isalpha())
```

```
False
```

```
1 texto4 = "HolaMundo%"
2 print(texto4.isalpha())
```

```
False
```

6. `isdigit()`: Devuelve verdadero si todos los caracteres de la cadena son dí-

gitos. En caso contrario, devuelve falso.

Ejemplos:

```
texto="hola mundo"
print(texto.isdigit())
```

```
False
```

```
texto2="HolaMundo"
print(texto2.isdigit())
```

```
False
```

```
texto3="HolaMundo2"
print(texto3.isdigit())
```

```
False
```

```
texto4="HolaMundo%"
print(texto4.isdigit())
```

```
False
```

```
texto5 = "123"
print(texto5.isdigit())
```

```
True
```

Una lista con el resto de métodos para cadenas de texto, lo podrán encontrar en el siguiente enlace: `https://docs.python.org/2.5/lib/string-methods.html`

4.2.2 Métodos para las listas.

1. `append(elemento)`: Este método agrega `elemento` al final de una lista.

   Ejemplo:

   ```
   lista = [1,2,3]
   lista.append("xD")
   print(lista)
   ```

   ```
   [1, 2, 3, "xD"]
   ```

2. `count(elemento)`: Devuelve el número de veces que `elemento` aparece en la lista.

Ejemplo:

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

```
3
```

3. `extend(argumento_lista)`: Este método extiende una lista agregando uno a uno los elementos de `argumento_lista` al final.

   Ejemplos:

```
lista = [1,2,3]
lista.extend(["x","D"])
print(lista)
```

```
[1, 2, 3, "x", "D"]
```

```
lista = [1,2,3]
lista.extend(range(2))
print(lista)
```

```
[1, 2, 3, 0, 1]
```

4. `index(elemento,inicio,final)`: Este método recibe un `elemento` como argumento y devuelve el índice de su primera aparición en la lista, de tal modo que `elemento` esté contenido en el rango desde `inicio` hasta `final`.

   Nota: Los argumentos `inicio` y `final` son opcionales y si solo se coloca uno, se entenderá que se busca en ese índice el elemento. Si el elemento no se encuentra en la lista dará `ValueError`.

   Ejemplos:

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

```
1
```

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

```
1
```

5. `insert(i,elemento)`: Este método inserta el `elemento` en la lista, en el índice `i`.

Ejemplo:

```
lista = [1,2,3,4,5]
lista.insert(3,0)
print(lista)
```

```
[1, 2, 3, 0, 4, 5]
```

6. `remove(elemento)`: Este método recibe como argumento un elemento, y borra su primera aparición en la lista.

   Nota: Si el elemento no se encuentra en la lista dará `ValueError`.

   Ejemplo:

```
lista = [1, 2, 3, 0, 4, 5]
lista.remove(0)
print(lista)
```

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

7. `pop(i)`: Este método elimina el elemento en el índice `i` y lo devuelve.

   Nota: El argumento `i` es opcional, y si no se coloca eliminará y devolverá el último elemento de la lista.

   Ejemplos:

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

print(lista)
```

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

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

print(lista)
```

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

8. `clear()`: Elimina todos los elementos de la lista.

   Ejemplo:

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

```
[]
```

9. `sort()`[16]: Este método ordena los elementos de una lista de menor a mayor.

   Ejemplo:

```
1  lista = [1,25,76,8,9]
2  lista.sort()
3  print(lista)
```

```
[1, 8, 9, 25, 76]
```

Una lista con el resto de métodos para listas, lo podrán encontrar en el siguiente enlace: `https://docs.python.org/3/tutorial/datastructures.html`

4.2.3 Métodos para tuplas

Las tuplas al ser muy similares a las listas comparten los mismos métodos salvo que estos no modifiquen sus valores y estructura de como fueron creados. Por ejemplos los siguientes métodos se pueden usar en las tuplas, y se usan de igual forma que con las listas.

1. `index(elemento,inicio,final)`
2. `count(elemento)`

Nota: Los métodos para las cadenas de texto y listas modifican directamente al objeto al que se le aplica, y no a una copia de ellos como se ha podido observar en los ejemplos anteriores.

4.3 Diccionarios

Hasta el momento, hemos visto estructuras en **Python** que nos permiten guardar diferentes tipos de datos, estas son las listas y tuplas. Una característica en común entre las listas y tuplas es que sus elementos están ordenados por un índice, sin embargo, a veces no es necesario tener ningún orden con los datos que guardamos, en esos casos, es necesario usar un tipo de estructura de datos que en **Python** es llamada **diccionario**.

Diccionario: Es un tipo de datos que nos permite guardar valores de diferentes tipos (como las listas y tuplas), incluso otros diccionarios. El tipo de datos para los diccionario es `dict`.

[16]En el enlace para los métodos de una lista podrán encontrar que este método tiene argumentos y cómo usarlos.

La diferencia más importante entre las listas y tuplas, con respecto a los diccionarios, es que sus elementos son identificados por una **clave**, dicha clave no tiene necesariamente un orden o relación dentro de las demás claves[17], solo sirve para poder acceder al valor al cual está asociado dentro del diccionario.

La sintaxis para un diccionario es:

```
mi_diccionario = {clave1 : valor1, clave2 : valor2,...,claven : valorn}
```

Nota: Las claves pueden ser de tipo: `int`, `float`, `str` y `tuple`. Por otro lado los valores puede ser del tipo: `int`, `float`, `str`, `list`, `tuple` y `dict`.

Ejemplo:

```
1  mi_diccionario ={"Nombre":"Lazaro", "Edad":23, (2,2):"xD"
       ,33:5,2:[1,2]}
2  print(mi_diccionario)
```

```
» {"Nombre": "Lazaro", "Edad": 23, (2, 2): "xD", 33: 5, 2: [1, 2]}
```

Observación: En el ejemplo anterior, podemos notar que las claves no son necesariamente del mismo tipo (lo mismo sucede con los valores), más aún, el tipo de dato usado para la clave, no es necesariamente el mismo tipo de dato usado para el valor que está asociado.

Para acceder a los valores del diccionario, lo haremos usando la clave asociado al valor que necesitemos, para ello seguiremos la sintaxis siguiente:

```
mi_diccionario[clave].
```

Ejemplo:

```
1  mi_diccionario ={"Nombre":"Lazaro", "Edad":23, (2,2):"xD"
       ,33:5, 2:[1,2]}
2  print(mi_diccionario["Nombre"])
3  print(mi_diccionario[2])
```

```
» "Lazaro"
  [1, 2]
```

Observación: Para acceder al valor `"Lazaro"`, usamos la clave `"Nombre"` (de igual forma para `[1, 2]`, que en este caso su clave es 2).

Una característica que comparten los diccionarios con las listas, es que podemos cambiar sus valores (más precisamente podemos cambiar el valor a una cla-

[17] Esta característica hace que muchas veces los diccionarios sean vistos como estructuras de datos tipo mapa.

ve), la forma de hacerlo es la misma que se hace con las listas, solo que usando la clave, en vez de un índice.

Ejemplo:

