

# Ficha 1

## Fundamentos de Programación

### 1.] Breve referencia histórica: el advenimiento de las computadoras.

Resulta difícil objetar que la segunda mitad del siglo XX (y lo que va del siglo XXI) se vio fuertemente influenciada por la aparición de uno de los inventos más revolucionarios de la historia de la civilización: la computadora<sup>1</sup>. Y es que este aparato no sólo revolucionó la vida moderna, sino que en gran medida se ha convertido también en el sostén de la misma: hoy en día las computadoras se usan en casi cualquier actividad que admita alguna clase de automatización, y cada vez con más frecuencia también en tareas que otrora eran consideradas "exclusivas" (y distintivas) de la especie humana, como la creación de arte (pictórico, literario, musical), la interpretación de textos y el procesamiento de voz e imágenes, por citar sólo algunos ejemplos comunes.

Por otra parte, la computadora como concepto tecnológico ha sido desde el primer momento de su aparición un invento que no ha parado de mostrar innovaciones y mejoras, al punto que su evolución ha tomado por sorpresa a técnicos, científicos, académicos y empresarios de todas las épocas desde la década del 40 en el siglo XX, provocando muchas frases desafortunadas y errores notables en cuanto a las predicciones efectuadas respecto del futuro del invento.

Como sea, en aquellos lugares del mundo que han alcanzado cierto grado de desarrollo tecnológico resulta casi imposible imaginar la vida cotidiana sin el uso de computadoras. Se utilizan para control de ventas y facturación, para control de stock, para apoyo en diversas áreas de la medicina, en la guerra y la defensa, en la edición de textos, en educación, en el control de comunicaciones, en navegación aérea, espacial, naval y terrestre, en el diseño arquitectónico, en el cálculo, y por supuesto, en el juego...

De hecho, es tal el impacto que las computadoras tienen en esta vida moderna, que incluso llegó a especularse con las más diversas "debacles" tecnológicas cuando estaba llegando a su fin el siglo XX, y se temía que la tradicional forma de representar fechas usando sólo dos dígitos para indicar el año pudiera causar mundialmente problemas enormes (y graves): si una computadora en un banco tomara el año "02" como "1902" y no como "2002", los resultados en el ámbito de la liquidación de intereses en una cuenta podrían ser inimaginablemente dañinos para la economía mundial, y ni qué hablar del peligro que habría producido un error semejante en el sistema de control de lanzamiento de misiles de una superpotencia, si ésta hubiera tomado el "efecto año 2000" a la ligera. En todo el mundo, miles de horas-hombre fueron destinadas a revisar y corregir ese problema durante los años anteriores al 2000, y finalmente puede decirse que todo transcurrió sin problemas.

<sup>1</sup> Todas las referencias históricas que siguen en esta sección, están basadas en los libros que se citan con los números [1] (capítulo 1), [2] (capítulo 1) y [3] (capítulo 4) de la bibliografía general que aparece al final de esta Ficha.

Las primeras computadoras que pueden llamarse “operativas” aparecieron recién en la década del 40 del siglo XX, y antes de eso sólo existían en concepto, o en forma de máquinas de propósito único que pueden considerarse antecedentes válidos de las computadoras, pero no computadoras en sí mismas. Pero el hecho aparentemente sorprendente es que a pesar del corto tiempo transcurrido desde su aparición como invento aplicable y útil, la computadora ha evolucionado a un ritmo increíble y ha pasado a ser el sustento tecnológico de la vida civilizada... ¡y todo en no más de sesenta años! Esto no es un detalle menor: ningún otro invento ha tenido el ritmo de cambio e innovación que ha tenido la computadora. Si los automóviles (por ejemplo) hubieran innovado al mismo ritmo que las computadoras, hoy nuestros autos serían similares al famoso vehículo volador mostrado en la trilogía de películas de “Volver al Futuro” (quizás exceptuando el “pequeño” detalle de su capacidad para viajar en el tiempo...)

Probablemente la primera persona que propuso el concepto de máquina de cálculo capaz de ser programada para procesar lotes de distintas combinaciones de instrucciones, fue el inglés *Charles Babbage* [1] [2] [3] en 1821, luego ayudado por su asistente *Ada Byron*<sup>2</sup>, también inglesa. Ambos dedicaron gran parte de sus vidas al diseño de esa máquina: Babbage diseñó la máquina en sí misma, sobre la base de tecnología de ruedas dentadas, y Byron aportó los elementos relativos a su programación, definiendo conceptos que más de un siglo después se convertirían en cotidianos para todos los programadores del mundo (como el uso de instrucciones repetitivas y subrutinas, que serán oportunamente presentadas en estas notas). La máquina de Babbage y Byron (que Babbage designó como “*Analytical Engine*” o “motor analítico”) era conceptualmente una computadora, pero como suele suceder en casos como estos, Babbage y Byron estaban demasiado adelantados a su época: nunca consiguieron fondos ni apoyo ni comprensión para fabricarla, y ambos murieron sin llegar a verla construida.

En 1890, en ocasión del censo de población de los Estados Unidos de Norteamérica, el ingeniero estadounidense *Herman Hollerith* [2] diseñó una máquina para procesar los datos del censo en forma más veloz, la cual tomaba los datos desde una serie de tarjetas perforadas (que ya habían sido sugeridas por Babbage para su Analytical Engine). Puede decirse que la máquina de Hollerith fue el primer antecedente que *efectivamente llegó a construirse* de las computadoras modernas, aunque es notable que el único propósito del mismo era la tabulación de datos obtenidos en forma similar al censo citado: no era posible extender su funcionalidad a otras áreas y problemas (cuando una máquina tiene estas características, se la suele designar como “máquina de propósito específico”). La compañía fundada por Hollerith para el posterior desarrollo de esa máquina y la prestación de sus servicios, originalmente llamada *Tabulating Machines Company*, se convirtió con el tiempo en la mundialmente famosa *International Business Machines*, más conocida como *IBM*.

En 1943, con la Segunda Guerra Mundial en su punto álgido, el matemático inglés *Alan Turing* [2] [3] estuvo al frente del equipo de criptógrafos y matemáticos ingleses que enfrentaron el problema de romper el sistema de encriptación de mensajes generado por la máquina *Enigma* usada por los alemanes. Para esto, el equipo de Turing trabajó en secreto para desarrollar una máquina desencriptadora que se llamó *Bombe*, gracias a la cual finalmente lograron romper el código *Enigma*. Sin embargo, cuando los alemanes cambiaron

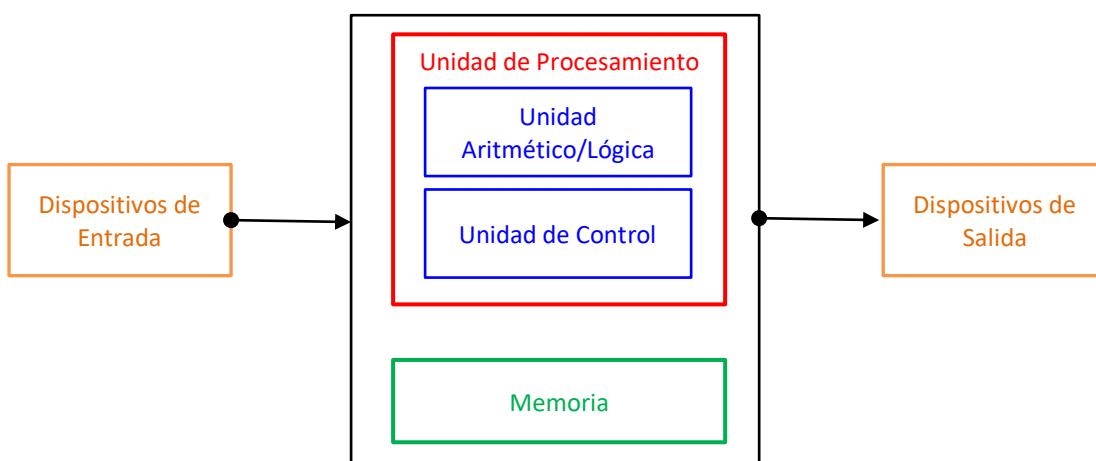
<sup>2</sup> Ada Byron, hija del célebre poeta inglés Lord Byron, era Condesa de Lovelace, y por este motivo suele aparecer en diversos textos y fuentes de consulta como *Ada Lovelace* en lugar de Ada Byron.

el código *Enigma* por el código *Lorenz SZ42*, Turing recurrió a *Tommy Flowers* para que diseñe y construya una nueva máquina, a la cual llamaron *Colossus*. Gracias a esta máquina finalmente se lograron descifrar los mensajes *Lorenz SZ42* y eso definitivamente llevó a los Aliados a la victoria. La historia y el aporte tanto de *Turing* como de *Flowers*, permanecieron en estricto secreto de estado hasta la década de 1970: ni siquiera los familiares y amigos cercanos de ambos estaban al tanto de lo que habían logrado<sup>3</sup>.

La primera máquina que podemos llamar computadora en el sentido moderno, apareció en 1944 de la mano del ingeniero estadounidense *Howard Aiken* y su equipo de colaboradores de la firma IBM. La máquina se llamó *Mark I* y fue construida para la Universidad de Harvard, aunque quedó obsoleta casi en paralelo con su desarrollo, pues usaba conmutadores mecánicos controlados en forma electrónica, mientras que al mismo tiempo otros investigadores ya estaban desarrollando la tecnología de tubos al vacío que derivaría en computadoras totalmente electrónicas. De hecho, en 1945 apareció la *Electronic Numerical Integrator and Calculator*, más conocida como *ENIAC*, que fue la primera computadora electrónica de gran escala. La ENIAC fue diseñada por los ingenieros estadounidenses *John Mauchly* y *Presper Eckert* para la Universidad de Pensilvania [2].

