
Algoritmos y Programación I

Curso de verano 2018



Índice

Algoritmo.....	4
Programación	5
Tipos de lenguajes.....	5
Lenguajes de alto nivel y de bajo nivel.....	5
Lenguajes compilados e interpretados	6
Programación estructurada	7
Pseudocódigo	8
Tipos de estructuras de los algoritmos	8
Estructura secuencial	8
Diagramas de flujo	9
Estructuras selectivas o condicionales	10
Estructuras repetitivas o iterativas	11
Python	13
Características	13
Comenzando	14
Variables.....	15
Nombres de variables	16
Tipos de variables.....	17
Otros tipos.....	18
Constantes.....	19
Operadores.....	20
Aritméticos	20
De asignación	20
Comparación	21
Lógicos.....	21
Tablas de valores de verdad.....	21
Estructuras selectivas.....	22
Estructuras repetitivas	24
Funciones	25
Comentarios	26
Parámetros.....	27

Devolución o retorno de una función	27
Pasaje por referencia y por valor	28
Retornos múltiples	29
Módulos	30
Programa principal (main).....	30
Módulo math.....	31
Módulo random	32
Cadenas de caracteres (strings)	36
Tuplas y Listas.....	37
Recorriendo una lista	42
Slices – sublistas	42
Listas y strings	43
Matrices.....	45
Tablas (vectores de registros)	46
Funciones lambda	47
Diccionarios	48
Ordenamientos en diccionarios	50
Estructuras más complejas con diccionarios.....	51
Un poco de maquillaje	53

Algoritmo

Según el DLE: conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

Escribimos otra vez la definición para destacar algunos conceptos:

Conjunto **ordenado** y **finito** de **operaciones** que permite hallar la solución de un **problema**.

- Operaciones. Por este término entendemos a cualquier instrucción unitaria gobernada por un verbo principal, como podría ser: planchar, sacar un tornillo, poner en el horno, imprimir, sumar, etc.
- Problema. El algoritmo tiene una finalidad: resolver un problema. Sin un objetivo que puede ser muy acotado o muy amplio no tenemos un algoritmo.
- Ordenado. Las operaciones tienen que tener un orden determinado. Si el objetivo es cambiar una rueda de un automóvil, no lo puedo resolver si en primer lugar trato de sacar la rueda y en segundo lugar aflojar las tuercas que la sostienen.
- Finito. Significa que las operaciones tienen que tener un fin. Un algoritmo para

obtener la suma de la siguiente serie $\sum_{n=0}^{\infty} \frac{1}{2^n}$ no podría ser:

1. Suma vale 0.
2. Reemplazar n por su valor actual (la primera vez vale 0), y calcular 2^n .
3. Luego hacer $1 /$ el resultado obtenido en el paso 2.
4. Guardar en suma el valor que tenía más el resultado del paso 3.
5. Incrementar n en 1.
6. Repetir los pasos 2 a 5 infinitamente.

Hasta el paso 5 cumple la definición de algoritmo, pero en el paso 6 se indica que las operaciones a realizar son infinitas.

Ejemplos de algoritmos

- Instrucciones para cambiar una rueda
- Recetas de cocina
- Instrucciones para darse de alta en la AFIP
- Instrucciones para grabar una película

Programación

Según el DLE: acción y efecto de programar.

Programar: elaborar programas para su empleo en computadoras.

Programa: serie ordenada de operaciones necesarias para llevar a cabo un proyecto.

Proyecto: idear, trazar o proponer el plan y los medios para la ejecución de algo.

La palabra “proyecto” la vamos a utilizar bastante, en especial cuando tengamos que realizar aplicaciones extensas y que resuelvan problemas complejos.

Tipos de lenguajes

Lenguajes de alto nivel y de bajo nivel

El idioma que una computadora comprende es el lenguaje binario, es decir, unos y ceros. ¿Por qué sucede esto? Sencillamente porque le es sencillo detectar dos intensidades bien marcadas de corriente. Una intensidad más alta que traducirá como un uno, y otra más baja que interpretará como un cero.

Si quisiéramos, por ejemplo, que interprete todos los caracteres de nuestro lenguaje de manera directa, tendríamos que distinguir entre cada una de las letras (26 símbolos en el alfabeto latino) pero tenemos mayúsculas y minúsculas, es decir 26×2 , además de las que puedan llevar acento. A esto debemos agregarle los dígitos (10 símbolos más), y los espacios, paréntesis, corchetes y llaves. Estamos hablando de unos 80 símbolos que deberían interpretarse con 80 niveles de corrientes distintas. Esto, para los electrónicos, sería muy engorroso de resolver. Además de aumentar de manera considerable la posibilidad de errores al no distinguir dos impulsos diferentes de corriente. Hay que tener en cuenta que la diferencia entre dos símbolos contiguos sería muy pequeña y una leve perturbación podría convertir un símbolo en otro.

Entonces, el lenguaje que una máquina comprende es la combinación de unos y ceros, como para nosotros son las palabras. Por ejemplo: 10001011 00010101 11110001 etc.

Nótese que estos unos y ceros se agruparon, no de forma inocente, en tiras de 8 valores. Cada una de estas “palabras” representaría un byte. El byte es la unidad de información, como lo es la célula para la biología, el byte lo es para la informática. A cada uno de los valores los llamamos bits. El byte está compuesto por 8 bits. Las potencias de 2 facilitan

la arquitectura de las computadoras. Por este motivo los sistemas operativos son de 32 bits (4 bytes), 64 bits (8 bytes), etc.

Haciendo un paralelismo, el byte para un ser humano sería una palabra y el bit, una letra. Sin embargo, para una persona sería muy difícil, por no decir imposible, escribir largos programas con unos y ceros. Para esto cada computadora trae un lenguaje ensamblador específico.

Las instrucciones en lenguaje ensamblador tienen el siguiente estilo:

```
MOV R1, R2
```

```
INC R2
```

```
SUB R2, R1
```

Como se observa, tampoco es un lenguaje muy amigable. Este lenguaje traduce las instrucciones al lenguaje máquina para que pueda ejecutarlas. Decimos que es un lenguaje de muy bajo nivel porque las instrucciones están directamente relacionadas con las instrucciones que ejecuta la máquina.

Los lenguajes de programación como C tienen un nivel de abstracción que está por encima de los lenguajes ensambladores. Por eso decimos que son de más alto nivel. Este lenguaje es más comprensible por un humano medianamente entrenado, pero menos comprensible para la máquina, debiendo traducir al ensamblador y luego al lenguaje máquina.

El lenguaje Python, que aprenderemos, es aun de más alto nivel que C. Para tener una idea, un sencillo programa que imprima por pantalla la palabra “hola” y no haga otra cosa, en ensamblador podría requerir unas 15 líneas, en C unas 5 líneas y en Python solo una.

Lenguajes compilados e interpretados

- Compilados. Los lenguajes compilados como C necesitan de un compilador que traduzca las instrucciones al lenguaje ensamblador y las guarde en un archivo ejecutable. Este proceso se realiza una sola vez.
- Interpretados. En cambio, en los lenguajes interpretados se necesita de un intérprete que va traduciendo línea por línea las instrucciones a medida que se van ejecutando. Esto se realiza cada vez que el programa se ejecute.

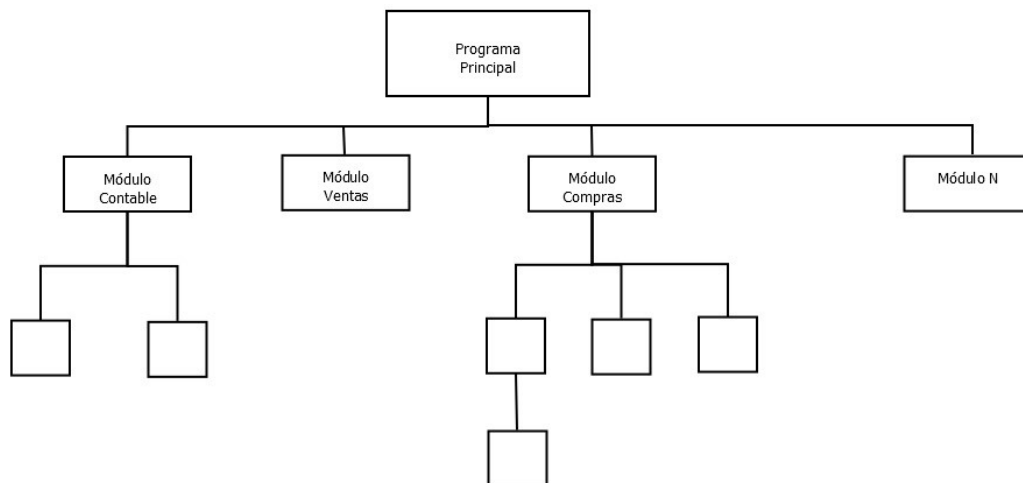
Un ejemplo gráfico sería adquirir un libro traducido del inglés al español (esto en el caso de los compilados). El texto ya está traducido y uno lo lee las veces que quiera. En cambio, si recibimos la visita de una persona que hable en inglés acompañado por un traductor que nos va diciendo frase por frase lo que la persona dice, sería el equivalente a los lenguajes interpretados.

	COMPILADO	INTERPRETADO
VENTAJAS	<ul style="list-style-type: none"> • Su ejecución es más rápida en general 	<ul style="list-style-type: none"> • Se puede correr aunque no todas las instrucciones sean correctas
DESVENTAJAS	<ul style="list-style-type: none"> • Si no compila no se puede correr nada 	<ul style="list-style-type: none"> • En general su ejecución es más lenta

Programación estructurada

Cuando los programas empiezan a ser largos, y tienen que hacer varias cosas, se deben modularizar. Por ejemplo, una aplicación que se encarga de la gestión de una empresa, como ser ventas, compras, la parte contable, sueldos, etcétera, puede tener cientos de miles de líneas de código. Sería muy ingenuo tratar de escribirlo todo de corrido, ya que el código sería imposible de alterar o corregir porque sería imposible encontrar algo. Además, la secuencia no tiene por qué siempre ser la misma, en algún momento se querrá consultar la parte contable y, después, las ventas y, en otro momento, primero las compras, etc.

La idea de modularizar es que el programa principal llame a los módulos que se encargan de cada cosa, en este caso tendríamos un módulo para la parte contable, otro para ventas, etc. A su vez, cada módulo llamará a otros submódulos y así. En forma de esquema se vería de la siguiente forma:



La modularización es muy importante porque, por ejemplo, si tuviéramos que hacer algún cambio en el módulo Contable, sabemos que afectará solo a dicho módulo y a los submódulos, sin afectar al resto, como Ventas, Compras, etc.

- ¿Cómo se modulariza? En principio a través de funciones. Luego aprenderemos a crear módulos pre compilados que utilizaremos en nuestros proyectos.
- ¿Dónde se definen las funciones? Antes del programa o estructura principal o antes de cualquier otra función que la llame.
- ¿Dónde se utilizan? Desde el programa principal o desde cualquier otra función. Tienen que estar definidas antes —más arriba— para poder llamarlas.
- ¿Hasta cuándo se debe seguir subdividiendo en funciones?

Esto es muy importante.

- Cada función o procedimiento tiene que hacer UNA SOLA cosa. Por ejemplo, si se deben cargar los datos de una matriz cuadrada y, luego, calcular su determinante, necesitaremos una función que cargue la matriz y otra que calcule y devuelva su determinante.
- Una regla para tener en cuenta es que una función tiene que tener, como mínimo dos o tres líneas (si solo tiene una línea no hace falta ninguna función) y, como máximo, unas 15 o 20 líneas. Esto es solo un indicador, a lo mejor puede haber alguna función de más de 20 líneas que sea correcta, pero en general si hay más de 10 o 15 líneas hay que sospechar que no se está modularizando de forma adecuada.

Pseudocódigo

El pseudocódigo o falso lenguaje es un lenguaje de computación inexistente que sirve para definir un algoritmo, pero no para hacer su prueba en una máquina. Como es una convención podemos escribir instrucciones en español (un habitante de Indonesia podría hacerlo en indonesio).

Tipos de estructuras de los algoritmos

En programación estructurada nos manejaremos con combinaciones de tres tipos de estructuras: las secuenciales, las selectivas (o condicionales) y las repetitivas (o iterativas).

Estructura secuencial

Es la estructura más simple, son las instrucciones que se ejecutan una detrás de otra. En general las instrucciones serán: imprimir algo, leer algún dato ingresado por el teclado, hacer algún cálculo.