```
1  mi_diccionario ={"Nombre":"Lazaro", "Edad":23, (2,2):"xD"
       ,33:5, 2:[1,2]}
2  mi_diccionario[(2,2)] = ":D"
3  print(mi_diccionario)
```

```
» {"Nombre": "Lazaro", "Edad": 23, (2, 2): ":D", 33: 5, 2: [1, 2]}
```

Observación: A la clave `(2,2)`, le cambiamos su valor `"xD"` por `":D"`.

Hacer el proceso anterior genera un efecto muy interesante, si la clave no existe dentro del diccionario, el efecto que resulta es de agregar un nuevo valor dentro del diccionario, al final de la secuencia, con la clave colocada y el valor asignado. Veamos un ejemplo de lo anterior.

Ejemplo:

```
1  mi_diccionario ={"Nombre":"Lazaro", "Edad":23, (2,2):"xD"
       ,33:5, 2:[1,2]}
2  mi_diccionario["Apellido"] = "López"
3  print(mi_diccionario)
```

```
» {"Nombre": "Lazaro", "Edad": 23, (2, 2): ":D", 33: 5, 2: [1, 2], "
    Apellido": "López"}
```

Observación: Como se observa en el ejemplo anterior, la clave `"Apellido"` no existe dentro del diccionario, por lo que **Python** lo agrega al final como nuevo elemento, con la clave `"Apellido"` y el valor asociado `"López"`.

4.3.1 Métodos de los diccionarios

Al igual que las listas y las tuplas, los diccionarios tienen métodos, y la forma de usarlos es la misma. A continuación presentamos los más usados.

1. `update(otro_diccionario)`: Este método agrega al final de la secuencia del diccionario los elementos de `otro_diccionario`.

   Ejemplo:

   ```
   1  diccionario = {"Australia":"Canberra","El Salvador":"San
          Salvador"}
   2  diccionario.update({"Italia":"Roma", "Japón":"Tokio"})
   3  print(diccionario)
   ```

   ```
   » {"Australia": "Canberra", "El Salvador": "San Salvador", "
       Italia": "Roma", "Japón": "Tokio"}
   ```

Observación: Si el diccionario a agregar solo tiene un elemento, entonces el efecto es de agregar un elemento al diccionario, muy parecido al método `append()` de las listas.

2. `keys()`: Devuelve la lista de claves del diccionario.

   Nota: La lista que devuelve este método es del tipo `dict_keys`, así que para convertirla enteramente a una lista debemos usar `list()`.

   Ejemplo:

```
1  diccionario = {"Australia":"Canberra","El Salvador":"San Salvador"}
2  lista_claves = list(diccionario.keys())
3  print(lista_claves)
```

```
» ["Australia", "El Salvador"]
```

3. `values()`: Devuelve la lista de valores del diccionario.

   Nota: La lista que devuelve este método es del tipo `dict_values`, así que para convertirla enteramente a una lista debemos usar `list()`.

   Ejemplo:

```
1  diccionario = {"Australia":"Canberra","El Salvador":"San Salvador"}
2  lista_valores = list(diccionario.values())
3  print(lista_valores)
```

```
» ["Canberra", "San Salvador"]
```

Una lista con el resto de métodos para diccionarios, lo podrán encontrar en el siguiente enlace: `https://www.tutorialspoint.com/python3/python_dictionary.htm`

4.4 Ejercicios

Indicaciones: En cada uno de los siguiente problema debe crearse un programa o programa modular[18] (que funcione como módulo y programa).

1. Escribir funciones para realizar las siguientes conversiones:

   a) De entero decimal a:

   - binario entero sin signo (64 bits de precisión)

[18] Este módulo debe contener todas las funciones auxiliares que se requieran para la creación del programa o programas que se solicitan en cada literal.

- binario entero con signo (63 bits de precisión)
- octal sin signo (64 bits)
- hexadecimal sin signo (64 bits)

b) de decimal no entero a:

- binario entero sin signo (32 bits de precisión entera, 8 bits de precisión no entera)
- binario entero con signo (31 bits de precisión entera, 8 bits de precisión no entera)

c) de binario entero sin signo a:

- octal
- hexadecimal

2. Escribir funciones para:

   a) Generar conjuntos a partir de datos almacenados en una lista.

   b) Determinar la pertenencia de un elemento en un conjunto.

   c) Determinar si un conjunto es subconjunto de otro.

   d) Determinar la unión de dos conjuntos.

   e) Determinar la intersección de dos conjuntos.

   f) Determinar la diferencia entre dos conjuntos.

3. Escribir funciones para:

   a) Leer y formar una matriz A de dimensiones m y n.

   b) Calcular la potencia x de una matriz cuadrada A.

   c) Calcular el determinante de una matriz cuadrada A.

4. Determine si una frase es palíndromo o no. Ignore signos de puntuación. Por ejemplo, «se van sus naves» es palíndromo pues se lee igual de izquierda a derecha que de derecha a izquierda.

5. Recibir los nombres completos de diez compañeros de clase e imprimirlos con formato centrado, y asegurando inicial mayúscula. Intente hacer dos versiones: una con lo métodos de «str» y otra sin ellos.

6. Utilizando cadenas, descomponer un número real con no más de tres espacios decimales en suma de potencias de 10.

7. Usando listas y tuplas, reciba dos listas de datos enteros y luego cree el producto cartesiano de dichas listas.

8. Dado un conjunto de enteros ingresados por el usuario, cree una lista de pares ordenados que corresponda a las siguientes relaciones:

   a) R en A, $R = \{(x,y) : x^2 + y^2 = 10^2\}$

   b) R en A, $R = \{(x,y) : \exists k, x = y \times k\}$

   c) R en A, $R = \{(x,y) : x \equiv y \mod 5\}$

   d) R en A, $R = \{(x,y) : x < y\}$

   e) R en A, $R = \{(x,y) : x + y = 10\}$

9. Para algún par de conjuntos del problema anterior, genere las siguientes relaciones:

   a) R en $A \times A$, $R = \left\{(x,y,z,w) : \frac{x}{y} = \frac{z}{w}\right\}$

10. Para una lista de pares de enteros menores que 50, grafíquelos en el plano cartesiano.

5. FICHEROS

5.1 Matrices

Una matriz en **Python** es simplemente una lista de listas, donde cada sublista es la representación de cada fila de la matriz.

Ejemplo: Consideremos la matriz A de la siguiente manera:

$$A = \begin{pmatrix} 2 & 1 & 2 \\ 3 & 3 & 3 \\ 1 & 3 & 0 \end{pmatrix}$$

La representación en **Python** de A es:

```
1  A = [[2,1,2],[3,3,3],[1,3,0]]
2  print(A)
```

```
[[2, 1, 2], [3, 3, 3], [1, 3, 0]]
```

Observación: Notemos que por cada fila de la matriz A se tiene una sublista. Además en este curso usaremos las letras mayúsculas para denotar variables del tipo matriz.

Todo lo que se puede hacer con lista, se puede hacer con las matrices creadas de la siguiente forma.

5.1.1 Matriz en Numpy.

Hay dos formas de ver matrices en **Numpy**, una es usando el módulo **array**[19] y la otra es usando el módulo **matrix**. Nos ocuparemos del segundo[20], ya que este implementa de manera natural las operaciones de matrices en los operadores usuales (suma, resta y multiplicación).

Nota: Para usar **matrix** en nuestros programas, debemos importar la libreria **Numpy**.

La sintaxis para crear una matriz usando **matrix** es:

```
A = numpy.matrix(lista_de_datos).
```

Ejemplo: Consideremos la matriz A de la siguiente manera:

$$A = \begin{pmatrix} 2 & 1 & 2 \\ 3 & 3 & 3 \\ 1 & 3 & 0 \end{pmatrix}$$

La representación en **Python** de A usando **matrix** es:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4  print(A)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]
```

Observación: Es de resaltar que de esta forma, la presentación en pantalla de la matriz, es más parecida a la forma usual.

5.1.1.1. Atributos de las matrices. Por la forma que hemos creado las matrices, estas guardan información importante para nosotros, estos datos son conocidos

[19] Se deja al estudiante investigar el uso y diferencia entre usar *array* y *matrix*.
[20] Investigar las ventajas y desventajas entre usar *array* y *matrix*.

atributos[21]. La sintaxis para acceder a los atributos de una matriz es:

```
mi_matriz.variable_atributo.
```

Los atributos más importantes que podemos encontrar son:

1. `A.H`: Devuelve la matriz transpuesta conjugada (si sus elementos son números complejos, tendremos la matriz transpuesta conjugada, sino es la matriz transpuesta) de la matriz `A`.

2. `A.T`: Devuelve la matriz transpuesta de la matriz `A`.

   Ejemplo:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4
5  print(A)
6
7  print(A.T)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

matrix([[2, 3, 1],
        [1, 3, 3],
        [2, 3, 0]])
```

3. `A.I`: Devuelve la matriz inversa de la matriz `A` (si tiene).

   Ejemplo:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4
5  print(A)
6
7  print(A.I)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

matrix([[ 3.        , -2.        ,  1.        ],
        [-1.        ,  0.66666667,  0.        ],
        [-2.        ,  1.66666667, -1.        ]])
```

[21] Los atributos son variables creadas con información importante del objeto (esto se estudia en programación II).

4. `A.shape`: Devuelve una tupla con las dimensiones de la matriz `A`.

   Ejemplo:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4
5  print(A)
6
7  print(A.shape)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

(3, 3)
```

5. `A.size`: Devuelve el número de entradas de la matriz `A`.

   Ejemplo:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4
5  print(A)
6
7  print(A.size)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

9
```

5.1.1.2. Operaciones con matrices. Las operaciones se pueden realizar de manera natural con los operadores de suma, resta y multiplicación en las matrices.

Ejemplos:

```
1  import numpy as np
2
3  a = [[2,1,2],[3,3,3],[1,3,0]]
4  A = np.matrix(a)
5  print(A)
6
7  b = [[2,3,4],[0,1,0],[5,6,8]]
8  B = np.matrix(b)
9  print(B)
10
11 print(A+B)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

[[2 3 4]
 [0 1 0]
 [5 6 8]]

matrix([[4, 4, 6],
        [3, 4, 3],
        [6, 9, 8]])
```

```
1   import numpy as np
2
3   a = [[2,1,2],[3,3,3],[1,3,0]]
4   A = np.matrix(a)
5   print(A)
6
7   b = [[2,3,4],[0,1,0],[5,6,8]]
8   B = np.matrix(b)
9   print(B)
10
11  print(A-B)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

[[2 3 4]
 [0 1 0]
 [5 6 8]]

matrix([[ 0, -2, -2],
        [ 3,  2,  3],
        [-4, -3, -8]])
```

```
1   import numpy as np
2
3   a = [[2,1,2],[3,3,3],[1,3,0]]
4   A = np.matrix(a)
5   print(A)
6
7   b = [[2,3,4],[0,1,0],[5,6,8]]
8   B = np.matrix(b)
9   print(B)
10
11  print(A*B)
```

```
[[2 1 2]
 [3 3 3]
 [1 3 0]]

[[2 3 4]
 [0 1 0]
 [5 6 8]]
```

```
» matrix([[14, 19, 24],
          [21, 30, 36],
          [ 2,  6,  4]])
```

5.1.1.3. Métodos de las matrices. Las matrices tienen métodos integrados. A continuación se presentan dos métodos muy usados:

1. `A.trace()`: Devuelve la suma de los elementos de la diagonal de la matriz `A`.

   Ejemplo:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4  print(A)
5
6  print(A.trace())
```

```
» [[2 1 2]
   [3 3 3]
   [1 3 0]]

  matrix([[5]])
```

2. `A.var()`: Devuelve la varianza de los elementos de la matriz `A`.

   Ejemplo:

```
1  import numpy as np
2  a = [[2,1,2],[3,3,3],[1,3,0]]
3  A = np.matrix(a)
4  print(A)
5
6  print(A.var())
```

```
» [[2 1 2]
   [3 3 3]
   [1 3 0]]

  1.1111111111111112
```

Nota: Ambos métodos tienen argumentos opcionales, estos sirven para acotar los elementos, a los que se les desea aplicar el efecto del método.

La documentación completa de atributos y métodos, de las matrices en formato **matrix**, la podrán encontrar en el siguiente enlace: `https://numpy.org/doc/1.18/reference/generated/numpy.matrix.html`

5.2 Archivos .txt

Python nos permite manipular archivos de escritura poco complejas, uno de ellos son los archivos .txt, estos archivos básicos de escritura podemos usarlos en la elaboración de nuestros programas, extrayendo su información y escribiendo en ellos resultados relacionados con la finalidad del programa principal.

A continuación, trataremos algunas funciones que nos permiten crear, abrir y modificar archivos .txt.

Nota: Los archivos creados o usados .txt, deben estar siempre en la misma carpeta de trabajo[22].

1. `open(nombre_archivo,modo_abrir)`: Esta función permite abrir un archivo de escritura, y devuelve un objeto[23] que guarda toda la información en el archivo en formato de tipo `str`.

   Los argumentos de `open()` son:

   - `nombre_archivo`: Debe ser una cadena de texto que indique el nombre del archivo (incluso su extensión).
   - `modo_abrir`: Este argumento nos permite indicarle a **Python** cómo vamos a abrir el archivo, si es crearlo, leerlo o modificarlo. A continuación, presentamos una tabla donde están los distintos modos.

Modo	Descripción
r	El archivo se abre en modo de solo lectura, no se puede escribir (argumento por defecto).
w	Se crea un nuevo archivo y se abre en modo de lectura (si existe un archivo con el mismo nombre, se borra).
a	El archivo se abre en modo de agregar, los datos escritos se agregan al final del archivo.
r+	El archivo se abre para lectura y escritura al mismo tiempo.
b	El archivo se abre en modo binario, para almacenar cualquier cosa que no sea texto.
U	El archivo se abre con soporte a nueva linea universal, cualquier fin de linea ingresada sera como un `\n` en **Python**.

[22] Esto permite mayor comodidad, porque si estuvieran en otra carpeta, tendríamos que especificar la ruta de acceso a dichos archivos.

[23] Este tipo de dato en **Python 3** es una clase, se deja al estudiante investigar sobre las diferencias entre **Python 2** y **Python 3**.

Ejemplo:

```
>>> archivo = open("datos.txt","r")
... Sea abierto el archivo datos.txt en modo de solo lectura ...
```

Observación: Hay un tercer argumento, pero no es opcional, se motiva al estudiante investigar el funcionamiento de este tercer argumento.

2. `read()`: Este método permite leer todo el documento y devuelve una cadena de texto con el contenido del archivo.

   Nota: Este método tiene un argumento del tipo entero, el cual nos permite indicarle cuántos caracteres del archivo queremos leer.

   Ejemplo:

```
>>> archivo = open("datos.txt","r")

>>> contenido = archivo.read()
... En la variable contenido sea guardado toda la información del
    archivo datos.txt, está variable es del tipo str ...
```

   Observación: Si ya se ha leído todo el contenido del archivo, aplicar una vez más este método, hará que devuelva una cadena de texto vacía.

3. `write(texto)`: Este método permite escribir el contenido de una cadena de texto al archivo. El argumento `texto` debe ser una cadena de texto con la información que deseamos agregar al archivo.

   Ejemplo:

```
>>> archivo = open("datos.txt","a")

>>> texto = "\nHola de nuevo"
>>> archivo.write(texto)
... Se agrega la cadena de texto "Hola de nuevo"
    al final del archivo datos.txt ...
```

   Observación: La instrucción `\n` le dice a **Python** que haga un salto de linea.

   Nota: Si se abre el archivo en el modo `"r+"`, debe usarse el método `seek(0,2)` antes de usar `write()` (ejemplo: `archivo.seek(0,2)`), este método con esos argumentos le indica a **Python** que escriba al final del documento.

4. `close()`: Este método permite cerrar la manipulación del archivo. No es posible escribir ni leer en un archivo cerrado.

   Ejemplo:

```
>>> archivo = open("datos.txt","a")
>>> archivo.close()
... Se ha cerrado el archivo ...
```

Hay más métodos para trabajar con archivos, el resto los podrán encontrar en el siguiente enlace: `https://python-para-impacientes.blogspot.com/2014/02/operaciones-con-archivos.html`

5.2.1 Atributos de los archivos.

Los siguientes atributos tienen información útil de una variable, que guarda el contenido de un documento de escritura.

1. `closed`: Devuelve `False`, si el documento está cerrado, `True`, si está abierto.

   Ejemplos:

```
1  archivo = open("datos.txt","a")
2
3  print(archivo.closed)
```

```
» False
```

```
1  archivo = open("datos.txt","a")
2  archivo.close()
3
4  print(archivo.closed)
```

```
» True
```

2. `mode`: Devuelve el modo en que fue abierto el documento.

   Ejemplo:

```
1  archivo = open("datos.txt","a")
2
3  print(archivo.mode)
```

```
» "a"
```

3. `name`: Devuelve el nombre del documento abierto.

Ejemplo:

```
1       archivo = open("datos.txt","a")
2
3       print(archivo.name)
```

```
»    "datos.txt"
```

5.3 Archivos .csv

Un archivo .csv puede ser tratado en **Python** de una manera bastante parecida al de los archivos .txt, sin embargo, en este curso veremos cómo usar **Pandas** para trabajar archivos .csv. Recordemos que para usar **Pandas** hay que importarlo, además de abreviarlo como `pd` para mayor comodidad.

5.3.1 Métodos de Pandas.

Para manipular la información guardada en un archivo .csv, debemos usar ciertos métodos de la librería de **Pandas**, a continuación mostramos los más usados:

1. `read_csv(nombre_archivo,name,header)`, este método permite leer archivos .csv y guardar está información en una variable en forma de matriz.

   Los parámetros significan:

   - `nombre_archivo`: Debe ser una cadena de texto que tenga el nombre del archivo .csv a abrir (debe escribirse con la extensión).
   - `names`: Este argumento es opcional y debe ser una lista que contenga los encabezados que queramos colocar a cada columna.
   - `header`: Este argumento es opcional, debe ser un entero, que significa desde cuál fila será leído el documento (la primera fila leida la toma como los encabezados de la tabla). Si deseamos que todo el contenido sean valores a usar, debemos colocar `header=None`.

Ejemplos:

```
1       import pandas as pd
2
3       datos = pd.read_csv("datos.csv")
4
5       print(datos)
```

```
»      Nombres  Notas
   0    Jorge      7
   1   Carlos      6
```

```
2   Sofia     10
```

```
import pandas as pd

datos = pd.read_csv("datos.csv", names=["nombres","notas"])

print(datos)
```

```
   nombres  notas
0  Nombres  Notas
1    Jorge      7
2   Carlos      6
3    Sofia     10
```

```
import pandas as pd
datos = pd.read_csv("datos.csv", header = 2)

print(datos)
```

```
    Carlos   6
0   Sofia   10
```

```
import pandas as pd
datos = pd.read_csv("datos.csv", header = None)

print(datos)
```

```
         0       1
0  Nombres   Notas
1    Jorge       7
2   Carlos       6
3    Sofia      10
```

2. `DataFrame(diccionario)`: Este método de **Pandas** crea un DataFrame de **Pandas**, el argumento debe ser un diccionario, y cada elemento representa una columna de la tabla, además las claves serán tomadas como los encabezados de las columnas.

   Ejemplo:

```
import pandas as pd
nuevos_datos = pd.DataFrame({"Nombres":["Juan Carlos", "Policarpo"],"Notas":[7,5]})

print(nuevos_datos)
```

```
      Nombres  Notas
0  Juan Carlos      7
1    Policarpo      5
```

5.3.1.1. Métodos de los DataFrame de Pandas. Una vez leído o creado un archivo DataFrame de **Pandas**, podremos acceder a la información de manera similar a una matriz, más aún, podremos escribir sobre un archivo .csv o crear un nuevo archivo .csv. A continuación, mostraremos algunos atributos más usados:

1. `to_csv(nombre_archivo,mode,index,header)`: Este método permite escribir sobre un archivo .csv, la información que se encuentra en el DataFrame.

   - `nombre_archivo`: Este argumento debe ser una cadena de texto, y debe contener el nombre completo del archivo.csv, si no existe tal archivo se crea de manera automática.
   - `mode`: Este argumento es igual que el de la función `open()` (es opcional).
   - `index`: Este argumento es booleano, y si es `True`, agrega la numeración de cada fila, y si es `False`, no agrega la numeración de las filas (es opcional).
   - `header`: Este argumento es igual que el de `read_csv()` (es opcional).

   Ejemplos: Para crear un archivo .csv nuevo, los valores de los argumentos opcionales deben ser.