En el mismo proyecto ENIAC trabajó *John von Neumann*, quien realizó aportes para el diseño de un nuevo tipo de máquina con programas almacenados designada como *EDVAC* (*Electronic Discrete Variable and Calculator*) [2]. El diseño de *von Neumann* incluía una *unidad de procesamiento* dividida en una *unidad aritmético-lógica* y una *unidad de control*, además de contar con una *unidad de memoria* para almacenar tanto al programa como a sus datos, y también permitía *periféricos de entrada y salida* (ver Figura 1). Este esquema con el tiempo se convirtió prácticamente en un estándar, al que se conoció como el *Modelo de von Neumann* o también *Arquitectura von Neumann* y muchas computadoras diseñadas desde entonces se basaron en el.

**Figura 1: Esquema general de la Arquitectura von Neumann.**



<sup>3</sup> La historia de *Alan Turing* al frente del equipo que desarrolló la máquina *Bombe*, ha sido llevada al cine en 2014 a través de la película dirigida por *Morten Tyldum* cuyo título original es "The Imitation Game" (y que en Latinoamérica se conoció como "El Código Enigma"). En diversos pasajes de la película pueden verse reproducciones de las máquinas *Enigma* y *Bombe*. La película incluso estuvo nominada al Oscar, así como el actor *Benedict Cumberbatch* que interpretó al propio *Alan Turing*.

A finales de la década de 1950 y principios de la de 1960 los tubos al vacío fueron reemplazados por los transistores, lo que permitió abaratar costos y aumentar potencia. La aparición de los circuitos integrados hacia finales de la década del 60 permitió reunir muchos transistores en un solo circuito, abaratando aún más el proceso de fabricación, y permitiendo reducir cada vez más el tamaño (y no sólo el costo) del computador. Esto abrió el camino que llevó a nuestros días: a finales de la década de 1970 aparecieron las primeras computadoras personales (*Apple II* e *IBM PC*), cuyo costo y tamaño las hacían accesibles para el público promedio... y el resultado está a la vista.

Sin embargo, el hecho que las computadoras hayan ocupado el sitio que ocupan en tan breve plazo, y que evolucionen en la forma vertiginosa que lo hacen, no es en realidad una sorpresa: simplemente se debe al hecho de su propia *aplicabilidad general*. Las computadoras aparecieron para facilitar tareas de cálculo en ámbitos académicos, científicos y militares, pero muy pronto se vio que podrían usarse también en centenares de otras actividades. Invertir en mejorar el invento valía la pena, y el invento mismo ayudaría a descubrir y definir nuevas áreas de aplicación.

## 2.] Algoritmos y programas.

En nuestros días las computadoras conviven con nosotros. Donde miremos las encontraremos<sup>4</sup>. Es común que haya incluso más de una en cada hogar con nivel de vida promedio. Estamos tan acostumbrados a ellas y tan acostumbrados a usarlas para facilitar tareas diversas, que ni siquiera pensamos en lo que realmente tenemos allí. Si se nos pregunta: ¿qué es una computadora? la mayor parte de las personas del mundo dará una respuesta aproximada, intuitivamente correcta, pero casi nunca exacta y completa. Puede decirse que eso mismo ocurrirá con la mayor parte de los inventos que hoy usamos o vemos a diario, y sin embargo hay una sutil diferencia: sabemos qué es un avión, aunque no sabemos bien cómo funciona ni cómo logra volar. Sabemos qué es un lavarropas, aunque no sabemos bien cómo funciona. Lo mismo se puede decir de los automóviles, las heladeras, los relojes, etc.

Pero con las computadoras es diferente. Podemos darnos cuenta que una computadora es un aparato que puede aplicarse para resolver un problema si se le dan los datos del mismo, de forma que procesará esos datos en forma veloz y confiable y nos entregará los resultados que buscamos. Pero el hecho es que nos servirá para *muchos problemas diferentes*, incluso de áreas diferentes. La computadora no es una herramienta de propósito único o limitada a una pequeña gama de tareas. Si la proveemos con el *programa* adecuado, la computadora resolverá cualquier problema. Y aquí está la gran cuestión: podemos ser usuarios de un automóvil sin saber cómo funciona, pues sólo necesitamos que funcione y cumpla su único cometido: trasladarnos donde queremos ir. Pero si queremos que una computadora resuelva los problemas que enfrentamos día a día en nuestro trabajo, o en nuestro ámbito de estudio, o en nuestro hogar, no nos basta con prenderla y limitarnos a verla funcionar. De una forma u otra deberemos cargar en ella los programas adecuados, y a través de esos programas realizar nuestras tareas. Un usuario común de computadoras se limitará a cargar programas ya desarrollados por otras personas. Pero un usuario avanzado, con

<sup>4</sup> Una referencia complementaria general para esta sección, incluye los libros que aparecen con los números [1] (capítulo 14) y [4] (capítulo 1) en la bibliografía disponible al final de esta Ficha.

conocimientos de programación, podrá formular sus propios programas, o desarrollarlos para terceros (percibiendo eventualmente una remuneración o pago por ello...) [4].

La tarea de programar un computador no es sencilla, salvo en casos de problemas o ejemplos muy simples. Requiere inteligencia, paciencia, capacidad de resolver problemas, conocimientos técnicos sobre áreas diversas, creatividad, auto superación, autocritica, y muchas horas de estudio.

El punto de partida para el largo camino en la formación de un programador, es la propia definición del término *computadora*: se trata de un aparato capaz de ejecutar una serie de órdenes que permiten resolver un problema. La serie de órdenes se designa como *programa*, y la característica principal del computador es que el programa a ejecutar puede ser cambiado para resolver problemas distintos.

La definición anterior sugiere que la computadora obtendrá los resultados esperados para un problema dado, pero sólo si alguien plantea y carga en ella el programa con los pasos a seguir para resolverlo. En general, el conjunto finito de pasos para resolver un problema recibe el nombre de *algoritmo* [1] [4]. Si un algoritmo se plantea y escribe de forma que pueda ser cargado en una computadora, entonces tenemos un *programa*.

En última instancia una computadora es lo que conoce como una “*máquina algorítmica*”, pues su capacidad esencial es la de poder seguir paso a paso un algoritmo dado. Notemos que la tarea del computador es básicamente ejecutar un algoritmo para obtener los resultados pedidos, pero el planteo del algoritmo en forma de programa que resuelve un problema es realizado por una persona, llamada *programador*. En ese sentido, las computadoras no piensan ni tienen conciencia respecto del problema cuyos resultados están calculando: simplemente están diseñadas para poder ejecutar una orden tras otra, hasta llegar al final del programa.

Los algoritmos entonces, son parte fundamental en el proceso de programación, pero no se encuentran sólo en el mundo de la programación de computadoras: los algoritmos están a nuestro alrededor, y nos permiten llevar a cabo un gran número de tareas cotidianas: encontramos algoritmos en un libro de recetas de cocina, en un manual de instrucciones, o en los gráficos que nos permiten ensamblar un dispositivo cualquiera, por citar algunos ejemplos. También ejecutamos algoritmos cada vez que hacemos un cálculo matemático<sup>5</sup>, o conducimos un automóvil, o grabamos un programa de televisión.

Los algoritmos tienen una serie de propiedades importantes, que son las que permiten su uso generalizado en tareas que pueden descomponerse en pasos menores. Esas propiedades son las siguientes:

- i. El conjunto de pasos definido para el algoritmo debe tener un *final previsible*. Un conjunto de pasos infinito no es un algoritmo, y por lo tanto no es susceptible de ser ejecutado. La obvia conclusión de esta propiedad es que si un algoritmo está bien planteado, se garantiza que quien lo ejecute llegará eventualmente a un punto en el cual se detendrá. Por ejemplo, una secuencia de pasos diseñada para aplicarse sobre cada uno de los números naturales (los números enteros del 1 en adelante), no es un

---

<sup>5</sup> De hecho, la palabra *algoritmo* proviene del nombre del matemático árabe *al-Jwarizmi*, que en el siglo IX dC planteó las reglas de las cuatro operaciones aritméticas para el sistema decimal. El conjunto de esas reglas se conoció en Europa como "algoritmia" (después "algoritmo"), y se generalizó para designar a cualquier conjunto de reglas para un problema dado.

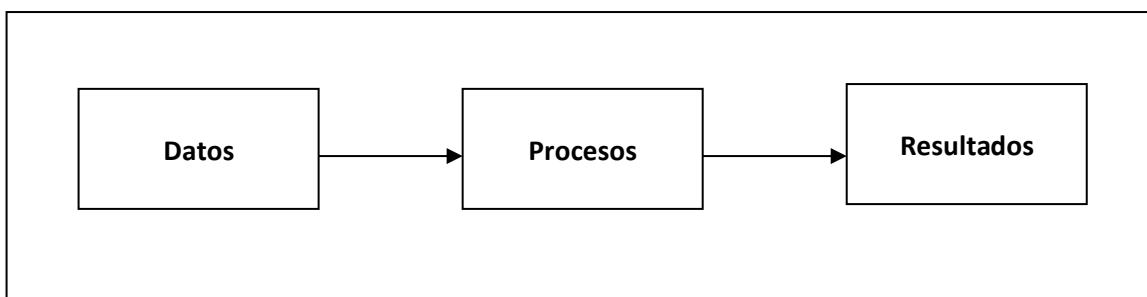
algoritmo pues los números naturales son infinitos, y la secuencia planteada nunca terminaría de ejecutarse.