Ejercicio 1. Definir en pseudocódigo un programa que pida al usuario que ingrese dos números, los sume y los muestre.

Resolución

```
1. num1 ← ingresar ("Ingrese un número")
2. num2 ← ingresar ("Ingrese otro número")
3. resultado ← num1 + num2
4. imprimir (resultado)
```

Estamos diciendo que la instrucción *ingresar* espera que un usuario ingrese algo por teclado, en este caso avisa que es un número y lo guarda en una variable. En la línea 1 lo guarda en *num1*, y el otro número en *num2*. La flecha indica que el valor ingresado se debe guardar en esa variable.

En la línea 3 se suman los valores y se guardan en otra variable que llamamos *resultado*. Por último, con la instrucción *imprimir*, mostramos la suma por pantalla.

Nótese algo, si hubiéramos escrito resultado con comillas, de esta forma:

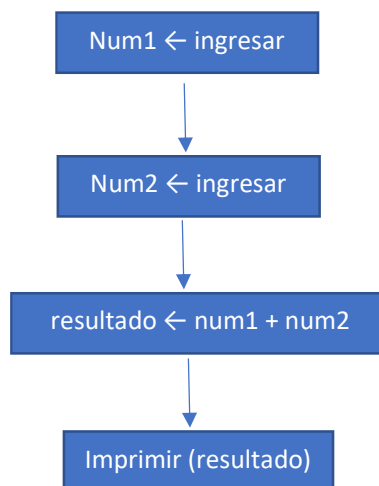
Imprimir("resultado")

Por pantalla veríamos la palabra "resultado", en lugar del valor numérico. Es decir, si queremos que algo salga textual lo encerramos entre comillas, de lo contrario, el lenguaje interpreta que es una variable y busca el valor guardado allí.

Este programa tiene un flujo secuencial, es decir, si lo ejecutamos varias veces, por más que ingresemos números distintos, el flujo del algoritmo siempre será el mismo.

Diagramas de flujo

Son diagramas que sirven para ver la estructura de un algoritmo de manera rápida. El programa anterior, en un diagrama de flujo quedaría:



Estructuras selectivas o condicionales

Son estructuras que desvían el flujo de ejecución en diferentes ramas según alguna condición.

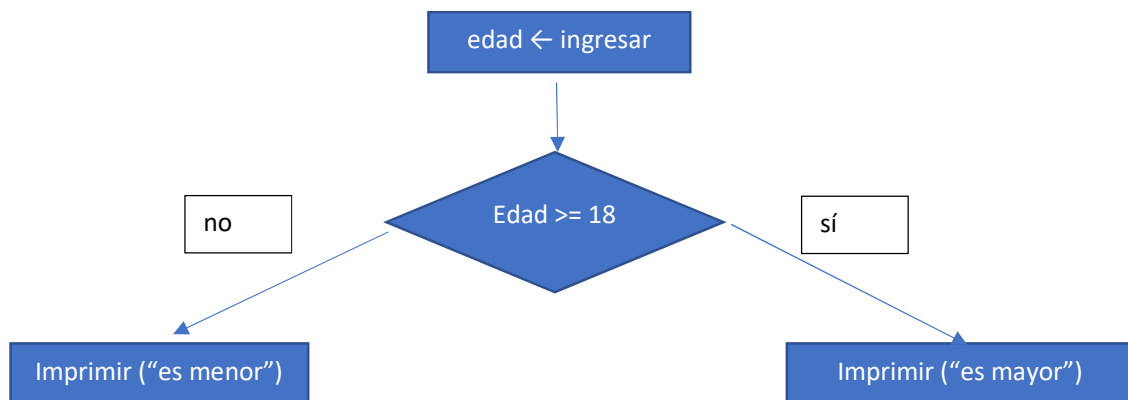
Ejercicio 2. Se ingresa una edad y un cartel indica “si es mayor” o “si es menor” de edad.

```
1. edad ← ingresar ("Ingrese su edad")
2. si (edad >= 18):
    • imprimir ("es mayor de edad")
3. si no:
    • imprimir ("es menor de edad")
```

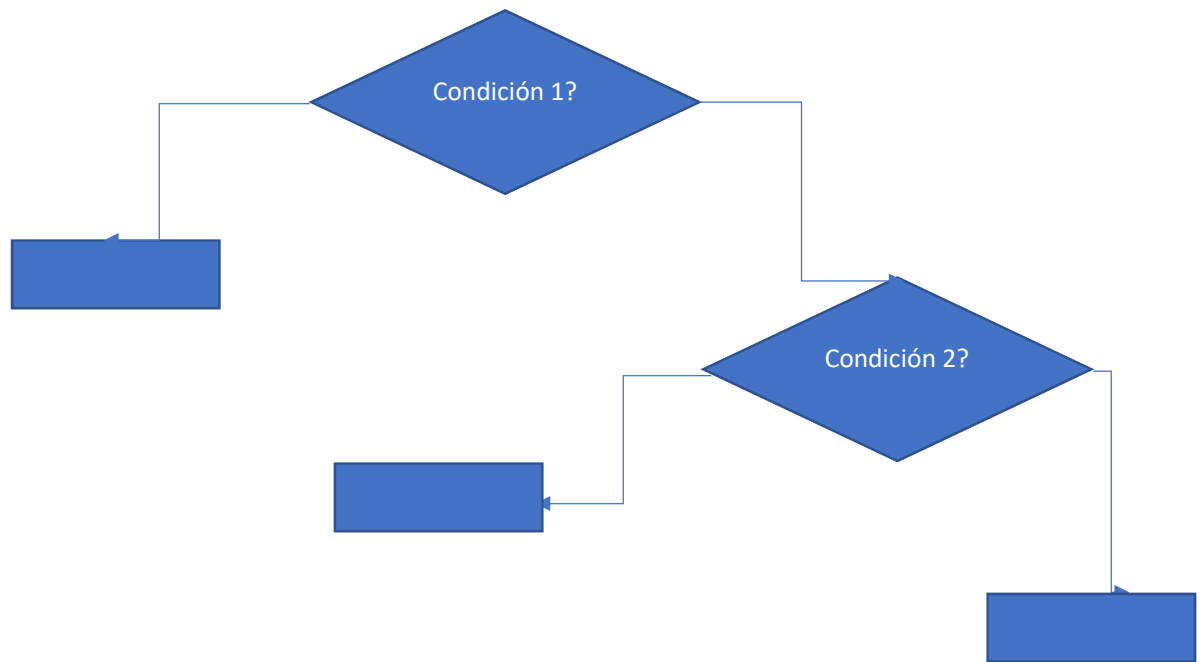
En un diagrama de flujo, la verificación de la condición se representa mediante un rombo:



En nuestro ejemplo:



Las consultas pueden estar anidadas, es decir, adentro de una pregunta puede haber otra, con lo cual quedarían estructuras más complejas.



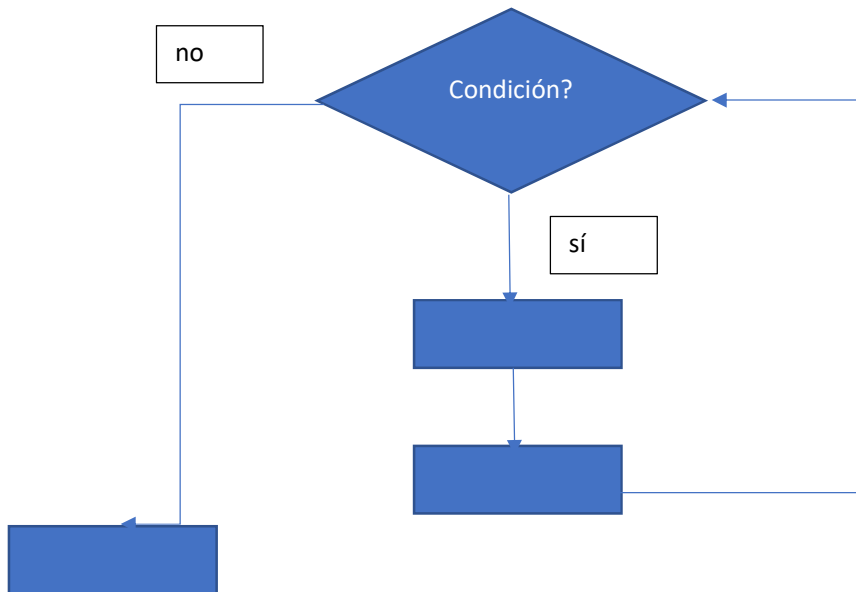
Estructuras repetitivas o iterativas

Sirven cuando hay que repetir una serie de pasos varias veces mientras se cumpla cierta condición.

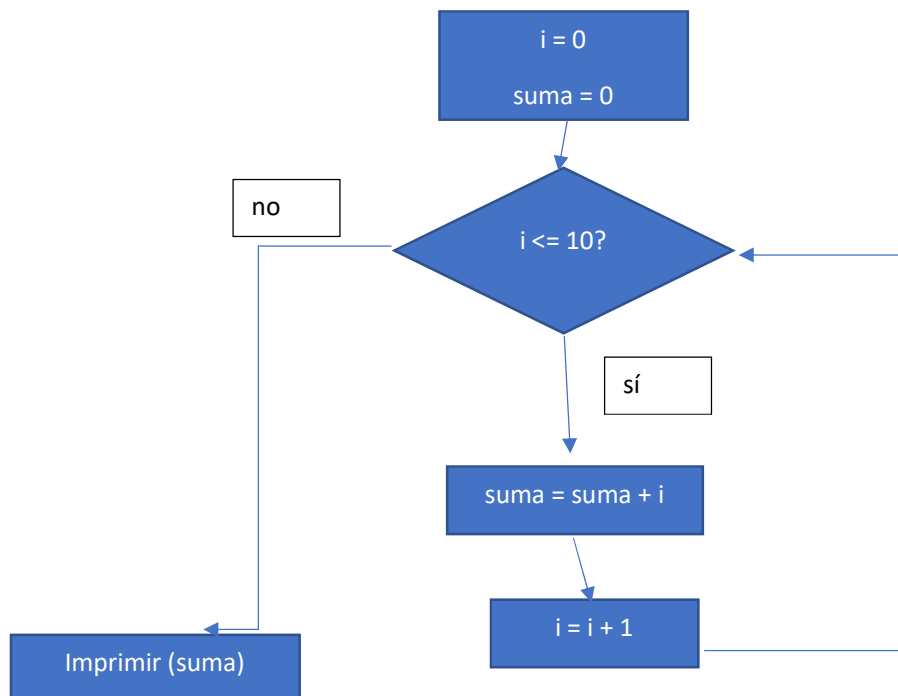
Ejercicio 3. Sumar los números naturales del 1 al 10 e imprimir su valor.

```
1. i ← 1
2. suma ← 0
3. mientras (i <= 10):
    • suma ← suma + i
    • i ← i + 1
4. imprimir (suma)
```

La representación en un diagrama de flujo es similar a la condicional pero una de las flechas vuelve al ciclo:



En nuestro ejemplo:



También, al igual que las condicionales, podemos tener repetitivas anidadas o una combinación de todo.

Python

Características

- Muy alto nivel
- De rápido aprendizaje
- Versátil
- Interpretado y compilado. El bloque principal de una aplicación es interpretado, pero los módulos que utilice vienen compilados, por lo que no es lento en la ejecución.
- Orientado a objetos, aunque puede utilizarse en programación estructurada.
- Muy popular en la actualidad (2018)
- Tipado dinámico (se verá enseguida qué significa esto)

El nombre tiene que ver con el grupo británico de humoristas Monty Python (foto de portada), ya que Guido Van Rossum, el creador del lenguaje, es fanático de estos comediantes.

Si bien es un lenguaje con muchas virtudes no lo recomiendo como el primer lenguaje a ser aprendido. ¿Por qué? Por distintos motivos, el principal es que se pierden algunas nociones.