```
>>> import pandas as pd
>>> nuevos_datos.to_csv("datos3.csv")

... En este caso se agrega la numeración de las filas ...

>>> import pandas as pd

>>> nuevos_datos.to_csv("datos3.csv",index=False)
... En este caso no se agrega la numeración de las filas ...
```

   Ejemplo: Para escribir en un archivo .csv (ya existente), los valores de los argumentos opcionales deben ser.

```
>>> import pandas as pd
>>> nuevos_datos.to_csv("datos.csv",mode="a",index=False,header
    =False)

... Agrega al final toda la información de nuevos_datos
    al archivo llamado datos.csv ...
```

Observación: El modo `"a"`[24] indica que agregue al final[25] del archivo la información. Por otro lado, `index=False` y `header=False`, le indican a **Python** que no agregue, ni numeración ni encabezados.

2. `iloc[]`: Este método permite acceder a uno o varios elementos guardados en el DataFrame de **Pandas** (como si fuera una matriz). Dependiendo del argumento que coloquemos así podremos obtener cierta información. A continuación veamos algunos de ellos:

   - Si se ingresa solo un entero positivo o cero, devuelve la fila correspondiente al entero ingresado.
   - Si se ingresan dos enteros positivos de la siguiente forma `a:b` (`a` puede ser cero), devolverá un DataFrame con los mismos encabezados del original, pero con las filas desde `a` hasta `b-1`.
   - Si se ingresan dos enteros positivos de la siguiente forma `a,b` (ambos pueden ser cero), devolverá el elemento en la fila `a` y columna `b` (los encabezados no se cuentan como datos).

   Ejemplo:

```
1  import pandas as pd
2
3  datos = pd.read_csv("datos.csv")
4  print(datos)
5
6  print(datos.iloc[1])
7
8  lista = list(datos.iloc[1])
9  print(lista)
```

```
  Nombres  Notas
0   Jorge      7
1  Carlos      6
2   Sofia     10

Nombres    Carlos
Notas           6
Name: 1, dtype: object

["Carlos", 6]
```

Observación: Como podemos ver, hemos extraído la fila dos (que es la numerada como 1) del DataFrame, y esta es presentada como columna, in-

[24] El modo *a* deja preparado el curso de escritura en la siguiente línea.

[25] Normalmente cuando se usa Excel para crear un archivo .csv, automáticamente deja el curso de escritura en la siguiente linea, si por alguna razón esto no es así, agregue manualmente el curso de escritura en la siguiente línea.

dicando a qué encabezado le corresponde cada valor. Además vemos que podemos convertir la fila seleccionada en una lista.

Ejemplo:

```
1  import pandas as pd
2
3  datos = pd.read_csv("datos.csv")
4  print(datos)
5
6  print(datos.iloc[1:3])
7
8  print(datos.iloc[2,1])
```

```
  Nombres  Notas
0   Jorge      7
1  Carlos      6
2   Sofia     10

  Nombres  Notas
1  Carlos      6
2   Sofia     10

10
```

3. `sort_values(by,ascending)`: Este método devuelve un DataFrame de **Pandas** con la información ordena del DataFrame original. El orden es de acuerdo a los valores de la columna indicada en `by`, para seleccionar la columna debemos colocar el encabezado, correspondiente a la columna a ordenar (si el encabezado es texto, debe colocarse como cadena de texto, para el valor del argumento).

   El argumento `ascending` es opcional. Si `ascending = True`, entonces se ordenan de forma ascendente, sino `ascending = False` se ordenan de forma descendente.

   Ejemplo:

```
1  import pandas as pd
2
3  datos = pd.read_csv("datos.csv")
4  print(datos)
5
6  print(datos.sort_values(by = "Notas"))
```

```
  Nombres  Notas
0   Jorge      7
1  Carlos      6
2   Sofia     10

  Nombres  Notas
1  Carlos      6
0   Jorge      7
```

```
2   Sofia     10
```

Observación: Notemos que si no se especifica `ascending = True`, por defecto toma ese valor para el parámetro opcional.

Ejemplo:

```
import pandas as pd

datos = pd.read_csv("datos.csv")
print(datos)

print(datos.sort_values(by = "Notas", ascending = False))
```

```
  Nombres  Notas
0   Jorge      7
1  Carlos      6
2   Sofia     10

  Nombres  Notas
2   Sofia     10
0   Jorge      7
1  Carlos      6
```

Además de los métodos descritos anteriormente, existen otras formas de acceder a la información de nuestros DataFrame, por ejemplo, si queremos acceder a una columna de nuestra base de datos, lo podemos hacer de la siguiente manera `mis_datos[Encabezado]`.

Ejemplo:

```
import pandas as pd

datos = pd.read_csv("datos.csv")
print(datos)

print(datos["Nombres"])

lista = list(datos["Nombres"])
print(lista)
```

```
  Nombres  Notas
0   Jorge      7
1  Carlos      6
2   Sofia     10

0     Jorge
1    Carlos
2     Sofia
Name: Nombres, dtype: object

["Jorge", "Carlos", "Sofia"]
```

Observación: Lo que hemos hecho en el ejemplo anterior, es poder extraer la

columna `Nombres`. Además el encabezado al ser cadena de texto, se ingresa de esa forma. También podemos convertir esta columna en una lista.

Finalmente, podemos extraer partes del DataFrame, de una manera parecida al ejemplo anterior (en este caso usando la numeración de la filas).

Ejemplo:

```
1  import pandas as pd
2
3  datos = pd.read_csv("datos.csv")
4  print(datos)
5
6  print(datos[1:3])
```