- ii. Una vez que se plantea un algoritmo para una tarea o problema dado, y se verifica la corrección del mismo, ***no es necesario volver a pensar la solución*** del problema cada vez que se desee obtener sus resultados. Los algoritmos permiten ahorrar tiempo y esfuerzo a quienes los aplican. Por ejemplo, cada vez que sumamos dos números de varios dígitos, aplicamos un conocido algoritmo basado en sumar primero los dígitos de las unidades, luego los de las decenas, y así sucesivamente, acarreando los excesos hacia la izquierda, hasta obtener la suma total. *Sabemos* que ese algoritmo funciona, aunque no nos detenemos a pensar *porqué* funciona.
- iii. Un algoritmo consiste en una combinación de pasos básicos más pequeños, que eventualmente pueden ser ***automatizados***. Esto implica que quien ejecuta un algoritmo no necesariamente debe ser una entidad inteligente (como una persona), sino que también puede ser ejecutado por una máquina, o por un animal convenientemente entrenado. Esa es la “magia” detrás de las computadoras: poseen un componente muy sofisticado llamado ***procesador***, electrónicamente diseñado para realizar ciertas operaciones básicas (esto es, operaciones aritméticas, comparaciones y operaciones lógicas) sobre los datos que recibe. Un programa es una combinación de esas operaciones básicas que el procesador ejecuta una tras otra, a una velocidad muy superior a la del propio cerebro humano.

Desde el momento que un algoritmo es la secuencia de pasos finita que permite resolver un problema, queda claro que un algoritmo opera sobre la base de tres elementos obvios: comienza sobre los *datos* del problema, suponiendo que se aplican sobre ellos los *pasos*, *procesos* o *instrucciones* de los que consta el algoritmo, obteniendo finalmente los *resultados* (ver Figura 2).

Un requisito fundamental en el planteo de un algoritmo, es el conocimiento de cuáles son las *operaciones primarias, básicas o primitivas* que están permitidas de combinar para formar la secuencia de procesos que resuelve el problema. Este detalle subordina fuertemente el planteo. Por ejemplo, supongamos que se nos pide plantear un algoritmo que permita obtener, a cualquiera que lo siga, dibujos aceptables de una cara humana esquematizada. Es obvio que ese algoritmo podría incluir combinaciones de instrucciones tales como “dibuje un óvalo de forma que el diámetro mayor quede en posición vertical” (para representar los contornos de la cara), o como “dibuje dos pequeños óvalos (que asemejen los ojos) dentro del óvalo mayor”. Pero, ¿qué pasaría si entre el juego de operaciones elementales que se suponen permitidas no estuviera contemplada la capacidad de dibujar óvalos? Si el algoritmo estuviera siendo diseñado para ejecutarse por una máquina limitada tipo “plotter” de dibujo, pero sin capacidad para desplazamientos diagonales, el dibujo de curvas sería imposible, o muy imperfecto.

**Figura 2: Estructura de un Algoritmo**



Por otra parte, quien diseña el algoritmo debe analizar el problema planteado e identificar a qué resultados se pretende arribar. Una vez identificado el o los resultados esperados, se procede a identificar cuáles son los datos que se disponen y por último qué acciones o

procesos ejecutar sobre dichos datos para obtener los resultados. Esto ayuda a poner orden en las ideas iniciales, dejando claros tanto el punto de llegada como el de partida. Si no se tienen claros los datos, se corre el riesgo de plantear procesos que luego no pueden aplicarse simplemente por la ausencia de los datos supuestos. Si no se tienen en claro los resultados, se puede caer en el planteo de un algoritmo que no obtiene lo que se estaba buscando... y en el mundo de los programadores, hay pocas cosas que resulten tan frustrantes e inútiles como la solución adecuada al problema equivocado [4].

Finalmente, el algoritmo debe plantearse de forma que pueda ser asimilado y ejecutado por quien corresponda. Si se diseña un algoritmo para ser ejecutado por una persona, podría bastar con escribirlo en un idioma o lenguaje natural, en forma clara y esquemática, aunque siempre asumiendo que la persona que lo ejecutará entiende las instrucciones primitivas de las que está compuesto. En general, una persona entenderá el algoritmo aunque el mismo tuviera alguna imprecisión o falta de claridad.

Pero si el algoritmo se diseña para ser ejecutado por una máquina, entonces debe ser planteado o escrito en términos que hagan posible que el mismo sea luego transferido a la máquina. Si la máquina es una computadora, el algoritmo debe escribirse usando un *lenguaje de programación*, generando así lo que se conoce como un programa [1] [4]. Existen muchos lenguajes de programación. Algunos de los más conocidos son *C*, *C++*, *Basic*, *Pascal*, *Java* o *Python*, siendo este último el que usaremos a lo largo del curso.

Un *lenguaje de programación* es un conjunto de símbolos, palabras y reglas para combinar esos símbolos y palabras de forma muy precisa y detallada. Los lenguajes de programación prevén un cierto conjunto mínimo de palabras válidas (llamadas “*palabras reservadas*”) y un cierto conjunto de símbolos para operaciones generalmente matemáticas y lógicas (llamados “*operadores*”). Pero no cualquier combinación de esos operadores y palabras reservadas resulta en una orden válida para el lenguaje, de la misma forma que no cualquier combinación de palabras castellanas resulta en una frase comprensible en español. Los operadores y palabras reservadas deben combinarse siguiendo reglas específicas y muy puntuales. El conjunto de esas reglas se conoce como la *sintaxis* del lenguaje. Por ejemplo, en el lenguaje *Python* se puede preguntar si una proposición dada es cierta o falsa por medio de una *instrucción condicional* que se forma con las palabras reservadas *if* y *else*, junto con algunos signos especiales los dos puntos (:). Para escribir la condición a evaluar se pueden usar operadores como <, >, >=, etc. Pero la *sintaxis* o *conjunto de las reglas del lenguaje* exige que esas palabras, signos especiales y operadores se escriban de una forma determinada para que la instrucción condicional sea válida. Una forma válida de escribirla sería:

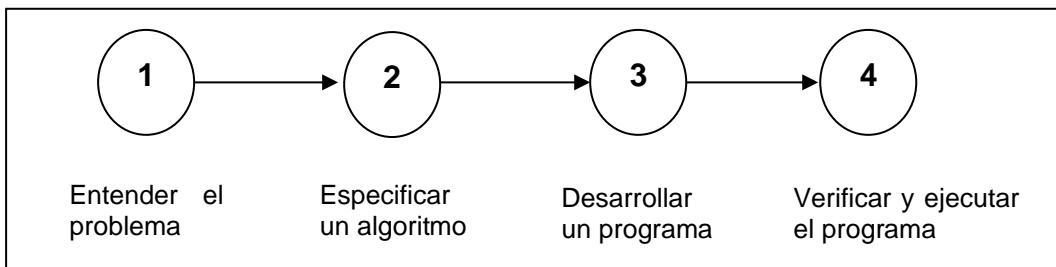
```
if x >= 10:  
    # realizar aquí alguna tarea, a ejecutarse  
    # si fuera cierto que x >= 10  
else:  
    # realizar aquí alguna otra tarea,  
    # a ejecutarse si no fuera cierto que x >= 10
```

Sin embargo, la siguiente combinación de las mismas palabras, símbolos y operadores (que originalmente estaban en color azul en el modelo anterior) es de *sintaxis totalmente incorrecta en Python* (y muy posiblemente también en cualquier otro lenguaje!):

```
else x: = 10 >  
: if
```

En definitiva, para que un computador pueda ser usado para resolver un problema, los pasos a seguir se ven en el siguiente esquema:

**Figura 3: Pasos para resolver un problema mediante un computador.**



Los pasos 1 y 2 del esquema anterior resultan esenciales para seguir adelante en el proceso. "Entender el problema" (es decir, lograr una "representación del problema") es básicamente saber identificar los *objetivos* (o *resultados*) a cumplir, junto a los *datos* de los cuales se parte, y ser capaz de plantear una estrategia general de resolución en la que se indiquen los *procesos básicos* a realizar para llegar a los resultados partiendo de los datos. En esta etapa de comprensión inicial, esa estrategia de planteo no requiere mayor nivel de detalle: basta con que se pueda simplemente comunicar en forma verbal una línea de acción general. Si bien puede parecer una etapa simple y obvia, los hechos muestran que en realidad se trata del punto más complicado y difuso en el proceso de resolver un problema. Si el problema no es de solución trivial, quien plantea el problema debe ser capaz de intuir una línea de acción que lo conduzca, eventualmente, a una solución correcta. Mientras más experimentada en resolución de problemas es la persona, mayor es la probabilidad que la línea de acción intuida sea correcta. Sin embargo, no hay garantías: quien intenta resolver el problema sigue rutas lógicas que su experiencia indica pueden ser correctas, pero el éxito no es seguro. A menudo los caminos elegidos llevan a puntos muertos, o terminan alejándose de la solución buscada. Pues bien: el conjunto de estrategias, procesos y caminos lógicos que una persona *intuye* que puede servir para dar con una solución a un problema no trivial (en cualquier campo o disciplina), se conoce con el nombre de *heurístico* (del griego *heuriskin*: lo que sirve para resolver), o más general, como *método heurístico*.

Es obvio que una persona con escasos o nulos conocimientos en el área del problema, difícilmente podrá encontrar una solución al mismo. Pero si los conocimientos se suponen dados, aun así los hechos muestran que encontrar una "estrategia resolvente" para un problema es una tarea exigente. Como vimos en la Figura 3, todo el trabajo que un programador debe realizar comienza con la inspiración de un heurístico: la *intuición de una estrategia resolvente*. Si en este punto falla o no logra ni siquiera comenzar, no podrá dar un solo paso más y la computadora no le servirá de mucho.