Por ejemplo, si escribimos

```
x = 5
x = "Juan"
```

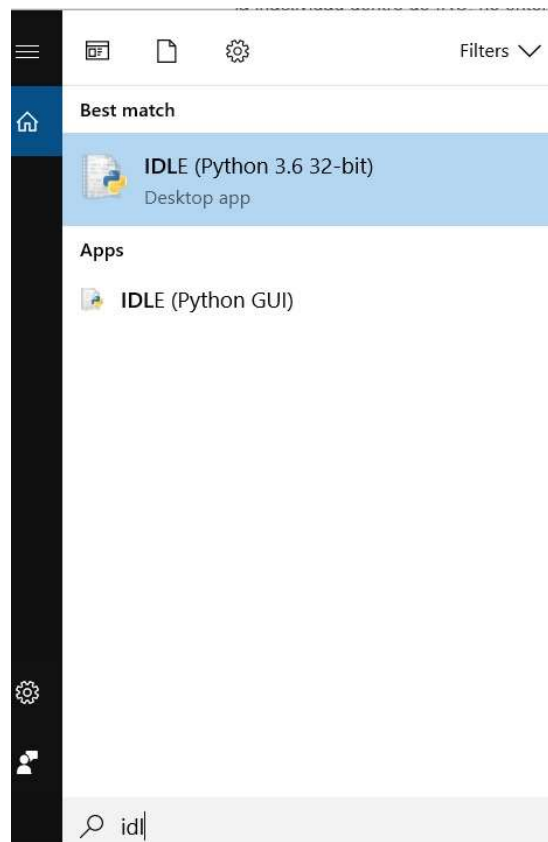
En cualquier otro lenguaje deberíamos decir de qué tipo es la variable. Es decir, no es lo mismo guardar un número que una cadena de caracteres. Sin embargo, Python lo acepta sin ningún problema. En la primera línea crea una variable (en realidad es un objeto) de tipo entera, y luego la destruye y crea otra de tipo string. Lo cual es peligroso, en especial a la hora de pasar parámetros a una función.

Por otro lado, si quisiéramos ordenar un vector en un lenguaje como C deberíamos escribir varias líneas de código, pero en Python, solamente alcanza con escribir el nombre del vector (en realidad una lista) un punto y la palabra "sort".

Si bien esto es una gran facilidad para el usuario, los programadores novatos no saben cómo es el proceso interno de ordenar una lista, por lo que pierden ese contacto.

Comenzando

El lenguaje Python es de fácil aprendizaje porque es muy similar a un pseudocódigo en inglés. En primer lugar, debemos bajar el intérprete, en el sitio oficial de Python se puede descargar en forma gratuita: <https://www.python.org/>. Luego de elegir el correspondiente al sistema operativo de la máquina, se baja la última versión (3.6) y se instala. Al finalizar podremos abrir el IDLE



Al ejecutarlo se abrirá una ventana que es donde tendremos la salida de nuestros programas. Para esto abrimos un nuevo archivo (File – new File) y le ponemos un nombre, en forma automática colocará extensión .py.

Entonces tenemos dos ventanas: en una, el nuevo archivo que creamos, escribiremos el código. En la otra veremos la salida.

Por tradición el primer programa en cualquier lenguaje es la impresión de un cartel en pantalla que diga “Hola mundo”.

Por lo tanto, escribimos

```
print ("hola mundo!!")
```

Guardamos el archivo y en el menú Run hacemos clic en Run Module, en la otra ventana veremos un cartel que dice “hola mundo!!”.

Pasado el entusiasmo inicial, como nadie nos pagará millones de pesos por un programa que diga “hola mundo!!”, lo modificamos para que nos salude.

Por lo tanto, podremos cambiar el “mundo” por “José”, “Rosa”, “Juan” o como nos llamemos. Sin embargo, este programa funcionaría solo para las personas que se llamen José o Rosa o Juan. Lo que se nos ocurre es que podríamos preguntarle el nombre al usuario y luego saludarlo.

Aquí empieza a jugar el papel de las variables. Para guardar ese dato, el nombre, debemos tener una variable en donde después podamos recuperar la información.

Esa variable, en principio, podría llamarse de cualquier manera, pero si vamos a guardar un nombre es deseable que se llame “nombre”.

Por lo tanto, tenemos que pedirle al usuario que ingrese su nombre por teclado y guardar lo que se ingresó.

Nuestro programa quedará:

```
1. nombre = input ("Ingrese su nombre: ")
2. print ("hola ", nombre)
```

Al correrlo, en la ventana donde vemos la salida, estará esperando que ingresemos nuestro nombre. Luego de ingresarlo y presionar *enter*, imprimirá el saludo.

Hay que notar que la palabra “hola” se escribió entrecomillas y nombre, como es una variable, no. Entre ellas, una coma separa los parámetros: uno es el texto “hola” y el otro la variable que guarda cierto valor ingresado.

Variables

Las variables las podemos pensar como “cajas” en donde se guarda información, en el ejemplo anterior, un nombre.

Una variable siempre deberá tener dos usos: uno es el que guarda la información, porque si no guardamos nada, no hay nada, ni siquiera está definida la variable. El otro uso es imprimirla, hacer un cálculo, etc. Es decir, utilizar la información que está ahí guardada.

Si en el programa anterior nunca imprimiéramos el nombre, quiere decir que a esa variable no le estamos dando ningún uso, por lo tanto, está de más.

Nombres de variables

Python distingue entre mayúsculas y minúsculas, por lo cual, las variables *nombre*, *Nombre* y *NOMBRE* son tres variables diferentes. De todas formas, no se recomienda utilizar este tipo de variantes.

Podemos usar cualquier letra, tanto en mayúsculas como en minúsculas, dígitos (pero no comenzar con uno) y guión bajo. Además, no pertenecer a una palabra reservada. Es decir:

- Cualquier combinación de letras.
- Guión bajo.
- Dígitos (pero no comenzar con ellos)

No acepta:

- Espacios en blanco.
- Palabras reservadas para el lenguaje. Por ejemplo, si quisiéramos poner de nombre "print" no lo permitiría porque es una instrucción del lenguaje.

Ejemplos

nombre	Válido
Num1	Válido
1num	Inválido: comienza con un número
_num	Válido
Num 1	Inválido: tiene un espacio
input	Inválido: es palabra reservada

Las anteriores son reglas que nos exige el lenguaje, de no cumplirlas, nuestro programa dará un error cuando lo ejecutemos.

Sin embargo, hay otras reglas que, si bien no son obligatorias, hacen del uso de normas pre establecidas como buenas prácticas de programación.

Nombres de variables

- El nombre de la variable tiene que describir el dato que guarda. Si fuéramos a guardar un nombre, la variable se podría llamar x1, y el programa funcionaría correctamente. Sin embargo, si estamos escribiendo programas largos, no recordaríamos que para guardar un nombre usamos una variable que se llama x1. Por lo tanto, lo correcto es *nombre* o alguna variante.

- Las variables, por convención van en minúsculas por completo, salvo que tengan palabras compuestas. Por ejemplo, *nombre* es correcto, en cambio *Nombre* o *NOMBRE*, no.
- Si tenemos una variable que guarda un sueldo en bruto y otra el sueldo a cobrar, deberíamos distinguirlas. No está bien llamarlas *sb* y *sn*, por sueldo bruto y sueldo neto. Es mejor poner las dos palabras. Pero, recordemos que no podemos poner un espacio. Entonces, las opciones para el sueldo bruto son:

sueldobruto
sueldo_bruto
sueldoBruto

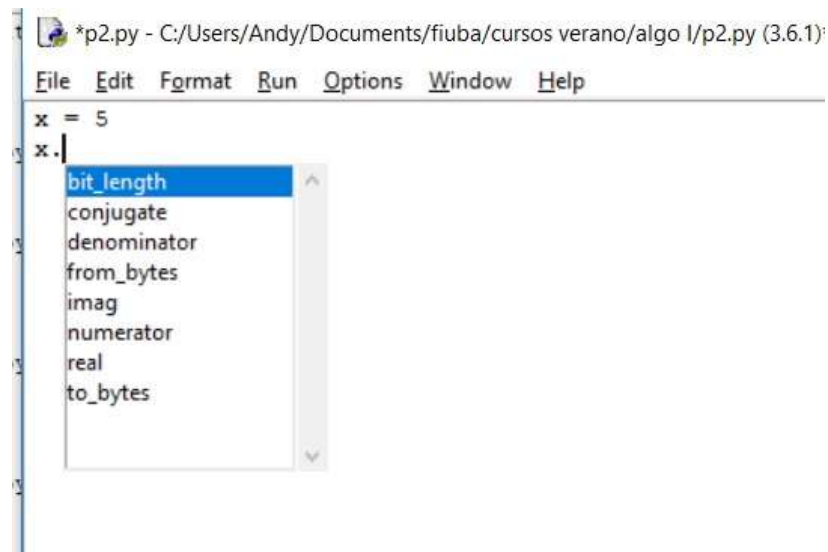
En el primer caso puede costarnos la lectura, y un vistazo rápido podría no reconocer ambas palabras. El segundo caso es el más claro y explícito. Sin embargo, el caso más utilizado es el último, que se llama camelCase (por las jorobas de un camello). Cualquiera de las dos últimas opciones que elijan es correcta.

Tipos de variables

¿Cómo? ¿Hay tipos de variables?

La respuesta es sí, y nos puede sorprender. Sin embargo, nosotros nunca decimos que tal variable es de tal tipo, pero es algo que Python asume de manera automática cuando guarda el dato.

Al escribir `x = 5`, asume que `x` es una variable de tipo entera y la maneja como tal. En realidad, lo habíamos mencionado, no es una variable, sino un objeto. Hagamos esta prueba: escribimos `x = 5`, y debajo escribimos `x.`, un punto, y esperamos un par de segundos. Vemos:



La lista que aparece al costado son todos los métodos de los que disponemos. ¿Qué son los métodos? Son funciones que tiene el objeto, y que se pueden utilizar. Por ejemplo, si escribimos `bit_length` y lo imprimimos, nos indica cuántos bits necesita para guardar el dato que tiene.

```
x = 5
print ("bits necesarios para guardar ", x, ": ", x.bit_length())
```

La salida será:

```
type Copyright, credits or license() for more information.
>>>
===== RESTART: C:/Users/Andy/Documents/fiuba/cursos verano/algo I/p2.py =====
bits necesarios para guardar 5 : 3
>>> |
```

Si ponemos un número más grande y lo corremos veremos que necesita más bits para almacenarlo.

Además disponemos de una función interesante que es *type*, y nos devuelve de qué tipo es la variable.

En este ejemplo, si escribimos

```
print (type (x))
```

La salida será:

```
===== RESTART: C:/Users/Andy/Documents/fiuba/cursos verano/algo I/p2.py =====
bits necesarios para guardar 5 : 3
<class 'int'>
>>>
```

Donde vemos que no es una variable, sino un objeto del tipo de clase “int” (de entero).

Otros tipos

- float. Si en lugar de `x = 5`, escribimos `x = 5.2`, nos dirá que es de tipo “flotante”.
- str. Si escribimos `x = “Juan”` nos dirá que es de tipo string. Nótese que no existe el tipo char. Se puede probar haciendo `x = ‘a’`, que también lo tomará como un string.

- `bool`. Si escribimos `x = True`, nos dirá que es de tipo `bool`, es decir, una variable que acepta solo dos valores: `True` (verdadero) o `False` (falso). Este tipo de variables nos ayudarán a decidir si debemos optar por una opción o por otra, o cortar un ciclo.

Con estos tipos que son los más básicos nos iremos arreglando por ahora. Nótese que no tiene distintas variables como `long int`, `unsigned char`, etc, como otros lenguajes.

Constantes

Las constantes nos servirán para manejar datos que no deben modificarse salvo situaciones excepcionales. Por ejemplo, si estamos manejando una lista de precios, deberíamos aplicar el IVA a cada uno. Entonces, cada vez que quisiéramos calcular el precio final deberíamos hacer

```
precio_final = precio * 1.21
```

Ese cálculo podría repetirse muchas veces en un programa medianamente largo. Hay que tener en cuenta que el valor del IVA no se modifica desde el año 2012 pero, quizá, alguna medida gubernamental podría indicar que a partir del próximo mes el IVA será del 25% (ojalá que no) o volver al 18% anterior (difícil que esto suceda).

Conclusión, tendríamos que entrar en nuestro programa y modificar todas las apariciones de 1.21 por 1.25 o 1.18, según el caso. Esto no solo es molesto y trabajoso, sino que podría dar lugar a algún error si olvidáramos modificarlo en algún lugar.

En estas ocasiones nos conviene utilizar constantes, y en lugar de escribir 1.21 indicar directamente IVA.

La mayoría de los lenguajes nos permiten trabajar con constantes, lamentablemente, Python, no. Por lo que simularemos su uso, sabiendo que en realidad es una variable más.

Las constantes las definimos en general al principio del programa o en un archivo aparte (más adelante veremos cómo incluir archivos a un proyecto). Por convención van completamente en mayúsculas.