```
   Nombres  Notas
0   Jorge       7
1  Carlos       6
2   Sofia      10

   Nombres  Notas
1  Carlos       6
2   Sofia      10
```

Observación: En el ejemplo anterior, se he extraido desde la fila 1 hasta 2, la sintaxis en este caso es `mis_datos[a:b]`, donde se extrae desde la fila numerada con `a` hasta la fila numerada con `b-1`. Además note que se devuelve otro objeto del tipo DataFrame de **Pandas**.

La documentación completa[26] de la librería **Pandas**, la podrán encontrar en el siguiente enlace: `https://pandas.pydata.org/pandas-docs/stable/reference/io.html`

En este documento se usan archivos .csv con **Pandas**, sin embargo existe una librería destinada al manejo de estos archivos, la cual es **csv**. En el siguiente enlace podrán ver cómo usar ciertos aspectos básicos de esa librería: `https://code.tutsplus.com/es/tutorials/how-to-read-and-write-csv-files-in-python--cms-29907`

5.4 Laboratorio - 1

Indicaciones:

- Debe crearse un programa o programa modular[27] (que funcione como módulo y programa).

[26] Los métodos presentados, solo abarcan los argumentos opciones más utilizados, sin embargo, como podrán ver en el enlace, cada método tiene más argumentos opciones, estos permiten manipular otros aspectos de los datos del DataFrame.

[27] Este módulo debe contener todas las funciones auxiliares que se requieran para la creación del programa o programas que se necesiten en cada numeral.

- Las funciones auxiliares deben tener documentación, especificando qué variable está devolviendo, y de qué tipo son, y lo mismo para los argumentos.
- El programa principal y funciones auxiliares deben tener una documentación apropiada, explicando procesos no tan obvios.
- Crear un archivo .csv que contenga la tabla de datos iniciales que se muestra en cada problema (esta pueden hacerla usando Excel).
- El programa principal debe crear un archivo .txt respondiendo las siguientes interrogantes:
    1. ¿Cuántas clases recomendaría?
    2. ¿Qué intervalo de clase sugeriría?
    3. ¿Qué límite inferior recomendaría para la primera clase?
    4. ¿Dónde tienden a acumularse los datos?
- El programa principal debe organizar los datos en una distribución de frecuencia y distribución de frecuencia relativa. Ambas tablas deben ser presentadas en pantalla y generar un nuevo archivo .csv que las contenga (no debe usarse Excel en esta parte).

Problemas:

1. El director de asuntos escolares de la Universidad de Nápoles Federico II tiene 16 solicitudes de la inscripción para el año próximo. Las puntaciones en el examen de conocimientos de los solicitantes son:

27	27	27	28	27	25	25	28
26	28	26	28	31	30	26	26

2. El Pollo Campestre tiene varias sucursales en el área metropolitana. El número de cambios de aceite en la sucursal de la 29 Calle Poniente en los pasados 20 días fue:

65	98	55	62	79	59	51	90	72	56
70	62	66	80	94	79	63	73	71	85

3. Al gerente local de Urbana pizza le interesa el número de veces que un cliente visita su restaurante durante un periodo de dos semanas. Las respuestas de 51 clientes fueron:

5	3	3	1	4	4	5	6	4	2	5	6	6	7	1
1	14	1	2	4	4	4	5	6	3	5	3	4	5	6
8	4	7	6	5	9	11	3	12	4	7	6	5	15	1
1	10	8	9	2	2									

4. Cleveland, una agencia de viajes ofrece tarifas especiales en ciertos creceros para personas de la tercera edad. El presidente de Cleveland quiere información adicional sobre las edades de las personas que toman cruceros. Una muestra aleatoria de 40 clientes que tomaron un crucero el año pasado revela estas edades.

77	18	63	84	38	54	50	59	54	56	36	26	50	34	44
41	58	58	53	51	62	43	52	53	63	62	62	65	61	52
60	60	45	66	83	71	63	58	51	71					

5.5 Laboratorio - 2

Indicaciones:

- Debe crearse un programa o programa modular[28] (que funcione como módulo y programa).
- Las funciones auxiliares deben tener documentación, especificando qué variable está devolviendo, y de qué tipo son, y lo mismo para los argumentos.
- El programa principal y funciones auxiliares deben tener una documentación apropiada, explicando procesos no tan obvios.
- El programa principal debe crear un archivo .txt que contenga la siguiente información.

    1. Indicar la elección del usuario.
    2. Media, mediana y moda de los datos.
    3. Indicar los cuartiles, deciles o percentiles, de acuerdo a lo elegido por el usuario.

- El programa principal debe desplegar un menú en la pantalla, donde le permita elegir al usuario, si quiere calcular **cuartiles**, **deciles** o **percentiles**. Además, el programa principal debe generar el **diagrama de cajas** de los datos proporcionados y guardarlo.

Problemas:

1. El director de asuntos escolares de la Universidad de Nápoles Federico II tiene 16 solicitudes de la inscripción para el año próximo. Las puntaciones en el examen de conocimientos de los solicitantes son:

27	27	27	28	27	25	25	28
26	28	26	28	31	30	26	26

2. El Pollo Campestre tiene varias sucursales en el área metropolitana. El número de cambios de aceite en la sucursal de la 29 Calle Poniente en los pasados 20 días fue:

65	98	55	62	79	59	51	90	72	56
70	62	66	80	94	79	63	73	71	85

[28] Este módulo debe contener todas las funciones auxiliares que se requieran para la creación del programa o programas que se necesiten en cada numeral.

3. Al gerente local de Urbana pizza le interesa el número de veces que un cliente visita su restaurante durante un periodo de dos semanas. Las respuestas de 51 clientes fueron:

5	3	3	1	4	4	5	6	4	2	5	6	6	7	1
1	14	1	2	4	4	4	5	6	3	5	3	4	5	6
8	4	7	6	5	9	11	3	12	4	7	6	5	15	1
1	10	8	9	2	2									

4. Cleveland, una agencia de viajes ofrece tarifas especiales en ciertos cruceros para personas de la tercera edad. El presidente de Cleveland quiere información adicional sobre las edades de las personas que toman cruceros. Una muestra aleatoria de 40 clientes que tomaron un crucero el año pasado revela estas edades.

77	18	63	84	38	54	50	59	54	56	36	26	50	34	44
41	58	58	53	51	62	43	52	53	63	62	62	65	61	52
60	60	45	66	83	71	63	58	51	71					

6. Recursividad

6.1 Funciones recursivas

Las funciones recursivas son funciones que se llaman a sí mismas durante su propia ejecución. Estas funciones se comportan de alguna manera como los ciclos, ya que estas repiten una y otra vez sus instrucciones hasta finalizar el proceso.

Como mínimo una función recursiva tiene dos partes, las cuales son:

1. **Caso base o de fin de recursión:** Es un caso donde el problema puede resolverse sin tener que hacer uso de una nueva llamada a sí mismo. Evita la continuación indefinida de las partes recursivas.
2. **Parte recursiva:** Relaciona el resultado del algoritmo con resultados de casos más simples. Se hacen nuevas llamadas a la función, hasta llegar al caso base.

Observación: Debemos conocer bien el momento en que la función deberá dejar de llamarse a sí mismas o tendremos una función recursiva infinita.

Veamos a continuación dos ejemplos de funciones recursivas, una donde no retorne nada y otra donde sí retorne un valor.

Ejemplo:

```
1       def tabla2(n):
2           if n>0:
3               print("2 *",n," = ",2*n)
4               tabla2(n-1)
5           else:
6               print("2 *",n," = ",2*n)
```

```
7
8       print(tabla2(5))
```

```
2 * 5 = 10
2 * 4 = 8
2 * 3 = 6
2 * 2 = 4
2 * 1 = 2
2 * 0 = 0
```

Observación: El ejemplo anterior nos muestra una función que despliega en pantalla la tabla del 2 hasta un cierto número n. En esta función hemos sustituido un ciclo `for` por la acción de llamarse a sí misma a la función, para lograr el efecto iterativo del ciclo y poder diseñar en pantalla la tabla del 2.

Ejemplo:

```
1       def factorial(n):
2           if n>0:
3               return n*factorial(n-1)
4           else:
5               return 1
6
7       print(factorial(0))
8
9       print(factorial(1))
10
11      print(factorial(5))
```

```
1

1

120
```

Observación: En el caso anterior tenemos una función que retorna el factorial de un número entero positivo n o cero, nótese que una vez más no hemos implementado un ciclo para encontrar el factorial (como podría pensarse en un principio), sino que sea ha hecho que la función se llame a sí misma, para crear el efecto de multiplicar n con todos los número enteros positivos anteriores a él.

En ambos ejemplos, podemos notar que hemos dejado claro que cuando $n = 0$ la función debe parar de llamarse, provocando que la acción iterativa de llamarse tenga un fin. Debe estar claro bajo qué instancia la función dejará de llamarse, porque si hay ambigüedad, podría resultar en una función iterativa infinita (nunca pararía).

6.2 Tipos de recursión

Dependiendo de la forma que las funciones recursivas pueden relacionarse entre sí, podemos tener diversos tipos de recursión. A continuación, presentaremos estos tipos de recursión.

1. **Recursividad simple:** Son las funciones que sólo hacen una llamada recursiva (a sí mismas). Se puede transformar con facilidad en algoritmos iterativos (ciclos).

   Ejemplo:

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

print(factorial(5))
```

```
120
```

2. **Recursividad múltiple:** Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de hacer de forma iterativa.

   Ejemplo:

```
def fibonacci(n):
    if n<=1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)

print(fibonacci(5))
```

```
8
```

3. **Recursividad anidada:** Una función recursiva es anidada cuando entre los parámetros que se pasan a la función se incluye una invocación a la misma.

   Ejemplo:[29]

```
def Ackermann(m,n):
    if m==0:
        return n+1
    elif m>0 and n==0:
        return Ackermann(m-1,1)
    else:
        return Ackermann(m-1,Ackermann(m,n-1))

print(Ackermann(3,2))
```

```
29
```

4. **Recursividad cruzada o indirecta:** Son algoritmos donde una función provoca una llamada a sí misma de forma indirecta, a través de otras funciones.

[29] Esta es la implementación en **Python** de la función de Ackermann, en el siguiente enlace, podrán encontrar un poco de información sobre esta función: `https://es.wikipedia.org/wiki/Funci%C3%B3n_de_Ackermann`

Ejemplo:

```
1   def par(n):
2       if n==0:
3           return True
4       else:
5           return impar(n-1)
6
7   def impar(n):
8       if n==0:
9           return False
10      else:
11          return par(n-1)
12
13  print(par(5))
14
15  print(impar(5))
```

```
False

True
```

6.3 Algoritmos de ordenamiento

Una tarea muy habitual en programación, es la de ordenar ciertos datos bajo cierto criterio[30], por esta razón, han surgido ciertas técnicas, que permiten ordenar datos de manera eficiente. A estas técnicas se les conoce como «algoritmos de ordenamiento».

A continuación, presentamos los principales algoritmos de ordenamiento:

- Ordenamiento de Batcher.
- Ordenamiento de burbuja.
- Ordenamiento de burbuja bidireccional.
- Ordenamiento por inserción.
- Ordenamiento por montículos.
- Ordenamiento de peine.
- Ordenamiento rápido.
- Ordenamiento por selección.
- Ordenamiento de Shell.

6.4 Ejercicios

Indicaciones: En cada uno de los siguiente problema debe crearse un programa o programa modular[31] (que funcione como módulo y programa).

[30] Las relaciones de orden más usadas son **el orden numérico** y **el orden lexicográfico**. Estas relaciones de orden son muy útiles en materias como **topología**.

[31] Este módulo debe contener todas las funciones auxiliares que se requieran para la creación del programa o programas que se solicitan en cada literal.

1. Para cada literal, hacer un programa que dada un número entero positivo calcule el n-ésimo termino de la sucesión aritmética, definida por los primeros términos de la progresión.

   a) $42, 39, 36, \ldots$

   b) $3, 8, 13, \ldots$

   c) $1/2, 5/8, 3/4, \ldots$

2. Hacer una función que reciba el primer termino de la sucesión aritmética, la diferencia d y un entero positivo n, y devuelva la suma de los primeros n términos de la sucesión.

3. Hacer un programa que dada un número entero positivo calcule el n-ésimo termino de la sucesión geométrica, definida por los primeros términos de la progresión. Los cuales son: $3, 1, 1/3, 1/9, \ldots$

4. Hacer una función que reciba el primer termino de la sucesión geométrica, la razón r y un entero positivo n, y devuelva la suma de los primeros n términos de la sucesión.

5. Hacer una función que reciba un entero $n \geq 0$, y devuelva el n-ésimo termino de la sucesión definida como:

$$a_n = 5a_{n-1} + 6a_{n-2}, n \geq 2, a_0 = 1, a_1 = 3$$

6. Hacer un programa que dado un entero $n \geq 0$, de vuelva el n-ésimo termino del siguiente sistema de relación de recurrencia.

$$\begin{cases} a_{n+1} = 2a_n - b_n + 2 \\ b_{n+1} = -a_n + 2b_n - 1 \end{cases} \quad n \geq 0, a_0 = 0, b_0 = 1$$

Nota: En todos los ejercicios solo usar funciones recursivas.

7. PROGRAMACIÓN FUNCIONAL

7.1 Función lambda

De manera informal una función **lambda** es una pequeña función, en el sentido que las instrucciones dentro de la función lambda son sencillas y cortas (a veces llamadas subrutinas). Las funciones lambda a menudo son usadas como argumentos que pasan a ser funciones de **orden superior** [32], o se usan para construir el resultado de una función de orden superior que necesita devolver.