Es en el segundo paso, con el planteo del algoritmo, donde la estrategia elegida debe ser detallada con el máximo y exhaustivo rigor lógico. En esta etapa, cada paso, cada condición, cada rama lógica del proceso, debe ser expresada sin ambigüedad. Toda la secuencia de acciones a cumplir para llegar a los objetivos planteados debe quedar evidenciada en forma clara y terminante. Normalmente, se emplean en esta etapa una serie de símbolos para facilitar la tarea de mostrar el algoritmo, formando lo que se designa como un *diagrama de flujo* junto a otros diagramas específicos. Cabe aclarar sin embargo, que hasta aquí lo importante es plantear en forma rigurosa la lógica del algoritmo, y no los detalles sintácticos

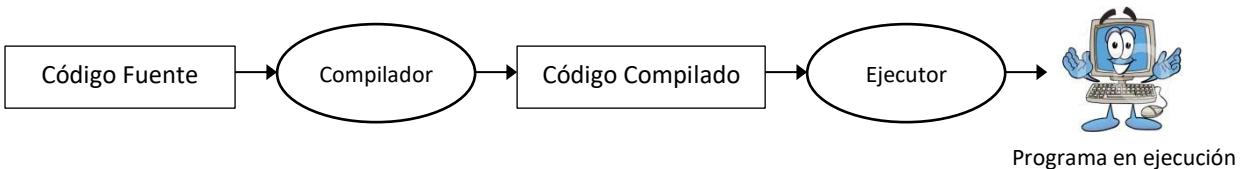
del programa que finalmente se cargará en la computadora. En el momento de plantear un algoritmo, los detalles sintácticos no son importantes (de hecho, ni siquiera es relevante cuál sea el lenguaje de programación que se usará).

En el paso 3, y sólo cuando se ha logrado el planteo lógico de la solución (usando un diagrama de flujo o alguna otra técnica), es que se pasa a la “etapa sintáctica” del proceso: el paso siguiente es desarrollar un programa mediante algún lenguaje de programación, lo cual requiere de ciertos conocimientos esenciales y básicos que se adquieren sin mayores dificultades en una o dos semanas de estudio.

El programa que escribe un programador usando un lenguaje de programación se denomina *programa fuente* o *código fuente*. Antes de poder ejecutar un programa, se debe verificar si existen errores de escritura y/o otras anomalías en el código fuente. Para ello se usan otros programas (que vienen incluidos con el kit de desarrollo o *SDK* del lenguaje usado) llamados *compiladores* o *intérpretes* según la forma de trabajo de cada uno.

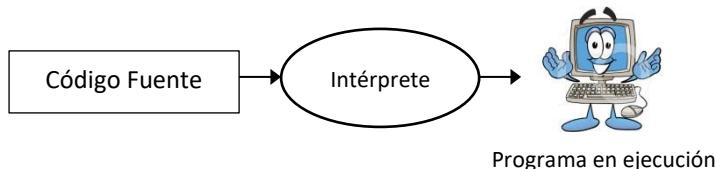
Un *compilador* es un programa que verifica el archivo de *código fuente* de otro programa detectando errores tales como palabras mal escritas, signos mal usados, etc. Como resultado, si todo anduvo bien, el *compilador* produce *otro* archivo contenido el llamado *programa compilado* o *código compilado* o también *código ejecutable*, y es este *programa compilado* el que *realmente* se ejecuta en la computadora (ver la Figura 4). Ahora sí aparecen las reglas del lenguaje y detalles tan triviales como la ausencia de una coma harán que el compilador no pueda *entender* el programa, acusando errores que impedirán que ese programa comience a ejecutarse.

**Figura 4:** Esquema general de trabajo de un *compilador*.



El esquema de trabajo de un *intérprete* es ligeramente distinto, ya que un *intérprete* **no genera otro archivo** con el código compilado, sino que ejecuta directamente, línea por línea, el *código fuente* (ver la Figura 5). Si la línea que actualmente está analizando no contiene errores, la ejecuta y pasa a la siguiente, continuando así hasta llegar al final del programa o hasta encontrar una línea que contenga un error de sintaxis (en cuyo caso, la ejecución se interrumpe mostrando un error).

**Figura 5:** Esquema general de trabajo de un *intérprete*.



Algunos lenguajes de programación son *compilados*. Otros son *interpretados*. Y algunos trabajan en forma *mixta*, compilando el código fuente para producir un archivo de *código intermedio* no ejecutable en forma directa (llamado *código de byte*) que a su vez se ejecuta por medio de un *intérprete especial* de ese lenguaje (y que suele designarse como *máquina virtual* del lenguaje). Ejemplos conocidos de *lenguajes compilados* son *C*, *C++* y *Pascal*. Como

ejemplo de *lenguaje mixto* podemos citar a *Java*. Y en nuestro caso más próximo, tenemos a *Python* que finalmente es un ejemplo de *lenguaje interpretado*.

A pesar de todo lo dicho, cabe aclarar que por razones de comodidad y de *uso y costumbre* entre los programadores, tanto los errores de sintaxis detectados e informados por un compilador como los detectados e informados por un intérprete son designados indistintamente como *errores de compilación*: es decir, errores en la *sintaxis* de un programa fuente que impiden que ese programa comience a ejecutarse y/o finalice en forma completa (si está siendo ejecutado por un *intérprete*).

Como vimos, un programa no es otra cosa que un algoritmo escrito en un lenguaje de programación. Conocer las características del lenguaje es un paso necesario, pero no suficiente, para poder escribir un programa: alguien debe plantear antes un algoritmo que resuelva el problema dado. En todo caso, la computadora (a través del compilador o el intérprete) ayuda a encontrar *errores sintácticos* (pues si un símbolo falta o está mal usado, el compilador o el intérprete avisa del error), pero no puede ayudar con los métodos heurísticos para llegar a un algoritmo.

Cuando el programa ha sido escrito y verificado en cuanto a sus errores de compilación, el mismo puede ser cargado en la computadora y finalmente en el paso 4 se procede a solicitar a la computadora que lo *ejecute* (es decir, que lleve a cabo las tareas indicadas en cada instrucción u orden del programa). Si el programa está bien escrito y su lógica subyacente es correcta, puede confiarse en que los resultados obtenidos por el ordenador serán a su vez correctos. Si el programa contiene errores de lógica, aquí debe realizarse un proceso de revisión (o verificación) y eventualmente repetir los pasos desde el 2 hasta el 4. Veremos más adelante numerosos ejemplos de aplicación de este proceso.

Para terminar esta introducción, debe notarse que el motivo básico por el cual una persona que sabe plantear un algoritmo recurre a un computador para ejecutarlo, es la gran velocidad de trabajo de la máquina. Nadie duda, por ejemplo, que un ingeniero sabe hacer cálculos de estructuras, pero con una computadora adecuadamente programada podrá hacerlo mucho más rápido.

### 3.] Fundamentos de representación de información en sistema binario.

Una computadora sólo sirve si posee un programa cargado al que pueda interpretar y ejecutar instrucción por instrucción. Además del propio programa, deben estar cargados en el interior de la computadora los datos que ese programa necesitará, y del mismo modo la computadora almacenará los resultados que vaya obteniendo.

Esto implica que una computadora de alguna forma es capaz de representar y retener información dentro de sí. En ese sentido, se llama *memoria* al dispositivo interno del computador en el cual se almacena esa información y básicamente, la memoria puede considerarse como una gran tabla compuesta por celdas individuales llamadas *bytes*, de modo que cada *byte* puede representar información usando el *sistema de numeración binario* basado en estados eléctricos.

Para entender lo anterior, digamos que esencialmente una computadora sólo puede codificar y decodificar información mediante *estados eléctricos*. Cada celda o *byte* de memoria, está compuesto a su vez por un conjunto de ocho terminadores eléctricos, que reciben cada uno el nombre de *bit* (acrónimo del inglés *binary digit* o *dígito binario*). En la

práctica, se suele reducir la idea de *bit* a la de un componente que en un momento dado puede estar *encendido* (o con *valor 1*) o bien puede estar *apagado* (o con *valor 0*) [1]. Por este motivo, decimos que una computadora sólo puede usar e interpretar el *sistema de numeración binario* (o de *base 2*), que consta únicamente de los dígitos 0 y 1.

En base a los dos únicos dígitos 0 y 1, la computadora debe codificar cualquier tipo de información que se requiera. Si se trata de representar números enteros, está claro que los dígitos 0 y 1 del sistema binario servirán sin problemas para representar el 0 y el 1 del *sistema de numeración decimal* (o de *base 10*). Por lo tanto, un único bit podría usarse para representar al 0 o al 1 en la memoria. Pero si se quiere representar al número 2 o al 3, está claro que necesitaremos *más bits* y combinar sus valores [5].

Dado que un bit permite representar dos estados posibles (esto es:  $2^1 = 2$  estados), entonces dos bits *juntos* pueden representar  $2^2 = 4$  estados. Con eso alcanzaría para representar en binario a los dígitos del 0 al 3 de la base 10:

**Figura 6: Representación en binario de los primeros cuatro números decimales.**

Dígito decimal	Representación binaria (precisión: 2 bits)
0	00
1	01
2	10
3	11

La cantidad de bits usada para representar en binario un valor cualquiera, se conoce como la *precisión* de la representación. En la tabla anterior, se representaron los cuatro primeros dígitos decimales, con precisión de dos bits.

Puede verse claramente por qué motivo en la memoria de un computador, los bits se agrupan tomados de a ocho para formar bytes: una precisión de 8 bits permitiría representar  $2^8 = 256$  números enteros diferentes en un único byte. Por lo tanto, si a modo de ejemplo asumimos que *sólo se representarán números positivos*, entonces un único byte puede usarse para representar números enteros en el intervalo [0, 255] (lo cual equivale a 256 números, incluido el 0). Si se quiere representar números enteros positivos mayores a 255, entonces la computadora *agrupa dos o más bytes*, e interpreta el contenido agrupado para obtener un único valor. Así, usando *dos bytes* agrupados, se tiene un total de 16 bits que implican  $2^{16} = 65536$  números enteros positivos diferentes, en el intervalo [0, 65535].

Y así continúa: mientras mayor sea la magnitud del número que se quiere representar, mayor será la cantidad de bits que requerirá para su representación interna en un computador, y mayor será entonces la cantidad de bytes que ocupará. Como los números enteros negativos también se representan en binario, usando una técnica designada como *complemento a dos*, entonces un único byte normalmente puede almacenar tanto negativos como positivos, en precisión de 8 bits, pero esto (y otros detalles técnicos) hace que el rango de valores ahora sea el del intervalo [-128, 127] (o sea, 256 números desde el -128 hasta el 127, incluyendo al 0). Si se usan dos bytes, entonces el rango de valores enteros es [-32768, 32767] (o sea, 65536 números, desde el -32768 hasta el 32767 incluido el 0). Usando 4 bytes, se tienen 32 bits de precisión, con  $2^{32} = 4294967296$  valores enteros diferentes (más de 4 mil millones de números) en el intervalo [-2147483648, 2147483647] (que incluye al 0) si se admiten negativos.