En el ejemplo del IVA, escribiríamos:

```
IVA = 0.21
```

Entonces, el cálculo del precio lo haríamos de la siguiente manera:

```
precio_final = precio * ( 1 + IVA )
```

De esta forma, si el día de mañana este impuesto cambia su valor, simplemente tendremos que tocar una línea de código para adaptarlo.

Hay que tener cuidado de no modificar su valor, ya que el lenguaje permite hacerlo.

En cuanto a las reglas en los nombres corren las mismas que para las variables, con algún cuidado especial: van completamente en mayúsculas y si tienen nombres compuestos nos veremos forzados a utilizar un guión bajo para unir las palabras. Por ejemplo:

```
DESCUENTO_A_CLIENTES = 0.10
```

Operadores

Aritméticos

Para hacer cálculos se usan los operadores que naturalmente manejamos:

Operador	Operación que realiza	Ejemplo	Resultado
+	suma	5 + 3	8
-	resta	6 - 4	2
*	multiplicación	3 * 5	15
/	División con decimales	5 / 3	1.666...
//	División entera	5 // 3	1
%	Resto de una división	5 % 3	2
**	Potencia	2 ** 3	8

De asignación

Para asignar un valor a una variable, lo que en pseudocódigo hacíamos con una flecha, en Python es el “=”.

```
x = 5
```

Hay operadores que son mezclas entre aritméticos y de asignación, que sirven para abreviar la escritura. Por ejemplo, si queremos sumar 1 a una variable podemos escribir

```
x = x + 1
```

En la forma abreviada

```
x += 1
```

Esto mismo se puede utilizar con el operador de multiplicación, división, etc.

Comparación

Los operadores de comparación sirven para relacionar dos elementos en consultas si son iguales, distintos, si uno es mayor o menor que otro, etc.

Hay que tener cuidado porque para ver si un elemento es igual a otro no se utiliza el “=” ya que este es de asignación. Se usa un doble igual “==”.

Estos operadores devuelven verdadero (True) o falso (False).

==	Igual
>	Mayor
>=	Mayor o igual
<	Menor
<=	Menor o igual
!=	Distinto

Lógicos

Los operadores lógicos los usaremos cuando queremos ver más de una condición. Por ejemplo, queremos ver si la variable a es mayor que 5 y la b menor que 8. Tenemos un conector lógico que es el “y”.

and	y
or	o inclusivo
not	Negación

Tablas de valores de verdad

Cuando consultamos si a es mayor que 5 y b menor que 8, nos devuelve verdadero solo si ambos son verdaderos. En cualquier otro caso nos da falso porque una o ambas condiciones no se cumplen.

En cambio, si el conector lógico es una o (or) nos da verdadero cuando se cumple una condición (cualquiera) o ambas. Por eso decimos que es inclusivo.

La negación invierte el valor: si el resultado es verdadero lo convierte en falso y viceversa.

not

Valor	Resultado
True	False
False	True

Valor 1	Valor 2	and	or
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Estructuras selectivas

Las estructuras selectivas se manejan con las sentencias *if*, *else* y *elif*.

Se utiliza de la siguiente manera:

```
if (condición):  
    #sentencias  
else:  
    #otras sentencias, esta sección es opcional
```

El ejemplo que hicimos del usuario que ingresa una edad e indica si es mayor o no, en Python sería:

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

Sin embargo, al correrlo, nos da un error. ¿Qué sucedió? El problema es que edad es una variable de tipo string. Todo lo que se ingrese por teclado será de tipo string, y compara con 18 que es un entero. Por lo tanto, debemos cambiar la variable que es de tipo string a tipo entera. Esto se llama “casteo”, y se realiza indicando el tipo al que se quiere convertir. En este caso int.

Cambiamos la primera línea por

```
edad = int(input("Ingrese su edad: "))
```

Ahora funciona correctamente.

Nótese que detrás de la consulta por la condición va un dos puntos, y las sentencias que se escriban debajo tienen que mantener cierta sangría, es decir, dejar algunos espacios. ¿Cuántos? Uno, dos, los que se desee, pero siempre hay que mantener los mismos. Las buenas prácticas indican que estos espacios deben ser cuatro. El mismo IDE coloca esos espacios por defecto.

Todas las sentencias que mantengan esos cuatro espacios de margen estarán dentro del mismo bloque. En otros lenguajes los bloques se abren y cierran con las palabras begin y end (en Pascal) o con llaves de apertura y cierre (en C, Java, etc). En Python basta con dejar cierta sangría.

Ejercicio 4. Se pide ingresar dos notas que corresponden a un primer parcial y segundo parcial del CBC. Luego decidir si el estudiante promociona (si suma 13), va a final (si suma 8 pero menos de 13) o recursa.

Solución 1

```
p1 = int(input("Ingrese la nota del primer parcial: "))
p2 = int(input("Ingrese la nota del segundo parcial: "))
suma = p1 + p2

if (suma >= 13):
    print ("promociona")
else:
    if (suma >= 8):
        print ("tiene que dar final")
    else:
        print ("debe recursar")
```

Esto funciona correctamente. Observamos que hay bloques anidados, la misma sangría nos lo indica. Nótese que en el segundo if no tenemos que preguntar si suma es menor que 13 porque si no lo fuera hubiera ingresado en el primer if. Si bien este programa es correcto es preferible escribirlo de otra manera: utilizando la instrucción elif que resume un else con un if a un mismo tiempo.

Solución 2

Las primeras tres líneas permanecen igual, luego escribimos:

```
if (suma >= 13):
    print ("promociona")
elif (suma >= 8):
    print ("tiene que dar final")
else:
    print ("debe recursar")
```

Este programa es más corto y más claro que el anterior.

Ejercicio 5. ¿Cómo arreglarían el programa anterior para que promocione si suma 13 o más pero que no tenga un aplazo en uno de los dos parciales? Es decir, no se puede tener un 10 y un 3. Tratá de pensarlo y hacerlo. Luego, mirá la solución propuesta.

Solución

Aquí entran a jugar los operadores lógicos.

```
p1 = int(input("Ingrese la nota del primer parcial: "))
p2 = int(input("Ingrese la nota del segundo parcial: "))
suma = p1 + p2

if (suma >= 13) and (p1 != 3) and (p2 != 3):
    print ("promociona")
```

```
elif (suma >= 8):  
    print ("tiene que dar final")  
else:  
    print ("debe recursar")
```

Estructuras repetitivas

En principio nos manejaremos con dos maneras distintas de hacer ciclos:

- for – range
- while

En el primer caso usaremos una variable que vaya iterando (variando) los valores indicados en el rango (range). Por ejemplo, para imprimir los números del 1 al 10 haríamos:

```
for i in range(1, 11):  
    print (i)
```

La variable *i* se va moviendo por el rango indicado, en primer lugar, empieza en 1, luego en 2, hasta llegar al tope. Nótese que el límite inferior es el valor desde dónde comienza, pero el superior es uno más, ya que no lo incluye.

Por defecto incrementa de a un valor, pero podemos agregarle un parámetro más que indica el salto entre dos valores:

```
for i in range(1, 11, 2):  
    print (i)
```

En este caso imprime

```
1  
3  
5  
7  
9
```

También podemos hacer que sea descendente:

```
for i in range(10, 0, -1):  
    print (i)
```

Acá imprime

```
10  
9  
...  
1
```


Se aconseja fuertemente no modificar dentro del ciclo la variable de iteración porque los resultados pueden ser imprevistos. Por ejemplo, la salida del siguiente programa:

```
for i in range(1, 10):  
    i = i + 2  
    print (i)
```

No es

3

5

Etc

Sino

3

4

5

etc

La otra sentencia para hacer ciclos, while, se ejecuta mientras se cumpla cierta condición:

```
while (condición):  
    # sentencias
```

Por ejemplo, para listar los números del 1 al 10 con un while sería:

```
i = 1  
while (i <= 10):  
    print (i)  
    i += 1
```

Hay que notar que la variable no se incrementa de manera automática como en el for y debemos inicializarla. Por eso, cuando la longitud de los ciclos es de antemano conocida conviene utilizar un for ya que es más sencillo.

El while nos servirá cuando desconocemos la cantidad. Por ejemplo, se ingresan una serie de notas y se quiere calcular su promedio. El ciclo de ingreso finaliza cuando se ingresa una nota igual a cero. En este problema no sabemos cuántas notas se ingresarán, por este motivo debemos usar un while.

Funciones

Para realizar programas modulares como decíamos al principio necesitamos definir funciones que hagan ciertas tareas. Hasta ahora los ejemplos que indicamos eran programas muy sencillos que hacían solo una cosa, por lo que no necesitamos

modularizar, es decir, dividir el programa en funciones en las que cada una haga una tarea. Sin embargo, a medida que se van haciendo programas más complejos y largos, necesitaremos hacer esto. Por ejemplo, se cargan una serie de datos en una lista, luego se realizan cálculos y en base a estos cálculos se imprimen informes, podríamos tener tres bloques (tres funciones): una que cargue los datos, otra que haga los cálculos y otra que imprima el reporte. A su vez, cada una podría llamar a otras, dependiendo la complejidad del problema.

Para definir una función, simplemente ponemos la palabra `def` seguida del nombre de la función y una lista de parámetros que podría estar vacía.

Debemos distinguir dos momentos: uno es el que la función se define (único), otro es cuando se llama (pueden ser uno o más llamados).

```
def saludo():  
    print ("hola")  
  
print ("voy a llamar a la función saludo")  
saludo()
```

En el primer bloque se define la función que se llama `saludo` y no lleva parámetros, por este motivo los paréntesis están vacíos. Debajo de los dos puntos va el código.

En el segundo bloque se llama a la función por su nombre y la lista de parámetros (en este caso es vacía).

Comentarios

Los comentarios en cualquier lenguaje de programación sirven para indicar o aclarar algunas cosas. Pero estas aclaraciones son para los seres humanos, la máquina no las procesa.

Es una buena práctica indicar qué hace una función mediante un comentario al principio. En Python, el signo `#` indica que lo que viene luego, hasta el final de la línea, es un comentario.

Si necesitamos comentarios de varias líneas se pueden usar las triples comillas `"""` que abren y cierran los comentarios.

```
# Defino la función saludo  
def saludo():  
    print ("hola")  
  
""" Bloque principal del programa  
    este comentario  
    tiene varias líneas """  
print ("voy a llamar a la función saludo")  
saludo()
```

Parámetros

Si quisiéramos que el módulo salude a distintas personas podemos pensar que sería útil pasarle el nombre por parámetro.

Ejemplo

```
# Defino la función saludo
def saludo(nombre):
    print ("hola ", nombre)

""" Bloque principal del programa
    este comentario
    tiene varias líneas """

print ("voy a llamar a la función saludo")
saludo("juan")
saludo("rosa")
```

Es decir, llamamos dos veces a la función saludo con diferentes parámetros.

Devolución o retorno de una función

En el caso anterior la función no retorna nada, solo imprime un saludo a la persona que se pasa por parámetro. La mayoría de las veces vamos a necesitar que la función nos haga una devolución, por ejemplo, un cálculo. Esto se realiza con la palabra return.

Ejemplo

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

La función sumar toma dos parámetros, los suma y devuelve su valor. Podríamos haberla escrito en una sola línea indicando

```
    return x + y
```

Hay que tener cuidado en el llamado porque Python no chequea compatibilidades. Por ejemplo:

```
a = 5
b = 8
print (sumar(a,b)) # primer llamado
a = 5.1
b = 3.2
print (sumar(a,b)) # segundo llamado
a = "juan"
b = "rosa"
```

```
print (sumar(a,b)) # tercer llamado
```

En el primer llamado suma dos enteros y devuelve 13.

En el segundo suma dos números flotantes y devuelve 8.3.

En el tercer llamado recibe dos strings, y la suma es un operador que está sobrecargado, esto significa que funciona de manera diferente dependiendo con qué tipo de variable esté tratando. Con los strings lo que realiza es la concatenación, por lo que devuelve "juanrosa".

Por supuesto, si le pasamos un string y un número dará error en el momento en que lo estemos ejecutando.

Pasaje por referencia y por valor

En otros lenguajes podemos indicar de forma explícita que ciertos parámetros los queremos pasar por referencia y otros por valor. ¿Qué significa esto?

- Por valor. Significa que la variable que pasemos cuando llamamos a la función se copia en una nueva variable. Por lo tanto, cualquier modificación que la función haga sobre esa variable será realizada en la copia. Es decir, la variable original sigue conservando el valor original.
- Por referencia. Significa que se pasa una referencia a la variable, es decir que la función trabaja directamente con la variable original, por lo tanto, cualquier modificación afectará a la variable original.