[32] En matemáticas y ciencias de la computación, funciones de orden superior son funciones que cumplen al menos una de las siguientes condiciones:

1. Tomar una o más funciones como entrada.

La comodidad de usar este tipo de funciones es que son funciones que se declaran en una línea. Los estudiantes encontrarán la primera utilidad de este tipo de funciones en los cursos de análisis numérico.

Sintaxis de las funciones lambda es:

```
nombre_funcion = lambda arg1, arg2, arg3, ...: instrucciones
```

Ejemplos:

```
1  por2 = lambda numero: numero*2
2  print(por2(3))
```

```
6
```

```
1  f = lambda x,y: x+y
2  print(f(2,3))
```

```
5
```

Nota: Las funciones lambda por defecto devuelven un resultado, por ende no es necesario escribir `return` cuando se usa Python, sin embargo, hay lenguajes de programación donde sí es necesario especificar qué se devolverá.

7.2 `filter, map, reduce.`

En esta última sección, estudiaremos tres tipos de funciones, que toman como parámetro otras funciones. Estas permiten expandir la utilidad de las funciones existentes, o las que se puedan crear.

7.2.1 `filter.`

Esta función toma como argumentos una función booleana[33] y una lista, y devuelve otra lista, con los valores de la lista ingresada que cumplen la función booleana. La lista que devuelve la función Filter es del tipo `filter`, así que debe usarse `list()`, para que tengamos una lista propiamente.

La sintaxis es: `filter(mi_funcion, mi_lista)`

Nota: Debe ingresarse solamente el nombre de la función booleana[34], sin paréntesis y sin argumentos.

2. Devolver una función como salida.

En matemática estas funciones se llaman operadores o funcionales. Ejemplos: la derivada y la antiderivada.

[33] El término **función booleana** se refiere que devuelve **True** o **False**.

[34] Si la función no es booleana, ¿cuál sería el resulta?

Ejemplo:

```
def multiplo_2(n):
    if n%2==0:
        return True
    else:
        return False

x = [1,2,3,4,5,6,7,8,9,10]

print(list(filter(multiplo_2,x)))
```

```
[2, 4, 6, 8, 10]
```

Observación: La función `multiplo_2(n)`, devuelve `True`, si `n` es par, y `False`, si `n` es impar. Además, para la función `filter`, solo se ingresó el nombre de la función booleana, que en este caso es `multiplo_2`.

7.2.2 `map`.

Esta función toma como argumentos una función, y una o varias listas, y devuelve otra lista con los valores que la función devuelve por cada elemento de la lista ingresada. La lista que devuelve la función Map es del tipo `map`, así que debe usarse `list()` para que tengamos una lista propiamente.

La sintaxis es: `map(mi_funcion, lista_1, lista_2, ..., lista_n)`

Nota: Debe ingresarse solamente el nombre de la función, sin paréntesis y sin argumentos. Además, debe ponerse tantas listas como argumentos tenga la función.

Ejemplos:

```
def multiplicar_2(n):
    return n*2

x = [1,2,3,4,5,6,7,8,9,10]

print(list(map(multiplicar_2,x)))
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
x = [1,2,3,4,5,6,7,8,9,10]

print(list(map(lambda n: n*2,x)))
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Observación: En ambos ejemplos, `map()` recorre la lista `x`, y devuelve una lista, con los valores de `x` evaluados en la función `multiplicar_2` y `lambda`.

7.2.3 `reduce`.

Esta función toma como argumentos una función y una lista, y devuelve un valor, el cual es el resultado de aplicar iteradamente los valores de la lista a la función. Al comienzo toma los primeros dos elementos de la lista ingresada, y evalúa en la función, usa este resultado como valor para el primer argumento de la función, y toma el siguiente valor de la lista para el valor del segundo argumento. Este último paso se aplica iteradamente hasta agotar la lista.

La sintaxis es: `reduce(mi_funcion, lista)`

Nota: Debe ingresarse solamente el nombre de la función, sin paréntesis y sin argumentos. Además, el efecto de Reduce sobre los elementos de la lista es de agrupamiento, por ejemplo, si queremos sumar los números $1, 2, 3, 4$ y 5, Reduce aplicaría la función suma de la siguiente manera: $((((1+2)+3)+4)+5) = (((3+3)+4)+5) = ((6+4)+5) = (10+5) = 15$.

Ejemplos:

```
1  from functools import reduce
2
3  def sumar(n,m):
4      return n+m
5
6  x = [1,2,3,4,5,6,7,8,9,10]
7
8  print(reduce(sumar,x))
```

```
» 55
```

```
1  x = [1,2,3,4,5,6,7,8,9,10]
2
3  print(reduce(lambda n,m:n+m,x))
```

```
» 55
```

Observación: Para poder usar Reduce, es necesario importarlo de la librería functools. La función debe poder recibir dos valores (no ser una función de un argumento). Además, si la función tiene más de dos argumentos, el resto de argumentos deben tener valores predeterminados.

7.3 Ejercicios

Indicaciones: En cada uno de los siguiente problema debe crearse un programa o programa modular[35] (que funcione como módulo y programa).

1. Para cada literal, hacer un programa que dado un entero positivo, calcule la suma

[35] Este módulo debe contener todas las funciones auxiliares que se requieran para la creación del programa o programas que se solicitan en cada literal.

parcial hasta el termino n-ésimo.

a) $\sum_{k=1}^{n}(4k-3)$

b) $\sum_{k=1}^{n}\frac{1}{2^k}$

c) $\sum_{k=1}^{n}\frac{3k-6}{2}$

2. Para cada literal, hacer un programa que dado un entero positivo, calcule el producto hasta el termino n-ésimo.

a) $\prod_{k=2}^{n}\frac{k}{k-1}$

b) $\prod_{k=0}^{n}(k+1)^2$

c) $\prod_{k=0}^{n}\frac{(-1)^k}{2^k+1}$

3. Hacer un programa verifique si un carácter es alfanumérico (es decir, minúsculas, mayúsculas o numérico) de una lista.
4. Hacer un programa que decida en una lista de animales y vegetales qué se puede producir con ellos. Por ejemplo de una vaca podemos obtener leche, del maíz tenemos tortillas. Haga que su programa pueda decidir entre tres animales y tres vegetales.
5. Hacer un programa que decida por el nombre de una comida si esté es de origen vegetal o animal. Su programa debe poder decidir entre tres comidas de origen animal (por ejemplo hamburguesa) y tres de origen vegetal (por ejemplo carne de soya).
6. Hacer un programa que dada una lista de cadenas de texto, devuelva la lista pero sin palabras que tengan menos de 4 letras.
7. Hacer un programa que dada dos lista de números, haga el producto cartesiano y devuelva una lista con pares ordenados donde la segunda coordenada es menor a la primera.

Nota: En todos los ejercicios debe usarse `filter`, `map` y `reduce`.