La representación de caracteres (letras, símbolos especiales, etc.) también se hace en binario, asignando a cada carácter una combinación predefinida de bits. Esas combinaciones constituyen estándares muy conocidos de la industria informática, y se representan en tablas en las que a cada carácter o símbolo, se le hace corresponder una combinación de bits que representa un número. Quizás el estándar más conocido sea el designado como *ASCII* (abreviatura de las iniciales de *American Standard Code for Information Interchange* o *Código Estándar Americano para el Intercambio de Información*). En el estándar *ASCII*, cada carácter se representa con un único byte (o sea, ocho bits). De aquí se deduce que un carácter en memoria ocupará un byte si se usa el estándar *ASCII*, y que una palabra (o *cadena de caracteres*) ocupará tantos bytes como caracteres incluya [1].

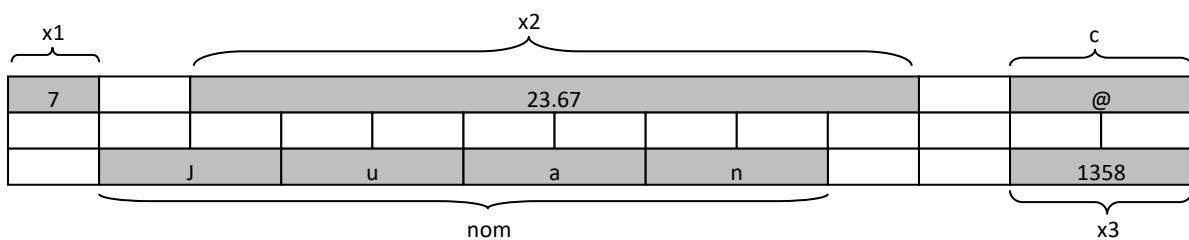
Dado que el estándar *ASCII* sólo emplea un byte por carácter, la cantidad de caracteres diferentes que pueden representarse mediante *ASCII* es de sólo 256. Eso era suficiente en las primeras épocas de la industria informática, pero a partir de los 80 en adelante comenzaron a surgir exigencias de representación de un mayor número de caracteres (como las letras de los alfabetos orientales, por ejemplo). En consecuencia, se propusieron otros estándares de mayor precisión y uno de ellos (muy difundido actualmente) es el estándar *Unicode* en el cual cada carácter puede ser representado con *uno o más bytes* (dependiendo del tipo de codificación que se use, como *UTF-8* o *UTF-16*, por ejemplo) en lugar de sólo uno.

#### 4.] Elementos básicos de programación en *Python*: tipos, variables y asignaciones.

Cuando se desea ejecutar un programa, el mismo debe cargarse en la memoria del computador. Un programa, entonces, ocupará normalmente varios cientos o miles de bytes. Sin embargo, con sólo cargar el programa no basta: deben cargarse también en la memoria los *datos* que el programa necesitará procesar. A partir de aquí, el programa puede ser ejecutado e irá generando ciertos *resultados* que de igual modo serán almacenados en la memoria. Estos valores (los datos y los resultados) ocuparán en memoria un cierto número de bytes, que como vimos depende del *tipo de valor* del que se trate (por ejemplo, en general en distintos lenguajes de programación un valor de *tipo entero* ocupa entre uno y ocho bytes en memoria, un valor de *tipo real* o de *coma flotante*, con punto y parte decimal, ocupa entre cuatro y ocho bytes, y un *caracter* ocupa uno o dos bytes dependiendo del estándar de representación empleado) [4].

En todo lenguaje de programación, el acceso a esos datos y resultados ubicados en memoria se logra a través de ciertos elementos llamados *variables*. Una *variable* es un grupo de bytes asociado a un nombre o identificador, de tal forma que a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a la variable. En el siguiente gráfico, se muestra un esquema conceptual referido a cómo podría verse la memoria de un computador, con cuatro variables (*x1*, *x2*, *x3*, *c*, *nom*) alojadas en ella, suponiendo que estamos usando el lenguaje *Python*:

**Figura 7: Memoria de un computador (esquema)**



En el gráfico (que es sólo una *representación libre* de la idea de memoria como tabla formada por casilleros), se supone que la variable *x1* contiene el valor 7, y del mismo modo el resto de las variables del cuadro contiene algún valor. El nombre de cada variable es elegido por el programador, de acuerdo a ciertas reglas que se verán más adelante.

Está claro que lo que *realmente* se almacena en cada variable es la *representación binaria* de cada valor, pero aquí por razones de simplificación mostramos los valores tal como los entiende una persona. La cantidad de bytes que se usó para graficar a cada variable está en correspondencia con el tipo de valor representado y también de acuerdo a la forma de trabajo de *Python*: en este lenguaje, una variable que almacene un número entero ocupará *automáticamente* la cantidad de bytes que requiera para poder representar ese valor. Por lo tanto, la variable *x1* sólo ocupará un byte (ya que el número 7 representado en binario puede almacenarse sin problemas en un único byte). Pero la variable *x3*, que también almacena un número entero, en este caso ocupa dos bytes ya que el valor 1358 en binario requiere al menos de dos bytes para poder ser representado.

En la misma gráfica, la variable *x2* se representó con ocho bytes. El motivo es que *x2* contiene el valor 23.67 que no es entero, sino de *coma flotante*. En Python, *todos* los números de *coma flotante* (o *reales*) se representan con ocho bytes de precisión, sin importar la magnitud del número.

Finalmente, la variable *c*, que contiene un único carácter (el signo @) se graficó con dos bytes. El motivo: Python utiliza el estándar *Unicode* para representar caracteres. Por la misma razón, la variable *nom* en la gráfica ocupa ocho bytes: dos por cada carácter de la palabra "Juan".

Para darle un valor a una variable en Python (o para cambiar el valor de la misma)<sup>6</sup> se usa la *instrucción de asignación*, que consiste en escribir el nombre de la variable, seguido del signo igual y luego el valor que se quiere asignar. El signo igual (=) se designa como *operador de asignación* [6]. Ejemplo:

```
x1 = 7
```

La instrucción anterior, internamente, convierte el valor 7 a su representación binaria, y almacena ese patrón de bits en la dirección de memoria en la que esté ubicada la variable *x1*, manejando automáticamente Python la cantidad de bytes que necesitará.

La clase de valores que una variable puede contener en un momento dado, se llama *tipo de dato*. Los lenguajes de programación proveen varios *tipos de datos* estándar para manejar variables. Así, en el lenguaje Python existen al menos tres tipos elementales que permiten manejar números enteros, números en coma flotante, y valores lógicos (*True* y *False*). Además, Python permite manejar cadenas de caracteres y muchos otros tipos de datos predefinidos. El cuadro de la Figura 8 muestra los nombres y algunas otras características de los tipos de datos que hemos citado (otros tipos serán estudiados oportunamente).

En muchos lenguajes (tales como C, C++ o Java) *no se puede* utilizar una variable en un programa si la misma no fue convenientemente *declarada* en ese programa. En esos lenguajes, la *declaración de una variable* se hace mediante la llamada *instrucción*

<sup>6</sup> La fuente de consulta general y más recomendable para el uso y aplicación del lenguaje Python, son sus propios manuales, tutoriales y documentación publicados en forma oficial por la *Python Software Foundation*. Esta documentación está designada con el número [5] en la bibliografía que aparece al final de esta Ficha.

*declarativa*, que implica indicar el nombre o identificador de la variable, e indicar a qué tipo de dato pertenece esa variable. Los lenguajes de programación que obligan a declarar una variable antes de poder usarla, comprueban que los valores asignados sean del tipo correcto y lanzan un error en caso de no serlo. En general, estos lenguajes se suelen denominar *lenguajes de tipado estático*.

Figura 8: Tabla de tipos de datos elementales en Python (sólo los más comunes).

Tipo (o Clase)	Descripción	Bytes por cada variable	Rango
<b>bool</b>	valores lógicos	1	[False, True]
<b>int</b>	números enteros	dinámico <sup>7</sup>	ilimitado <sup>8</sup>
<b>float</b>	números reales	8	hasta 15 decimales
<b>str</b>	cadenas de caracteres	2 * cantidad de caracteres	Unicode

Por su parte, *Python* y otros lenguajes (como *Perl* o *Lisp*) son *lenguajes de tipado dinámico*: esto significa que una misma variable puede recibir y almacenar valores de *tipos diferentes* durante la ejecución de un programa, y por lo tanto *las variables no deben declararse en base a un tipo específico prefijado antes de ser usadas*.

A diferencia de los lenguajes de tipado estático, una variable en Python se define (o sea, se aloja en memoria) *en el momento en que se le asigna un valor* (de cualquier tipo) por primera vez, y el tipo inicialmente asumido para esa variable es el que corresponda al valor que se le asignó. Por ejemplo, el siguiente esquema *define* cuatro variables en Python:

```
a = 14
b = 23.456
c = 'Python'
d = True
```

La variable *a* se define inicialmente como de tipo entero (*int*), y toma el valor 14. Respectivamente, las variables *b*, *c* y *d* se definen como flotante (*float*), cadena de caracteres (*str*) y lógica (*bool*). Note en los cuatro casos, el uso del *operador de asignación* (el *signo igual* =) para llevar a cabo la asignación de cada valor en la variable respectiva. Note también que en cada una de las cuatro instrucciones se usó directamente el operador de asignación y el lenguaje deduce el tipo y el tamaño en bytes de la variable de acuerdo al valor asignado. El *tipo es implícito* y en ningún caso se requiere usar el propio nombre de cada tipo (*int*, *float*, *str*, *bool* o el que fuese).