Ejemplo:

```
def f(x):  
    x = 8  
  
a = 5  
f(a)  
print(a)
```

Tenemos una función f que lo único que hace es asignarle el valor 8 a la variable que recibe. Afuera de la función definimos una variable a , asignándole el valor 5, luego llamamos a la función.

Cuando imprime a , si está

- pasada por referencia imprime 8.
- pasada por valor imprime 5.

En Python no podemos indicarle que tome los parámetros por referencia o por valor, lo hace de forma automática. Para decirlo de una forma simplificada:

- Las variables como x de tipo entera o las flotantes o strings pasan por valor. Es decir, no se modifican por más que las cambiemos adentro de la función.
- Las variables de tipo lista o diccionarios pasan por referencia.

En realidad, siempre están pasadas por valor, lo que sucede es que la variable que pasamos es una referencia o puntero al objeto, no es el objeto con todos sus datos. Cuando veamos listas explico este punto.

Retornos múltiples

Python, a diferencia de otros lenguajes, permite una devolución múltiple. Por ejemplo, si queremos devolver la suma y la resta de dos números podríamos hacer lo siguiente:

```
# Función que devuelve la suma y la resta
# de dos números
def sumar_restar(x,y):
    suma = x + y
    resta = x - y
    return suma, resta

a = 8
b = 5
# llamado de la funcion sumar_restar
suma, resta = sumar_restar(a,b)
print ("la suma es: ", suma, " y la resta: ", resta)
```

La salida será:

```
la suma es: 13 y la resta: 3
```

Si bien esto fue un ejemplo práctico para ver cómo Python puede devolver varios valores, tené en cuenta que no estamos cumpliendo un principio básico que dijimos: una función debe hacer UNA sola tarea. Por lo que conceptualmente no estaría bien ese código.

En el mismo ejemplo, si en lugar de colocar las dos variables, suma y resta, ponemos una sola, de esta forma:

```
resultado = sumar_restar(a,b)
print(resultado)
```

La salida será

```
(13, 3)
```

¿Qué es lo que sucedió? Convirtió los dos resultados en uno solo, agrupándolo en una tupla, estructura que pronto veremos.

Módulos

Hasta el momento, para poder familiarizarnos con el lenguaje, los ejemplos que hemos visto son muy simples y acotados. Los programas reales hacen muchas cosas: cargan datos, los actualizan, hacen cálculos e informes, muestran gráficos. No sería buena idea escribir todo nuestro código en un mismo archivo ya que cualquier cambio que quisiéramos hacer sería muy dificultoso y podría repercutir en otras partes del programa sin que nos diéramos cuenta.

Es una buena práctica agrupar las funciones mediante algún criterio y armar con ellas módulos. Por el momento el criterio de agrupación que vamos a usar será de tipo funcional, por ejemplo, si en un programa necesitamos cargar datos, realizar cálculos, generar reportes y mostrar ciertos gráficos, sería lógico contar con cuatro módulos: en uno estarían todas las funciones que se dedican a la carga de datos, en otro, las que los procesan, en un tercero, las que generan los reportes y en el cuarto las que realizan gráficos.

¿Cómo definimos un módulo?

Es muy sencillo, supongamos que tenemos dos funciones: factorial, la cual calcula el factorial de un número, y *ceros*, la cual calcula los ceros de una función cuadrática. El código sería:

```
def factorial(n):  
    # código  
  
def ceros(a, b, c):  
    # código
```

Con estas dos funciones queremos hacer un módulo que llamaremos *calculo*. Simplemente guardamos el archivo *calculo.py*. Cuando necesitemos utilizarlo, debemos importarlo con la instrucción *import* y el nombre.

Para el llamado de las funciones debemos poner el nombre del módulo, un punto, y el nombre de la función.

```
# uso del módulo  
import calculo  
  
print("El factorial de 5 es: ", calculo.factorial(5))  
print("Las raíces son: ", calculo.ceros(1, 1, -6))
```

Programa principal (main)

En lenguajes como C, C++ o Java, tenemos una función principal que se llama *main* y es donde comienza el programa. En Python no tenemos la necesidad de contar con una función principal para que se ejecute el código, simplemente escribimos las sentencias a ejecutar. Es bueno tener presente que ese bloque pertenece a lo que sería el programa principal o bloque que llama a las demás funciones.

En los diagramas de módulo que vimos al principio del texto sería el primer bloque o bloque raíz. Por eso es conveniente agregar algún comentario. En el ejemplo anterior:

```
# uso del módulo
import calculo

# -----
# Programa principal
# -----

print("El factorial de 5 es: ", calculo.factorial(5))
print("Las raíces son: ", calculo.ceros(1, 1, -6))
```

Una buena práctica es crear una función *main*. Siguiendo con el mismo ejemplo:

```
# uso del módulo
import calculo

# función main
def main():
    print("El factorial de 5 es: ", calculo.factorial(5))
    print("Las raíces son: ", calculo.ceros(1, 1, -6))

main()
```

Lo que hicimos es poner todo el bloque principal en una función *main*, y luego invocarla.

Módulo *math*

Para hacer ciertos cálculos debemos importar el módulo *math* que ya está escrito. Por ejemplo, para calcular alguna función trigonométrica o la raíz cuadrada de un número, necesitamos importar este módulo.

Como cualquier otro módulo, lo importamos con

```
import math
```

y utilizamos las funciones poniendo el nombre del módulo, punto, y la función. Ejemplo:

```
print("el coseno de 0 es: ", math.cos(0))
print("el arco coseno de 0 es: ", math.acos(0))
print("la raíz cuadrada de 4 es: ", math.sqrt(4))
```

La salida es:

```
el coseno de 0 es:  1.0
el arco coseno de 0 es:  1.5707963267948966
la raíz cuadrada de 4 es:  2.0
```

Módulo random

El módulo *random* es otro que utilizaremos bastante. Para probar ejercicios en donde hay que cargar muchos datos, la carga manual resulta molesta y nos lleva mucho tiempo. Por ejemplo, la carga de una matriz de 4 x 5 nos demanda ingresar 20 datos. Esto es solo si son datos simples. Si hay algún error o queremos testear algún cálculo, ingresar 20 datos cada vez que necesitemos correr el programa nos resultará engorroso. Y esto es un ejemplo de un caso mínimo.

Para estas ocasiones nos serán de utilidad las funciones random para cargar la matriz con datos aleatorios.

Importamos la librería:

```
import random
```

Ejemplo 1

```
# lista diez números aleatorios entre 0 y 1:  $0 \leq x < 1$ 
for i in range(10):
    x = random.random()
    print(x)
```

Ejemplo de salida

```
0.6433148836223158
0.9715545610874391
0.13886834124055802
0.6063499288613263
0.5079642449958351
0.49900393674964716
0.6842927000168526
0.6791401748342877
0.26854697327542354
0.35415537909146044
```

Cada vez que ejecutemos el código listará otros diez números diferentes. A veces necesitamos que liste los mismos números para detectar ciertos errores. En estos casos podemos modificar el número al que se llama “semilla” (seed) que es el número inicial con el que las funciones random hacen los cálculos.

Antes del ciclo agregamos la siguiente sentencia:

```
# fija el valor de la semilla
random.seed(0)
```

Salida:

```
0.8444218515250481
0.7579544029403025
0.420571580830845
0.25891675029296335
0.5112747213686085
0.4049341374504143
0.7837985890347726
0.30331272607892745
```



```
0.4765969541523558
0.5833820394550312
```

Cada vez que ejecutemos el código listará los mismos números. Aclaración: como semilla pusimos 0 pero podíamos haber puesto cualquier otro número.

Ejemplo 2

```
# lista diez números enteros entre 2 y 6:  $2 \leq x \leq 6$ 
for i in range(10):
    x = random.randint(2, 6)
    print(x)
```

Ejemplo de salida:

```
2
5
6
6
4
5
5
4
2
4
```

Ejemplo 3

```
# lista diez números enteros entre 3, 5, 7, 9
for i in range(10):
    x = random.randrange(3, 10, 2)
    print(x)
```

Funciona al igual que el range: arma un rango entre 3 (inclusive) y el 9 (el límite superior queda afuera), tomados de a 2. Es decir, emite números aleatorios eligiendo entre el 3, el 5, el 7 y el 9.

Ejemplo 4

Si bien el módulo random tiene más funciones veremos solo un ejemplo más. Supongamos que tenemos la siguiente lista de salidas posibles:

```
lista = ["cena", "cine", "baile", "paseo en barco", "teatro"]
```

Y no nos decidimos por ninguna, queremos que sea el azar que elija.

```
x = random.choice(lista)
print("el azar elige: ", x)
```

Salida:

el azar elige: baile

Módulo os

El módulo os (por sistema operativo) nos va a servir para limpiar la pantalla y, más adelante, para borrar o cambiarle el nombre a un archivo.

Cuando se imprimen algunos datos en diferentes partes de un programa, nos puede pasar que se mezcle la información. Por ejemplo, un listado del promedio de notas de todos los alumnos a continuación de los datos de un estudiante en particular. O en un juego, una matriz que representa un tablero en la jugada n y otra en la siguiente jugada. Esto puede dar lugar a confusiones, además de que no es una salida elegante. Nos conviene limpiar la pantalla (borrar la información anterior) antes de mostrar los datos actuales.

¿Cómo limpiamos la pantalla?

- En Windows con la instrucción: `os.system("cls")`
- En Linux con la instrucción: `os.system("clear")`

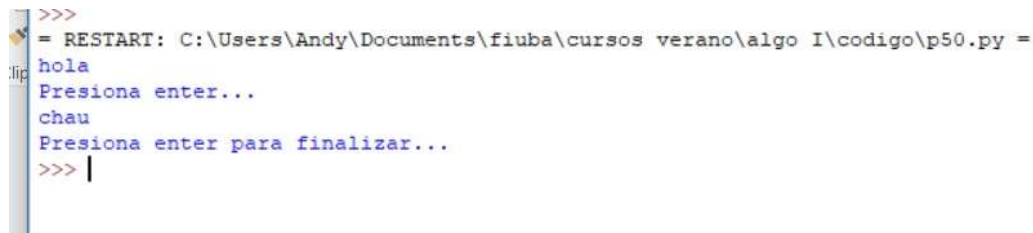
Hay que tener en cuenta que si estamos viendo la salida en el IDLE de Python no veremos el borrado en pantalla, sí cuando lo ejecutemos desde el mismo archivo.

Ejemplo 1

```
import os

print("hola")
# frenamos la ejecución del programa
input("Presiona enter...")
# limpiamos la pantalla
#os.system("cls")
print("chau")
input("Presiona enter para finalizar...")
```

Con input solamente podemos frenar la ejecución del programa. En la salida que vemos en la pantalla del IDLE no hay ningún borrado, como les decía.



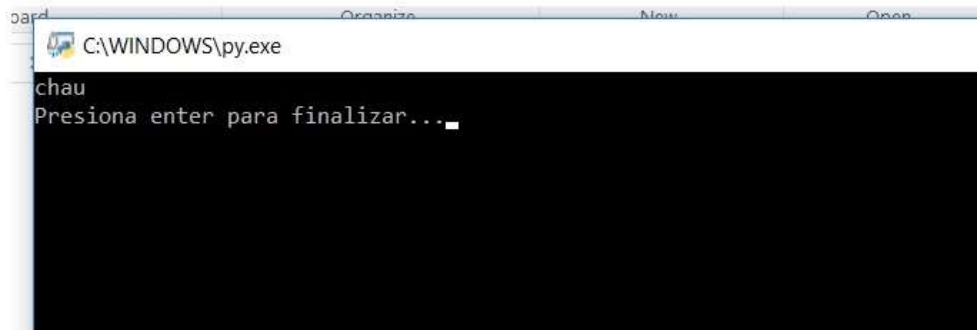
```
>>>
= RESTART: C:\Users\Andy\Documents\fiuba\cursos verano\algo I\codigo\p50.py =
hola
Presiona enter...
chau
Presiona enter para finalizar...
>>> |
```

Pero al correrlo desde el archivo (haciendo clic en el archivo correspondiente):

1ra parte



2da parte



En la segunda parte, luego de presionar el enter, vemos que borró lo que imprimió con anterioridad.

¿Cómo hacer para que corra tanto en Linux como en Windows?

La instrucción `os.name` nos indica en qué tipo de sistema operativo estamos. De esta manera, podemos elegir por una opción u otra.

Para hacerlo más prolijo armamos una función limpiar y simplemente, cuando deseemos limpiar la pantalla, invocamos a esta función.

```
def limpiar():  
    # estamos en Linux  
    if os.name == "posix":  
        os.system("clear")  
    # estamos en Windows  
    elif os.name in ("ce", "nt", "dos"):  
        os.system("cls")
```

Cadenas de caracteres (strings)

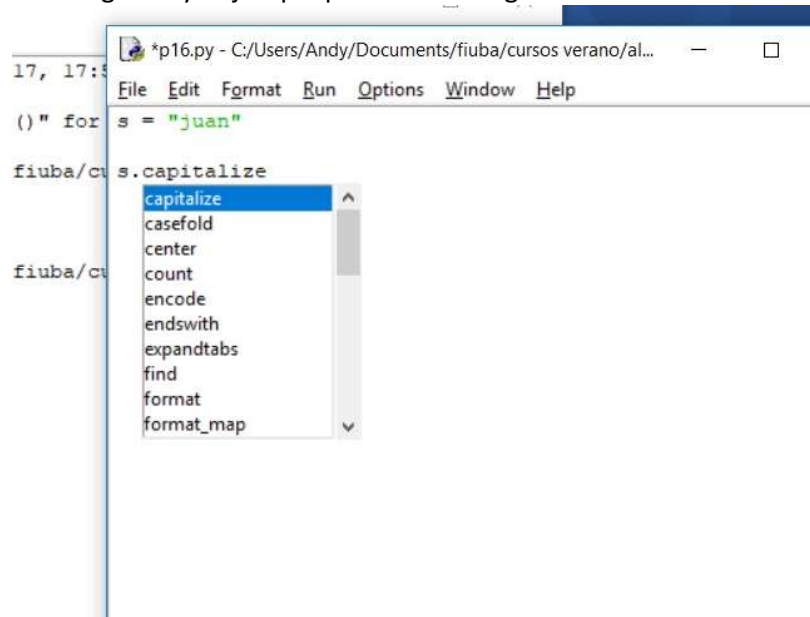
Python permite hacer cosas de muchas maneras diferentes. Por ejemplo, para invertir una lista *lis* podemos escribir

```
lis.reverse()
```

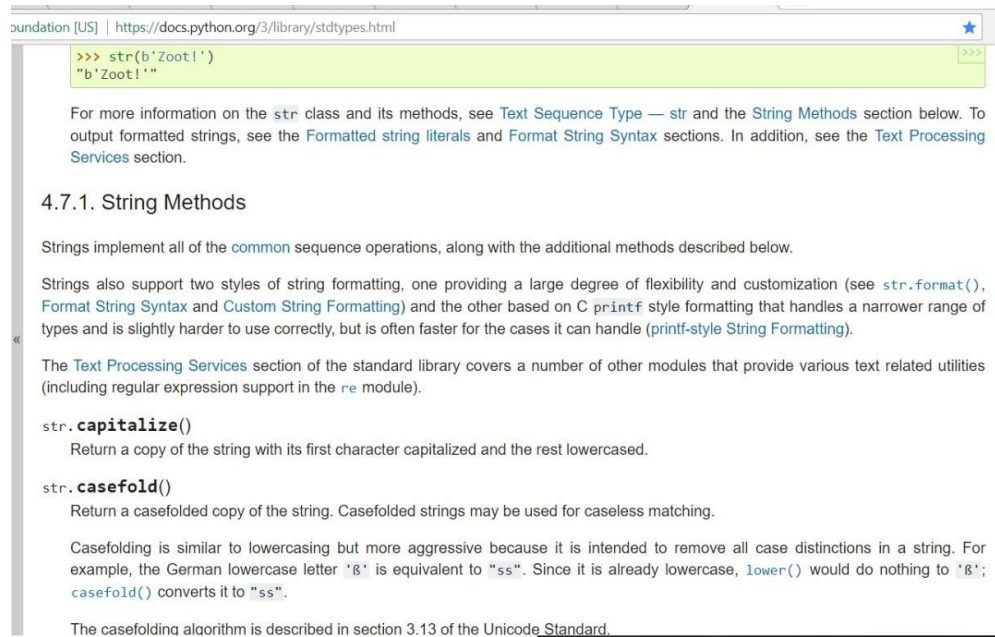
pero también se invierte escribiendo

```
lis = lis[::-1]
```

La finalidad de este apunte no es explicar las mil maneras diferentes de hacer lo mismo, porque el texto sería muy largo y lo único que conseguiría es marearlos. Esto mismo sucede con la clase string: tiene asociado más de 40 métodos. La idea no es verlos todos, sino mencionar algunos y dejar que puedan investigar otros.



La documentación de todos los métodos la pueden encontrar en el mismo sitio oficial de Python: <https://docs.python.org/3/library/stdtypes.html>



Por ejemplo el método `capitalize` cambia la primera letra por una mayúscula.

- Seguramente, más interesante es el método `count` al que le pasamos un carácter o una secuencia de caracteres y nos devuelve cuántas veces la encontró en el string.
- El método `find` nos indica en qué posición comienza una serie de caracteres que le indicamos por parámetro. Si no la encuentra devuelve `-1`.
- Hay otros métodos para saber si el carácter es un dígito, si es alfanumérico, etc. Hay un método “`split`” que lo vamos a usar mucho.

Tuplas y Listas

En Python una tupla y una lista son estructuras muy similares. Ambas asemejan lo que en Pascal sería un registro y lo que en C o Java sería una estructura. La diferencia entre una tupla y una lista es que la primera no puede modificar los datos que tiene y la segunda sí.

Por ejemplo:

```
tupla = (123, "juan perez", 18000)
```

puede representar una tupla (un registro) de un trabajador (“juan perez”) cuyo legajo es el 123, y tiene un sueldo de \$18.000.

La tupla se define con paréntesis. Cada campo se lo referencia por una posición: la primera es la 0 (cero), la segunda la 1 (uno), etc.

Entonces, si hacemos

```
print(tupla[0])
```

imprime 123.

Dato	123	juan perez	18000
Posición	0	1	2

Pero no podemos modificar ningún campo. Es decir, los valores son fijos. Por ejemplo, si queremos modificarle el sueldo y escribimos:

```
tupla[2] = 20000
```

Nos da un error. En cambio, con una lista no tendríamos problema:

```
lista = [123,"juan perez",18000]  
lista[2] = 20000
```

La diferencia al definir una tupla y una lista es que la primera se define con paréntesis y la segunda con corchetes.

Si la modificación del campo sueldo la hubiéramos hecho en una función la lista saldría modificada.

```
def aumentar_sueldo(lista, valor):  
    lista[2] = valor
```

```
aumentar_sueldo(lista, 20000)
```

En la sección de funciones habíamos visto que las listas se pasan por referencia en una función, por lo tanto, si esta variable lista la pasamos a una función en donde la modificamos, la lista queda cambiada. Pero hacía falta aclarar algo más.

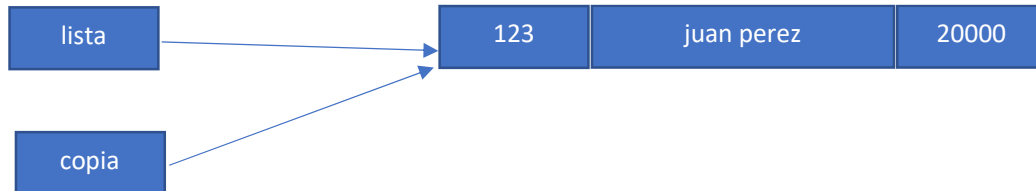
- lista se encuentra en alguna dirección, supongamos en la 1000.
- Pero los valores que tiene: 123, "juan perez" y 18000 no están en esa dirección. Es decir, no están adentro de lista, se encuentran en un lugar aparte. Por ejemplo, en la dirección 2000.
- En la dirección 1000, lo que hay, entre otras cosas, es la dirección 2000.

Entonces, cuando lista se pasa como parámetro en una función, se copian sus valores en otra dirección, supongamos la 3000. Pero al copiar lo valores copia el 2000, por eso sale de la función con los datos modificados.

Con un gráfico se entenderá mejor.



Paso 1: antes de ingresar en la función



Paso 2: lista se copia al ingresar en la función. Lista y copia son distintas pero ambas apuntan al mismo lugar, por eso modifica los datos.

Las tuplas tienen asociados solo un par de métodos, y como no podemos modificarlas las utilizaremos en muy pocas ocasiones.

En cambio, las listas las usaremos bastante, ya que podemos simular el manejo de un registro (como en el ejemplo de antes), pero también de un vector, o una matriz, o una tabla: vectores de registros.

Veamos varios ejemplos:

Ejercicio 6. Se desea guardar las notas de un examen en una lista. La carga la realiza un usuario desde el teclado, cuando ingresa un 0 termina la carga de notas.

Solución

Empezamos con una lista vacía.

```
notas = []
```

Con el método *append* vamos agregando elementos a la lista. Aprovechemos que vimos funciones para utilizarlas. Definimos una función para la carga de las notas en la lista.

```
def cargar(notas):
    nota = 1 #para que pueda entrar en el ciclo
    while (nota != 0):
        nota = int(input("Ingrese una nota, con 0 finaliza: "))
        if (nota != 0):
            notas.append(nota)
```

```
notas = []
cargar(notas)
```

- Una vez que cargamos las notas podríamos querer calcular su promedio, simplemente haríamos:

```
print("el promedio es: ", sum(notas)/len(notas))
```

La función `sum` hace la sumatoria de la lista de notas y `len` nos dice el tamaño o cantidad de elementos que tiene.

Un ejemplo de una corrida sería:

```
Ingrese una nota, con 0 finaliza: 7
Ingrese una nota, con 0 finaliza: 5
Ingrese una nota, con 0 finaliza: 3
Ingrese una nota, con 0 finaliza: 0
el promedio es: 5.0
```

- Podríamos querer contar cuántos se sacaron 4.

```
print(notas.count(4))
```

`count` cuenta la cantidad de apariciones del parámetro que le pasemos. Nótese que tanto `len` como `sum` no son métodos, son funciones. Es decir, no escribimos `notas.sum`, escribimos `sum` y como parámetro le pasamos `notas`.

En cambio, `count` es un método de lista.

- Si quisiéramos tener la lista ordenada, lo podemos hacer de dos maneras distintas:
 1. Utilizando la función `sorted` que no modifica a la lista original, sino que devuelve una nueva lista ordenada.
 2. Usando el método `sort` que sí modifica la lista original y no devuelve nada.

Usando sorted

```
notas_ordenadas = sorted(notas)
```

Usando sort

```
notas.sort()
```

Tanto `sorted` como `sort` ordenan, por defecto, de menor a mayor. Para ordenarlo de mayor a menor se puede ingresar un nuevo parámetro: `reverse = True`.

```
notas.sort(reverse = True)
```

- Agregar elementos:
 - Vimos que con `append` agrega al final.
 - Con `insert` agrega el valor en la posición que le indiquemos.

Ejemplo:

```
notas.insert(2, 8)    # inserta un 8 en la posición 2
```


- Quitar elementos:
 - Con el método `clear` quita todos los elementos y deja la lista vacía.
 - Con `remove` quita un valor de la lista. Por ejemplo `remove(3)` quitaría una nota 3. Si hay más de una solo quita la primera. Si no hay ninguna da error, por eso este método debe usarse luego del método `count`. Si hay cantidad positiva, entonces se puede remover, de lo contrario, no.
 - Con `pop` quita el elemento que esté en la posición indicada. Por ejemplo `pop(2)` quitaría el elemento que está en la posición 2. Si no le pasamos ningún parámetro, por defecto quita el último elemento de la lista.

Otros métodos

- `copy`. Copia la lista en otra.
- `index`. Devuelve la posición en la que está cierto elemento. Si hay más de uno devuelve la primera. Si no hay ninguno da error, por eso se aconseja usar en conjunto con `count`.
- `reverse`. Invierte la lista.

Nos falta ver el método `extend` que es importante.