Si se trata de asignar un número entero, es suficiente con escribir ese número luego del operador de asignación, eventualmente precedido de un signo menos. Si el número es de coma flotante, recuerde usar *el punto* (y no *la coma*) como separador entre la parte entera y la parte decimal. Si quiere asignar una cadena de caracteres o un carácter único a una variable, encierre la cadena o el único carácter entre *comillas dobles* o entre *apóstrofos* (o *comillas simples*) como se hizo en el ejemplo. Como veremos, el uso de comillas dobles o simples es indistinto en *Python*, siempre que se mantenga consistente: si abrió con comillas dobles, debe cerrar con comillas dobles, pero si abrió con apóstrofos, debe cerrar con apóstrofos. Y finalmente, si desea asignar un valor lógico (de tipo *bool*) a una variable, debe

<sup>7</sup> Es decir, la cantidad de bytes que ocupa una variable de tipo *int* se calcula y asocia en el momento en que se necesita.

<sup>8</sup> En este contexto, "ilimitado" significa que no hay un *límite teórico* al rango de valores enteros que se pueden representar en Python, pero hay un *límite práctico* que es la *capacidad de memoria disponible*.

recordar que sólo existen dos posibles valores *bool*: el *True* y el *False*, sin comillas y escritos exactamente así: la *T* en mayúscula para el primero, y la *F* en mayúscula para el segundo.

Como *Python* es un lenguaje de *tipado dinámico*, entonces en Python una *variable* *puede cambiar de tipo* si se le asigna un valor de un tipo diferente al que tenía inicialmente, sin que esto provoque ninguna clase de error, y puede hacerse esto tantas veces como se quiera o necesite el programador. En la siguiente secuencia, la variable *n* se asocia inicialmente con el valor entero 20, se muestra en la consola de salida ese valor, y luego se cambia el valor y el tipo de *n* en las instrucciones sucesivas, para ilustrar la idea:

```
# asignación inicial de un valor int a la variable n...
n = 20
print('Valor de n: ', n)

# ahora el valor de n cambia a "Juan" (que es un str)...
n = 'Juan'
print('Nombre almacenado en n: ', n)

# y ahora se vuelve a cambiar el valor de n a un boolean...
n = False
print('Valor booleano almacenado en n: ', n)
```

Observe el uso del *signo numeral (#)* para introducir un *comentario de línea* que será ignorado por el intérprete Python. Lo que sea que se escriba a la derecha del signo # será tomado como un texto fuera del programa por Python, y su efecto dura hasta el final de la línea. Si se desea introducir un comentario que ocupe más de una línea, se recomienda simplemente escribir esas líneas iniciando a cada una con su correspondiente numeral (#), como en el ejemplo que sigue:

```
# 
# Un comentario de párrafo...
# Asignación inicial de un valor int
# a la variable n
#
n = 20
print('Valor de n: ', n)
```

Todo el bloque escrito en rojo en el ejemplo anterior, será ignorado por Python como si simplemente no existiese. Existe un tipo de comentario de párrafo designado como *comentario de documentación* o *cadena de documentación* (conocido como *docstring* en la terminología propia de Python) que consiste en usar el signo *triple apóstrofo* (' ''') o *triple comilla* (" """) para indicar el inicio del texto comentado, y nuevamente el *triple apóstrofo* o la *triple comilla* para cerrar su efecto. Sin embargo, los *docstrings* son tipos especiales de comentarios, que se usan con el objetivo de documentar un sistema y no sólo para intercalar texto libre en un programa. Volveremos en una ficha posterior sobre los *docstrings*.

Los dos ejemplos anteriores nos muestran además (muy básicamente) la forma de usar la función predefinida *print()* para mostrar un mensaje y/o el valor de una variable en la consola estándar de salida [6].

Note también que *Python* es *case sensitive* (o sea, es sensible a la diferencia entre mayúsculas y minúsculas) tanto para palabras e identificadores reservados, como para identificadores del programador. Así, por caso, el nombre de la función es *print* y no *Print*, y como vimos, las constantes *True* y *False* deben escribirse con la primera letra en mayúscula y el resto en minúscula pues de otro modo se producirá un error.

El tamaño en bytes de una variable es automáticamente ajustado por Python de acuerdo al valor que se asigna en la misma, sin que ese detalle deba preocupar al programador. Así, si se trabaja con números enteros y se asignan a una variable valores como 4, 2345 o 54763, Python adecuará el tamaño de la variable usada a uno, dos o cuatro bytes según sea el caso, y así en forma similar con otros valores que requieran mayor espacio o con otros tipos de datos.

Un hecho importante derivado de la forma de trabajo de la instrucción de asignación, es que cuando se asigna un valor a una variable, esta asume el nuevo valor, y pierde cualquier valor anterior que hubiera contenido. Por ejemplo, considérese el siguiente segmento de programa:

```
a = 2
a = 4
```

En este caso, se comienza asignando el valor 2 a la variable *a*, pero inmediatamente se asigna el 4 en la misma variable. El valor final de *a* luego de estas dos instrucciones, es 4, y el 2 originalmente asignado se pierde. Lo mismo ocurre con cualquier otra variable, sea cual sea el tipo de la misma.

En Python una variable puede cambiar dinámicamente de tipo, pero lo que *no* se puede hacer es intentar usar una variable sin haberle asignado previamente un valor inicial alguna vez. El siguiente ejemplo provocará un error (suponiendo que la variable *b* no haya sido usada nunca antes):

```
a = 34
print('Valor de a: ', a)

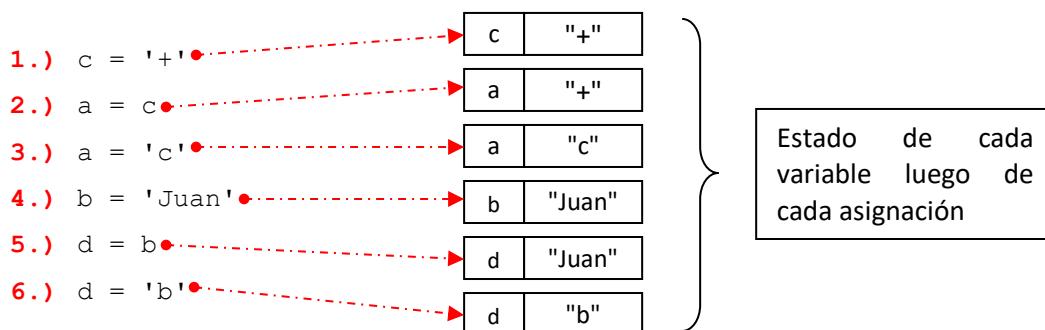
# esto provocará un error: b no está previamente asignada...
print("Valor de b: ", b)
```

En algunas situaciones, puede esperarse que una variable sea asignada con el valor *None* (que implícitamente equivale a un *False*), indicando que la variable es válida pero carece de valor en ese momento:

```
r = None
print('Valor actual de r: ', r)
```

Los siguientes ejemplos son válidos para ilustrar algunos elementos básicos en cuanto a gestión de variables y las posibilidades de la instrucción de asignación (los números a la izquierda de cada línea se usan para poder hacer referencia a cada ejemplo, y no forman parte de un programa en Python):

**Figura 9: Ejemplos de asignaciones de variables en Python.**



En la línea 1.) se asigna el carácter "+" a la variable *c*, y luego en la línea 2.) se asigna la variable *c* en la variable *a*. Esto significa que la variable *a* queda valiendo *lo mismo* que la variable *c* (es decir, ambas contienen ahora el signo "+")<sup>9</sup>. Observar la diferencia con lo hecho en la línea 3.), en donde se asigna a la variable *a* el carácter "c" (y **no** el *contenido de la variable c*). En la línea 4.) se asigna la cadena "Juan" a la variable *b*, y en la línea 5.) se asigna el contenido de la variable *b* en la variable *d* (de nuevo, ambas quedan valiendo lo mismo: la cadena "Juan"). En la línea 6.) se está asignando literalmente la cadena "b" a la variable *d*, por lo que las líneas 5.) y 6.) no son equivalentes...

Para finalizar esta sección, recordemos que el nombre o identificador de una variable (y en general, el identificador de cualquier elemento creado por el programador) es elegido por el propio programador, pero para ello deben seguirse ciertas reglas básicas, que indicamos a continuación [6]:

- El nombre o identificador de una variable en Python, sólo puede contener letras del alfabeto inglés (mayúsculas y/o minúsculas, o también dígitos (0 al 9), o también el guion bajo ( \_ ) (también llamado guion de subrayado).
- El nombre de una variable *no debe comenzar* con un dígito.
- Las palabras reservadas del lenguaje Python no pueden usarse como nombres de variables.
- El nombre de una variable puede contener cualquier cantidad de caracteres de longitud.
- Recordar que Python es *case sensitive*: Python hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales. El identificador *sueldo* no es igual al identificador *Sueldo* y Python tomará a ambos como dos variables *diferentes*.

Hacemos notar que estas reglas son las más elementales, y son las que fija la versión Python 2. Pero en la versión Python 3 se permite utilizar letras acentuadas, letras que no sean del alfabeto inglés, letras de idiomas no occidentales, etc.<sup>10</sup> En la medida de lo posible, sin embargo, trataremos de ajustarnos a las cinco reglas que dimos más arriba para evitar confusiones. Por ejemplo, los siguientes identificadores de variables son válidos en la versión Python 3:

**Figura 10: Ejemplos de identificadores de variables *válidos* en Python 3.**

Identificador	Observaciones
n1	Válido en Python 2 y Python 3
nombre_2	Válido en Python 2 y Python 3
sueldo_anterior	Válido en Python 2 y Python 3
x123	Válido en Python 2 y Python 3
año	Válido en Python 3 – No válido en Python 2 por el uso de la ñ (no inglesa)
número	Válido en Python 3 – No válido en Python 2 por el uso de la ú (no valen acentos en inglés)

<sup>9</sup> Cuando se asigna un carácter o una cadena de caracteres a una variable, las comillas (simples o dobles) usadas para delimitar al carácter o a la cadena, no quedan almacenados en la variable. En la gráfica se muestran las comillas como si estuviesen contenidas en las variables, pero sólo por razones de claridad.

<sup>10</sup> En definitiva, en Python 2 un identificador se forma usando caracteres ASCII que representen letras, guiones bajos o dígitos, pero en Python 3 se aceptan también caracteres Unicode (*no – ASCII*), para dar cabida a símbolos propios de otros idiomas diferentes del inglés.

Pero los siguientes son incorrectos, por los motivos que se indican:

**Figura 11: Ejemplos de identificadores de variables *no válidos* en Python 3.**

Identificador	Incorrecto por el siguiente motivo:
3xy	Comienza con un dígito.
dir ant	Usa un espacio en blanco como separador (caracter no válido).
nombre-2	Usa el guion alto y no el bajo como separador
else	Es palabra reservada del lenguaje.

## 5.] Visualización por consola estándar y carga por teclado.

Al ejecutar un programa, lo normal es que antes de finalizar el mismo muestre por pantalla los resultados obtenidos. Como ya adelantáramos, el lenguaje Python (versión Python 3) provee la función predefinida *print()* para hacerlo<sup>11</sup>. Esta función permite mostrar en pantalla tanto el contenido de una variable como también mensajes formados por cadenas de caracteres, lo cual es útil para lograr salidas de pantalla amigables para quien use nuestros programas [6]. La forma de usar la función se muestra en los siguientes ejemplos:

```
a = 3
b = a + 1
print(b)
```

Aquí, la instrucción de la última línea muestra en pantalla el valor contenido en la variable *b*, o sea, el número 4. Sin embargo, una salida más elegante sería acompañar al valor en pantalla con un mensaje aclaratorio, lo cual puede hacerse en forma similar a lo que sigue:

```
print('El resultado es: ', b)
```

Observar que ahora no sólo aparecerá el valor contenido en *b*, sino también la cadena "*El resultado es:*" precediendo al valor. En general, si lo que se desea mostrar es una combinación entre cadenas de caracteres y valores de variables, entonces cada elemento se escribe entre los paréntesis, usando una coma a modo de separador, en el orden en que se espera que aparezcan visualizados,

Notar también que para mostrar el valor de una variable, el nombre de la misma *no lleva comillas ni apóstrofos*, pues en ese caso se tomaría al nombre en forma literal. La instrucción que sigue, muestra literalmente la letra "b" en pantalla:

```
print('b')
```

Así como Python 3 provee la función *print()* para la visualización de mensajes y/o valores en consola estándar, también provee la función *input()* que permite realizar con comodidad operaciones de carga de datos desde el teclado mientras el programa se está ejecutando.

La función *input()* permite tomar como parámetro una cadena de caracteres (es decir, escribir esa cadena entre los paréntesis de la función), que acompañará en la consola al *prompt* o *cursor* que indique que se está esperando la carga de datos. Lo que sea que el

<sup>11</sup> Aquí corresponde una aclaración: en la versión Python 2, *print* no es una función sino una *instrucción* del núcleo del lenguaje, y por lo tanto se usa sin los paréntesis: en Python 2, la forma de mostrar el valor de la variable *b* sería *print b*, mientras que en Python 3 (donde *print* es una función) se usan los paréntesis: *print(b)*.

usuario cargue, será retornado por la función `input()` en forma de cadenas de caracteres (con el salto de línea final removido) [6]. Así, una instrucción de la forma:

```
nom = input('Ingrese su nombre: ')
```

provocará que se muestre en consola el mensaje "*Ingrese su nombre:*", con el cursor del *prompt* parpadeando a continuación y dejando el programa en estado de espera hasta que se el usuario ingrese datos y presione `<Enter>`. Si al ejecutar esta instrucción, por ejemplo, el usuario escribe la palabra "Ana" (sin las comillas) y presiona `<Enter>`, la variable `nom` quedará valiendo la cadena "Ana" (de nuevo, sin las comillas).

Si se quiere cargar por teclado un número entero o un número en coma flotante, deben usarse funciones predefinidas del lenguaje que hagan la conversión de cadena a número. Para eso, Python provee al menos dos de ellas:

- `int(cadena)`: retorna el número entero representado por la cadena tomada como parámetro (escrita entre los paréntesis).
- `float(cadena)`: retorna el número en coma flotante representado por la cadena tomada como parámetro.

En ambos casos, si la cadena tomada como parámetro no representa un número válido que pueda convertirse, se producirá un error en tiempo de ejecución y el programa se interrumpirá.

Por lo tanto, las siguientes expresiones permiten en Python cargar por teclado un número entero en la variable `n` (la primera instrucción) y un número en coma flotante en la variable `x` (la segunda):

```
n = int(input('Ingrese un valor entero: '))
x = float(input('Ingrese un valor en coma flotante: '))
```

Note que en ambas líneas de este ejemplo, se han usado las comillas simples o apóstrofos para encerrar las cadenas de caracteres. Recuerde que en Python es indistinto el uso de las comillas dobles o las simples para hacer esto, siempre que se mantenga consistente (abrir y cerrar con el mismo tipo de comilla). Las mismas dos instrucciones podrían haber sido escritas así:

```
n = int(input("Ingrese un valor entero: "))
x = float(input("Ingrese un valor en coma flotante: "))
```

O incluso combinando así:

```
n = int(input("Ingrese un valor entero: "))
x = float(input('Ingrese un valor en coma flotante: '))
```

Tenga en cuenta que en todo lenguaje existen convenciones de trabajo y consejos de buenas prácticas, y Python no es la excepción. En ese sentido, indicamos que lo más común es que se usen *comillas simples* para delimitar caracteres o cadenas de caracteres, y esa es la convención que hemos tratado de mantener a lo largo de esta ficha de estudio.

Puede verse que la posibilidad de hacer carga por teclado mientras el programa se ejecuta, permite mayor generalidad en el planteo de un programa [4]. A modo de ejemplo, suponga que se quiere desarrollar un pequeño programa en Python para sumar los números contenidos en dos variables. Un primer intento podría ser el siguiente:

```
a = 5
```

```
b = 3
c = a + b
print('La suma es: ', c)
```

El pequeño programa anterior funciona... pero podemos darnos cuenta rápidamente que así planteado es muy poco útil, pues indefectiblemente el resultado mostrado en pantalla será 8... El programa tiene muy poca *flexibilidad* debido a que el valor inicial de la variable *a* es siempre 5 y el de *b* es siempre 3. Lo ideal sería que *mientras el programa se ejecuta*, pueda *pedir* que el usuario ingrese por teclado un valor para la variable *a*, luego otro para *b*, y que luego se haga la suma (en forma similar a como permite hacerlo una calculadora...) Y eso es justamente lo que permite hacer la función *input()*. El siguiente esquema muestra la forma correcta de hacerlo:

```
a = int(input('Primer valor: '))
b = int(input('Segundo valor: '))
c = a + b
print('La suma es: ', c)
```

Observar que de esta forma, cada vez que se ejecuta el programa se puede cargar un valor distinto en cada variable, y obtener diferentes resultados sin tener que modificar y volver a ejecutar el programa.

## 6.] Operadores aritméticos en Python.

Hemos visto que siempre se puede asignar en una variable el *resultado de una expresión*. Una *expresión* es una fórmula en la cual se usan *operadores* (como suma o resta) sobre diversas variables y constantes (que reciben el nombre de *operandos* de la expresión). Si el resultado de la expresión es un número, entonces la expresión se dice *expresión aritmética*. El siguiente es un ejemplo de una *expresión aritmética* en Python:

```
num1 = 10
num2 = 7
suma = num1 + num2
```

En la última línea se está asignando en la variable *suma* el resultado de la expresión *num1 + num2* y obviamente la variable *suma* quedará valiendo 17. Note que en una asignación *primero* se evalúa cualquier expresión que se encuentre a la derecha del signo *=*, y *luego* se asigna el resultado obtenido en la variable que esté a la izquierda del signo *=*. La siguiente tabla muestra los *principales operadores aritméticos* del lenguaje Python (volveremos más adelante con un estudio más detallado sobre la aplicación de estos operadores) [6]:

**Figura 12: Tabla de operadores aritméticos básicos en Python.**

Operador	Significado	Ejemplo de uso
+	suma	a = b + c
-	resta	a = b - c
*	producto	a = b * c
/	división de coma flotante	a = b / c
//	división entera	a = b // c
%	resto de una división	a = b % c
**	potencia	a = b ** c

Los operadores de *suma*, *resta* y *producto* funcionan tal como se esperaría, sin consideraciones especiales. A modo de ejemplo, la siguiente secuencia de instrucciones calcula y muestra la suma, la resta y el producto entre los valores de dos variables *a* y *b*:

```
a = 5
b = 2

suma = a + b
resta = a - b
producto = a * b

print('Suma: ', suma)
print('Resta: ', resta)
print('Producto: ', producto)
```

Por otra parte, Python provee dos operadores diferentes para el cálculo de la *división*: el primero (`/`) calcula el cociente entre dos números, obteniendo siempre un resultado de coma flotante, sin truncamiento de decimales (lo que se conoce como *división real*). El segundo (`//`) calcula el *cociente entero* entre dos números, lo que significa que los decimales se truncan al hacer el cálculo, y siempre se obtiene sólo la parte entera de la división. El siguiente modelo aplica ambos operadores:

```
a = 7
b = 3

cr = a / b
ce = a // b

# la siguiente muestra el valor 2.3333333333333335
print('Division real: ', cr)

# la siguiente muestra el valor 2
print('Division entera: ', ce)
```

Un operador muy útil es el que permite calcular el *resto o módulo* de una división (`%`). En general, aplica sobre operandos de tipo entero (aunque puede aplicarse también sobre valores *float*). El uso de operador resto permite valerse sin problemas de diversas características de la llamada *aritmética modular* (que es justamente la parte de la aritmética que estudia las propiedades de los conjuntos de números que tienen el mismo resto al dividir por el mismo número). Por ahora, sólo nos interesa mostrar algunos ejemplos de aplicación:

```
a = 7
b = 3

r = a % b

# la siguiente muestra el valor 1
print('Resto: ', r)
```

En el esquema anterior, la variable *r* queda valiendo 1, ya que 1 es el resto de dividir 7 por 3 (el cociente entero es 2, quedando un resto de 1). En general, el operador *resto* es muy valioso en casos en que se quiere aplicar conceptos de *divisibilidad*: como la expresión  $r = x \% y$  calcula el resto de dividir en forma entera a *x* por *y*, y asigna ese resto en la variable *r*, entonces si ese resto *r* fuese 0, implicaría que *x* es divisible por *y*, o lo que es lo mismo, que *x* es múltiplo de *y*.