Supongamos que hay dos docentes que corrigieron exámenes y tienen dos listas de notas, por ejemplo

```
notas_1 = [5, 4, 9, 7]
notas_2 = [2, 2, 4]      # el segundo docente es más exigente!
```

Si queremos unir las dos listas debemos usar el método `extend`.

```
notas_1.extend(notas_2)
print(notas_1)
```

La salida será:

```
[5, 4, 9, 7, 2, 2, 4]
```

¿Qué hubiera pasado si usábamos `append`?

```
notas_1.append(notas_2)
print(notas_1)
```

Al ver la salida vemos lo siguiente:

```
[5, 4, 9, 7, [2, 2, 4]]
```

Lo que hizo fue agregar toda la segunda lista como un solo elemento, en lugar de agregar 3 elementos más, agregó 1 solo. Hay que tener cuidado cuando se usan estos métodos que pueden prestar a confusión.

Recorriendo una lista

Lo más probable es que tengamos que recorrer una lista elemento por elemento para hacer cálculos o buscar algún dato, entre otras cosas.

Lo primero que podríamos pensar es hacer un rango desde 0 hasta la última posición de la lista, en donde vamos obteniendo cada uno de los elementos. Esto lo hacíamos poniendo el nombre de la lista y entre corchetes la posición.

De esta forma recorreremos toda la lista y la imprimimos elemento por elemento.

```
for i in range(len(notas)):
    print(notas[i])
```

- Al no indicarle la posición de arranque, por defecto toma la posición 0.
- `len(notas)` devuelve el tamaño de la lista. Por ejemplo, si la lista tiene 8 elementos, la última posición es la 7, pero hay que recordar que en `range` el límite superior lo deja afuera.

Sin embargo, hay otra forma más simple de recorrer una lista, usando lo que llamamos iteradores.

```
for nota in notas:
    print(nota)
```

En este caso *nota* es un iterador que va tomando los elementos de la lista de a uno por vez. Como se ve, este código es más sencillo y más fácil de comprender que el anterior. Casi todos los recorridos que hagamos los haremos con iteradores.

Slices – sublistas

Los slices nos van a permitir quedarnos con ciertas porciones de una lista o string. Veamos un poco más sobre los subíndices.

Supongamos que tenemos la siguiente lista:

```
lista = [5, 4, 9, 7, 3, 10, 8]
```

Los subíndices irán desde 0 hasta 6, pero también podemos trabajar con subíndices negativos. El -1 es para el último elemento, el -2 para el penúltimo, etc.

Datos	5	4	9	7	3	10	8
Subíndices positivos	0	1	2	3	4	5	6
Subíndices negativos	-7	-6	-5	-4	-3	-2	-1

Con las siguientes instrucciones formamos nuevas listas a partir de la original.

```
lista = [5, 4, 9, 7, 3, 10, 8]
```

```
# corta la lista desde la posición 2 hasta la 4 inclusive
lista2 = lista[2 : 5]

# corta la lista desde la posición 2 hasta el final
lista3 = lista[2 : ]

# corta la lista desde el principio hasta la posición 1 inclusive
lista4 = lista[ : 2]

# corta la lista desde la penúltima posición hasta el final
lista5 = lista[-2 : ]

# corta la lista desde la posición 1 hasta la 5 inclusive
# saltando de a dos elementos: posición 1, 3, 5
lista6 = lista[ 1 : 6 : 2]

# arma una lista con el 1er valor, luego salta al 4to, luego al
7mo.
lista7 = lista[ : : 3]

# arma una lista desde la posición 5 hasta la 3ra inclusive
# yendo de a uno para atrás
lista8 = lista[ 5 : 2 : -1]

# devuelve la lista invertida
lista9 = lista[ : : -1]
```

Al imprimir estas listas nos queda:

```
lista2 = [9, 7, 3]
lista3 = [9, 7, 3, 10, 8]
lista4 = [5, 4]
lista5 = [10, 8]
lista6 = [4, 7, 10]
lista7 = [5, 7, 8]
lista8 = [8, 10, 3, 7]
```

Listas y strings

Hay ciertas funciones y métodos que convierten un string en una lista y viceversa. Veamos algunos.

split

```
s = "hola que tal"
l = s.split()
```

Arma una lista con las palabras, usando uno o más blancos como separador. La lista queda:

```
['hola', 'que', 'tal']
```

Usando otro separador

```
s = "hola,que,tal"
l = s.split(',')
```

Al indicar algún separador, en este ejemplo la coma, lo utiliza para hacer el split (la división). La lista queda igual que la anterior.

Ordenar alfabéticamente

```
s = "camino"
l = sorted(s)
print(l)
```

En este caso sorted devuelve una lista ordenada alfabéticamente:

```
['a', 'c', 'i', 'm', 'n', 'o']
```

Si queremos volver a armar un string con esta lista podríamos hacer lo siguiente:

```
s2 = ""
for letra in l:
    s2 = s2 + letra
```

Armamos un string vacío, recorremos la lista y le vamos agregando con el + cada letra. El nuevo string queda con las letras de “camino” ordenadas en forma alfabética:

```
acimno
```

Hay otra forma de unir una lista en un string, usando el método join (unir). Para hacer lo mismo que antes con este método escribimos:

```
s2 = "".join(l)
```

El "" arma un string vacío, y la instrucción join indica que debe unir el parámetro con ese string, como es vacío solo une los elementos de la lista.

Si hubiéramos puesto

```
s3 = "-".join(l)
```

El nuevo string queda unido con guiones:

```
a-c-i-m-n-o
```

Matrices

En Python no tenemos las estructuras array como en otros lenguajes, por lo que tanto los vectores como las matrices las trabajamos con listas. Precisando más, los vectores los trabajamos con listas simples, las matrices de dos dimensiones con listas de listas, las de tres, con listas de listas de listas, etc.

Ejercicio 7. Un negocio trabaja de lunes a viernes vendiendo tres productos. Se desea cargar en una matriz las cantidades vendidas de cada uno por cada día. Luego, en base a los datos, calcular algunos totales: cantidad vendida de un producto durante toda la semana, cantidad vendida de todos los productos en un día, etc.

Estos datos los podemos representar en una matriz de 5 x 3 o de 3 x 5. Ejemplo:

10	15	8	12	24
2	5	3	4	6
20	18	9	22	35

El 4 que está en la segunda fila y la cuarta columna, indica que del producto 2, el día jueves, se vendieron 4 unidades. Si quisiéramos sacar los totales vendidos del segundo producto deberíamos sumar todos los elementos de la segunda fila: $2 + 5 + 3 + 4 + 6 = 20$.

Si quisiéramos indicar cuántos artículos en total se vendieron el día martes, deberíamos sumar todos los elementos de la segunda columna: $15 + 5 + 18 = 38$.

¿Cómo representamos esa matriz en Python?

Sería una lista con tres listas, cada una con 5 elementos.

```
ventas = [ [10, 15, 8, 12, 24],
            [2, 5, 3, 4, 6],
            [20, 18, 9, 22, 35] ]
```

Es importante notar la diferencia entre estas tres impresiones:

```
print(ventas)           # imprime toda la matriz
print(ventas[0])        # imprime la primera fila
print(ventas[2][3])     # imprime el valor de la fila 2 columna 3
```

La salida de cada una es:

```
[[10, 15, 8, 12, 24], [2, 5, 3, 4, 6], [20, 18, 9, 22, 35]]
[10, 15, 8, 12, 24]
22
```

Si queremos obtener el total de ventas del segundo artículo, simplemente hacemos

```
print(sum(ventas[1]))
```

En cambio, para saber la cantidad de productos que se vendieron el día jueves:

```
total_jueves = 0
for articulo in ventas:
    total_jueves += articulo[3]

print("total vendido el dia jueves: ", total_jueves)
```

La salida será:

```
total vendido el dia jueves: 38
```

Tablas (vectores de registros)

La palabra “tabla” en informática se refiere a tabla de datos, la cual puede estar en un archivo o en alguna estructura en memoria, generalmente un vector de registros. Nuevamente, en Python lo manejaremos con una lista de listas (también se pueden usar diccionarios, lo veremos luego).

Por ejemplo, la siguiente lista representa los datos de un empleado:

legajo	nombre	sueldo
--------	--------	--------

Pero en una empresa podemos tener cientos (o miles) de empleados por lo que este registro se repetirá tantas veces como sea necesario con los distintos valores para cada empleado.

Ejemplo:

123	Pérez Juan	18000
87	González Rosa	22000
145	Campos Silvia	20000
...		

Esto lo podemos representar, como dijimos, con una lista de listas:

```
empleados = [ [123, "Pérez Juan", 18000],
               [87, "González Rosa", 22000],
               [145, "Campos Silvia", 20000] ]
```

La estructura es similar a la de una matriz con la diferencia de que cada fila representa los datos de un empleado.

A los datos de un registro (una fila) los llamamos “campos”. En este ejemplo tendríamos el campo “legajo” que es la primera posición, el campo “nombre”, la segunda, y el campo “sueldo”, la tercera.

¿Cómo ordenar según los distintos campos?

Si la lista la ordenamos usando `sorted` o `sort` la ordenará por defecto sobre el primer campo, en este caso por legajo.

```
empleados.sort()

for empleado in empleados:
    print(empleado)
```

La salida quedará:

```
[87, 'González Rosa', 22000]
[123, 'Pérez Juan', 18000]
[145, 'Campos Silvia', 20000]
```

Para ordenar por otros campos necesitamos usar funciones lambda.

Funciones lambda

Una función lambda es una expresión anónima de una línea. Decimos anónima porque no tienen asociado un nombre como las demás funciones, y se utilizan para pequeñas funciones cuyo código es muy simple.

El código es

```
lambda    parámetros:    devolución (sin return)
```

Por ejemplo, la función sumar sería así:

```
# mediante una función no anónima
def sumar(x, y):
    return x + y

# mediante una función lambda
suma = lambda x, y: x + y

# el llamado lo hacemos a través de la variable suma
print(suma(5, 3))
```

En el código resaltamos en azul los parámetros y en verde la devolución. Nótese que esta devolución está implícita en las funciones lambda, por lo que no hace falta la palabra `return`.

Además, no tiene nombre, `suma` sería una variable a la que se le asigna la función para poder ejecutarla.

¿Cómo utilizar estas funciones lambda para ordenar listas?

En el ejemplo de la lista de empleados, si quisiéramos ordenarla por algún otro campo, por ejemplo, por orden apellido, deberíamos indicarle que ordene por el segundo campo, es decir por el campo que está en la columna 1.

```
empleados.sort(key = lambda x: x[1])
```

Con el método sort le indicamos que ordene, pero en key le decimos que lo haga por la columna 1. La x representaría uno de los registros, podríamos haber utilizado cualquier nombre.

La impresión quedará:

```
[145, 'Campos Silvia', 20000]
[87, 'González Rosa', 22000]
[123, 'Perez Juan', 18000]
```

Y si quisiéramos ordenarlo por sueldo, de mayor a menor:

```
empleados.sort(key = lambda reg: reg[2], reverse = True)

for empleado in empleados:
    print(empleado)
```

La salida será:

```
[87, 'González Rosa', 22000]
[145, 'Campos Silvia', 20000]
[123, 'Perez Juan', 18000]
```

Diccionarios

La estructura anterior podríamos guardarla en un diccionario. Un diccionario será como una lista de registros, con la diferencia de que uno de los campos actuará como clave.

¿Qué es una clave?

Una clave es un campo o más que determinan de forma unívoca a un registro. Por ejemplo, en los datos de un empleado la clave sería el legajo, ya que los nombres se pueden repetir y los sueldos también.

Si los mismos datos los pusiéramos en un diccionario, el campo legajo sería la clave y el nombre y el sueldo serían los campos que están asociados a dicha clave.

Pero empecemos con un ejemplo más sencillo: legajo – nombre (sin el sueldo).

```
empleados = { 123 : "Perez Juan",
              87  : "González Rosa",
              145 : "Campos Silvia" }
```

Un diccionario se define con llaves en lugar de corchetes como una lista. Se coloca el valor de la clave, dos puntos, y el valor asociado a dicha clave.

Si quisiéramos ir llenando el diccionario a medida que se ingresan los datos por teclado, podríamos hacer una función cargar de esta forma:

```
def cargar(empleados):
    legajo = 1      # para ingresar al ciclo
    while (legajo != 0):
        legajo = int(input("Ingrese el número de legajo, con cero
sale: "))
        if (legajo != 0):
            nombre = input("Ingrese el nombre: ")
            # se agrega la clave con sus datos al diccionario
            empleados[legajo] = nombre
```

Antes de llamar a la función creamos un diccionario vacío, y luego la llamamos.

```
# se crea un diccionario vacío
empleados = {}

# se carga el diccionario
cargar(empleados)
```

Para imprimir un diccionario, si solo lo recorremos como si fuera una lista:

```
for elemento in empleados:
    print(elemento)
```

La salida serán solo las claves:

```
123
87
145
```

Entonces, para listar todos sus datos, debemos imprimir sus claves y los valores asociados:

```
for legajo in empleados:
    print(legajo, ":", empleados[legajo])
```

En este caso:

```
123 : Perez Juan
87  : Gonzalez Rosa
145 : Campos Silvia
```

Ordenamientos en diccionarios

En general un diccionario no necesitaremos ordenarlo porque nos manejaremos mediante su clave para obtener los valores, pero en caso de necesitar hacerlo, no tienen asociado el método sort como las listas. Podemos usar la función sorted, sabiendo que convierte al diccionario en una lista.

Ejemplos:

Ejemplo 1

```
lista = sorted(empleados)
print(lista)
```

Salida

```
[87, 123, 145]
```

Imprime solo las claves ordenadas.

Ejemplo 2

```
lista = sorted(empleados.values())
print(lista)
```

Salida

```
['Campos Silvia', 'Gonzalez Rosa', 'Perez Juan']
```

Imprime solo los nombres ordenados (values devuelve los valores asociados a las claves).

Ejemplo 3

```
lista = sorted(empleados.items())
print(lista)
```

Salida

```
[(87, 'Gonzalez Rosa'), (123, 'Perez Juan'), (145, 'Campos Silvia')]
```

Imprime todo: legajo y nombre, ordenado por legajo. Arma una lista de tuplas al utilizar el método items.

Ejemplo 4

```
lista = sorted(empleados.items(), key = lambda x : x[1])
print(lista)
```

Salida

```
[(145, 'Campos Silvia'), (87, 'Gonzalez Rosa'), (123, 'Perez Juan')]
```

Imprime todo como una lista de tuplas, ordenado por el campo nombre.

Estructuras más complejas con diccionarios

En general no tendremos un solo valor asociado a una clave. El ejemplo anterior no tendría mucho sentido en un problema real. Lo que va a suceder es que a una clave le asociemos muchos valores. Por ejemplo, a la clave DNI de una persona le podemos asociar el nombre, su dirección, fecha de nacimiento, etc. Por lo tanto, la estructura más común que usaremos es la de una lista de valores asociados a una clave.

Ejercicio 8. Se van cargando por teclado legajos de alumnos con notas, de manera desordenada. Cada legajo puede aparecer más de una vez en la secuencia de ingreso. Se desea indicar el promedio final para cada alumno.

Solución

Este problema se puede solucionar también con listas, pero lo haremos con diccionarios. La clave será lógicamente el número de legajo, luego necesitamos asociarle dos datos para poder sacar el promedio: suma total de las notas y cantidad de notas. Por supuesto que se podía tener una lista de notas solamente, sin la suma. Sin embargo, es más eficiente guardar solo dos datos que una gran cantidad de ellos. Todo depende del tipo de problema: si sabemos que las notas son de una sola materia, no serán más de dos o tres, entonces es más simple guardar la lista de notas directamente. Pero si estas fueran las de toda la carrera, podrían ser diez, quince o treinta, es mejor guardar la suma y la cantidad (dos datos contra quince o treinta).

El código para la carga podría ser:

```
""" coloca una nota en el diccionario
    si el legajo ya se encuentra, la suma a la anterior
    y cuenta una nota más.
    Si el legajo no está lo agrega al diccionario
    y anota que hay una sola nota """
def poner_nota(dic, leg, nota):
    if (leg in dic):
        dic[leg][0] += nota
        dic[leg][1] += 1
    else:
        dic[leg] = [nota, 1]

""" pide el legajo y la nota,
    carga el diccionario mientras leg no sea cero """
def cargar(dic):
    leg = 1
    while (leg != 0):
        leg = int(input("Ingrese el legajo, con 0 sale: "))
        if (leg != 0):
            nota = int(input("Ingrese nota: "))
            poner_nota(dic, leg, nota)
```

```
# programa principal
dic = {}
cargar(dic)
print(dic)
```

Lo único nuevo en este código es cuando consultamos si el legajo ya se encuentra en el diccionario, con la pregunta *in*.

```
if (leg in dic):
```

Si el legajo ya se encuentra, esta consulta dará verdadero, de lo contrario dará falso. Hay que notar que `dic[leg]` a secas asocia una lista a la clave: sumatoria de notas y cantidad. A su vez, para ir a cada posición de dicha lista debemos agregar un nuevo corchete con la posición correspondiente.

Si ingresamos estos datos como ejemplo:

```
Ingrese el legajo, con 0 sale: 123
Ingrese nota: 5
Ingrese el legajo, con 0 sale: 80
Ingrese nota: 6
Ingrese el legajo, con 0 sale: 123
Ingrese nota: 9
Ingrese el legajo, con 0 sale: 0
```

La salida será:

```
{123: [14, 2], 80: [6, 1]}
```

Vemos que para el legajo 123 sumó sus dos notas: 5 y 9, y contabilizó dos notas en total. Si queremos sacar el promedio, lo único que tenemos que hacer es recorrer el diccionario y realizar la división entre el primer campo y el segundo.

```
print("Los promedios son: ")
for leg in dic:
    print("Legajo:      ",      leg,      "      -      promedio:      ",
          dic[leg][0]/dic[leg][1])
```

Cuya salida será:

```
Los promedios son:
Legajo:  123  - promedio:  7.0
Legajo:  80  - promedio:  6.0
```

Hay otra forma de cargar el diccionario sin necesidad de preguntar si el legajo ya se encuentra: utilizando el método `setdefault`.

```
def poner_nota(dic, leg, nota):
    dic.setdefault(leg, [0, 0])
    dic[leg][0] += nota
    dic[leg][1] += 1
```

Este método hace lo siguiente: si el legajo existe, no hace nada. Si el legajo no existe, lo da de alta junto con los valores que les pasamos (una lista con ceros).
Luego hacemos la sumatoria de la nota y cargamos una nota más en cantidad.

Un poco de maquillaje

Cortamos el hilo de lo que veníamos viendo para dedicarnos apenas a prestar atención a cómo mostramos la información.

Es cierto que, en general, las cosas importantes no son las que se ven a simple vista. En el tema que nos ocupa, podríamos tener una aplicación visualmente muy agradable, con colores bien balanceados, un tipo y tamaño de letra cómodo, etcétera, pero si realiza mal los cálculos, no sirve para nada.

Sin embargo, el aspecto visual es lo que primero nos impacta, y un cliente podría decidir contratarnos o no dejándose llevar por esta impresión. Además, una salida por pantalla descuidada puede llevarnos a confusiones.

Por ejemplo, si queremos imprimir la siguiente lista de empleados:

```
empleados = [ [80, "Juan Carlos Rodríguez", 9720.35],  
               [178, "Rosa Rojo", 19520.85],  
               [1024, "Pedro Pablo Carmona González", 29255.70]  
             ]
```

Y escribimos el siguiente código:

```
print("Listado de empleados -----")  
print("Legajo - Nombre y Apellido - Sueldo")  
for empleado in empleados:  
    print(empleado)
```

La salida quedará:

```
Listado de empleados -----  
Legajo - Nombre y Apellido - Sueldo  
[80, 'Juan Carlos Rodríguez', 9720.35]  
[178, 'Rosa Rojo', 19520.85]  
[1024, 'Pedro Pablo Carmona González', 29255.7]  
>>>
```

Lo cual no es solo desagradable a la vista, sino que se presta a confusiones, ya que la información no está encolumnada.

Vamos a ver unas pocas bondades del método *format* de la clase *str*.

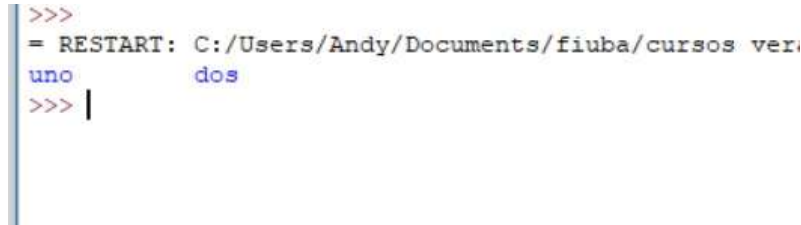
Por ejemplo, tenemos dos strings:

```
s1 = "uno"  
s2 = "dos"
```

Y queremos imprimirlos utilizando 10 posiciones para cada uno, alineados del lado izquierdo. Escribimos:

```
print("{:<10} {:<10}".format(s1, s2))
```

Y la salida será:

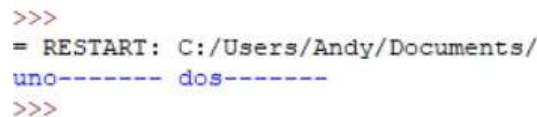


```
>>>
= RESTART: C:/Users/Andy/Documents/fiuba/cursos ver
uno      dos
>>> |
```

En la imagen no se nota que para el segundo string está dejando 10 lugares, pero si le indicamos que rellene el espacio utilizado, se verá mejor:

```
print("{: <-10} {: <-10}".format(s1, s2))
```

Ahora la salida será:



```
>>>
= RESTART: C:/Users/Andy/Documents/
uno----- dos-----
>>>
```

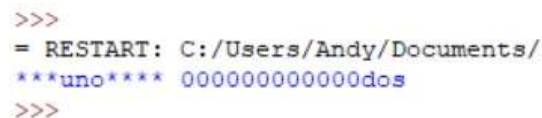
Entonces, entre llaves indicamos el formato para cada parámetro. A su vez va entre comillas, ya que estamos armando un string. Luego se escribe punto, y el método format con sus parámetros, es decir, con lo que se quiere imprimir.

Dentro de cada llave escribimos dos puntos y la cantidad de espacios que queremos que se usen para su impresión. Un símbolo menor indica que se justifique del lado izquierdo, un mayor del lado derecho y un “^” para centrarlo. El signo que esté entre el dos puntos y el de justificación se imprime.

Por ejemplo:

```
print("{:*^10} {:0>15}".format(s1, s2))
```

El primero, centrado en diez lugares, rellena con *. El segundo, justificado del lado derecho y rellena con 0.



```
>>>
= RESTART: C:/Users/Andy/Documents/
***uno*** 0000000000000dos
>>>
```

¿Qué pasa si les damos menos lugares que el tamaño que necesita para imprimir el dato?

Por ejemplo:

```
print("{:2} {:2}".format(s1, s2))
```

En este caso utiliza el tamaño real, como si no hubiéramos puesto nada.

```
= RESTART: C:/Us
uno dos
>>>
```

Si queremos que trunque la impresión, tenemos que agregarle un punto luego del dos puntos:

```
print("{:.2} {:.2}".format(s1, s2))
```

La salida será:

```
un do
>>>
```

Una combinación de ambas cosas:

```
print("{:-<10.2} {:<-10.2}".format(s1, s2))
```

Utiliza 10 lugares, justifica del lado izquierdo, rellena con un guión, pero además trunca en dos lugares.

```
un----- do-----
>>>
```

Para formatos numéricos se utiliza la “d” para los enteros y la “f” para los flotantes. Con algunos ejemplos se verá cómo utilizarlos.

```
x = 51
y = 1.41421356
```

```
print("{:5d} {:6.2f}".format(x, y))
```

Utiliza 5 lugares para el entero y seis lugares para el flotante, de los cuales dos son para la parte decimal. Por defecto alinea del lado derecho. Para alinear del lado izquierdo hay que poner un símbolo menor.

```
= RESTART: C:/Users/Andr
51 1.41
>>>
```

Con un 0 rellena los espacios.

```
print("{:05d} {:06.2f}".format(x, y))
```

Salida:

```
00051 001.41
```

Entonces, el ejemplo inicial, en donde queremos encolumnar todos los datos de los empleados, lo podríamos hacer:

```
print("{:-^42}".format("LISTADO DE EMPLEADOS"))
print()
print("{:<8}   {:<24}   {:^8}".format("Legajo", "Nombre y Apellido",
"Sueldo"))
print()
for emp in empleados:
    print("{:<8d} {:<24.22} {:.2f}".format(emp[0], emp[1], emp[2]))

input("Ingrese enter...")
```

Y la salida será:

```
-----LISTADO DE EMPLEADOS-----

Legajo   Nombre y Apellido           Sueldo
80        Juan Carlos Rodríguez      9720.35
178       Rosa Rojo                  19520.85
1024      Pedro Pablo Carmona Go     29255.70
Ingrese enter...
```

Como apreciamos, más agradable a la vista y legible.