Note que para usar y aplicar el operador *resto* *no es necesario* usar antes el operador *división*. Ambos operadores se pueden aplicar sin tener que usar el otro. Si usted sólo desea calcular el resto de una división, simplemente use el operador % para obtenerlo y no use el operador división.

El último de los operadores aritméticos básicos de Python es el operador potencia ( \*\* ) que permite en forma muy simple calcular el valor de  $x^y$ , para cualquier valor (entero o flotante) de  $x$  e  $y$ . El siguiente esquema muestra la forma básica de usarlo:

```
a = 2
b = 3

p = a ** b

# la siguiente muestra el valor 8 (o sea, 2 al cubo)
print('Potencia: ', p)
```

Veremos más adelante, a lo largo del desarrollo del curso, aplicaciones más relevantes de todos los temas presentados en esta ficha de introducción.

### 7.] Uso del *shell* de Python.

Si se ha descargado e instalado el *SDK* de *Python*, entonces (y sólo con eso) se cuenta ya con la posibilidad de escribir, testear y ejecutar en forma básica sus primeros programas en Python a través del *editor del shell de comandos* de Python. Saber cómo hacer esto será efectivamente muy útil para comprender mejor las secciones que siguen, por lo cual se hace aquí una pequeña introducción (aunque más pronto que tarde pasaremos a usar un *IDE* (*Integrated Development Environment* o *Entorno Integrado de Desarrollo*: un programa que permite editar, depurar, ejecutar y testear programas, mucho más sofisticado que el simple *editor del shell* que viene con el *SDK*).

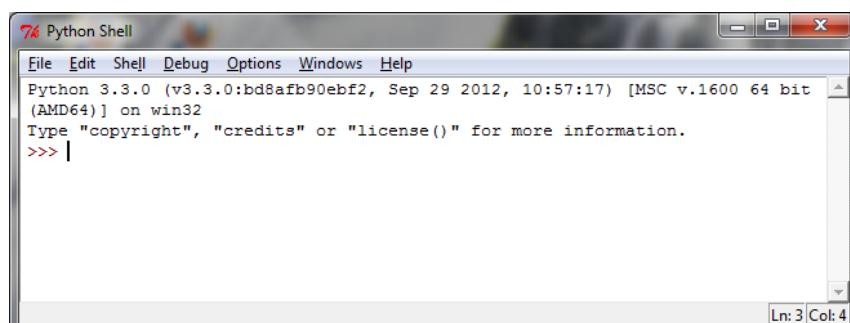
En general, este *editor del shell* forma parte de una sencilla aplicación designada como el *Python IDLE GUI* (*Python Integrated Development Environment – Graphic User Interface*) y se accede desde el grupo de programas Python haciendo click sobre el ícono correspondiente (y es útil también tener esta aplicación a mano a través de un acceso directo en su escritorio):

Figura 13: Ícono de acceso al Python IDLE GUI (o Shell de Python).



Al abrir el *IDLE GUI*, se muestra una simple ventana de edición de textos, pero integrada con el intérprete de Python, como la que se ve a continuación:

Figura 14: El shell de Python.



Esa ventana muestra el símbolo ">>>" que es el *prompt* (o *símbolo del sistema*) del shell de Python, indicando que el citado shell está a la espera de recibir instrucciones. Como Python es *interpretado*, en este momento se puede escribir cualquier instrucción de Python y presionar <Enter> para ejecutarla de inmediato (si está correctamente escrita) [7].

Como veremos oportunamente, en Python no es necesario escribir un programa sujeto a una estructura declarativa rígida: en principio, y sobre todo si se está trabajando directamente con el editor del shell, es suficiente con escribir una instrucción debajo de la otra y pedir la ejecución de cada una. En ese sentido, un lote de instrucciones Python correctamente escritas y listas para ser ejecutadas se conoce como un *script Python* (dejando la palabra "programa" para describir a una secuencia de instrucciones que responde a una estructura más elaborada... cosa que veremos).

Y bien: con el shell esperando instrucciones, no tenemos más que escribirlas y ejecutarlas. Puede comenzar, a modo de práctica básica y para asegurarse que todo está en orden, introduciendo y ejecutando (una a una) las instrucciones de la siguiente secuencia, que explicaremos más adelante. Solo asegúrese de no copiar y pegar el bloque completo en el editor del shell, ya que el intérprete espera ejecutar una por una las instrucciones (de lo contrario, obtendrá un mensaje de error):

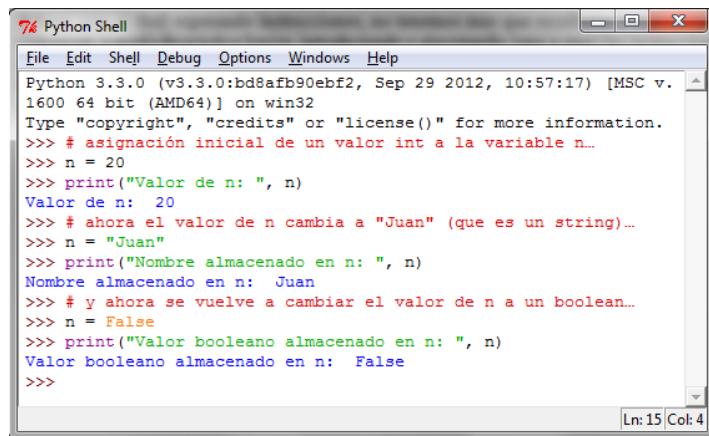
```
n = 20
print('Valor de n: ', n)

n = 'Juan'
print('Nombre almacenado en n: ', n)

n = False
print('Valor booleano almacenado en n: ', n)
```

Si todo anduvo bien, al terminar de editar y ejecutar estas instrucciones la ventana del shell debería mostrar algo como lo se muestra en la *Figura 15* (no verá las líneas escritas en rojo).

**Figura 15:** Ejemplo de uso del shell de Python.



The screenshot shows a Windows-style window titled "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main area displays Python code and its execution results. The code is as follows:

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.
1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> # asignación inicial de un valor int a la variable n...
>>> n = 20
>>> print("Valor de n: ", n)
Valor de n: 20
>>> # ahora el valor de n cambia a "Juan" (que es un string)...
>>> n = "Juan"
>>> print("Nombre almacenado en n: ", n)
Nombre almacenado en n: Juan
>>> # y ahora se vuelve a cambiar el valor de n a un boolean...
>>> n = False
>>> print("Valor booleano almacenado en n: ", n)
Valor booleano almacenado en n: False
>>>
```

The status bar at the bottom right indicates "Ln: 15 Col: 4".

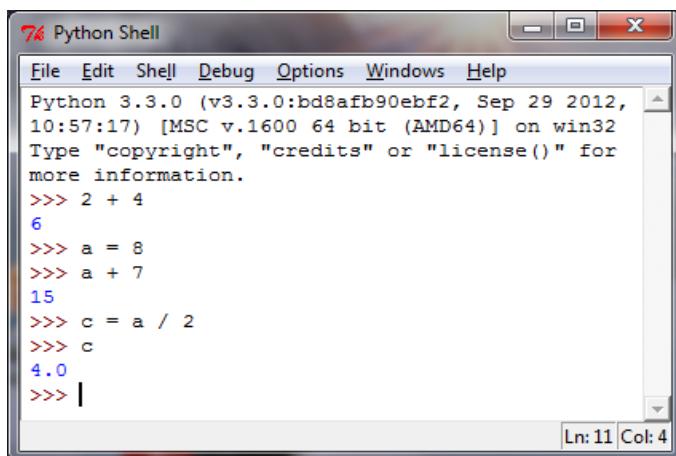
Es interesante notar que el intérprete del shell ejecuta cualquier expresión Python válida, por lo cual se puede usar a modo de "calculadora" introduciendo expresiones (aritméticas, lógicas, etc.) directas, como se muestra en la *Figura 16* de la página 24.

Observe además que si está trabajando con el editor del shell, no necesita usar la función *print()* para mostrar el valor de una variable o de una expresión: basta con escribir esa variable o expresión en el prompt, y simplemente el intérprete la reemplazará por su valor.

Obviamente, un script o cualquier conjunto de código fuente en Python, se puede almacenar en un archivo externo para poder recuperarlo, re-editarlo y volver a ejecutarlo cuando sea

necesario. En este sentido, la convención del lenguaje es que un archivo que contenga código fuente Python puede tener *cualquier nombre* que el programador desee, seguido de la extensión .py. La opción *File* del menú superior del shell de Python contiene a su vez las opciones que permiten grabar o recuperar un script. Dejamos para el estudiante la tarea de explorar el resto las opciones de la barra de menú del IDLE, y las aplicaciones que pudieran tener.

**Figura 16:** Uso del shell de Python a modo de calculadora.



```
74 Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012,
10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for
more information.

>>> 2 + 4
6
>>> a = 8
>>> a + 7
15
>>> c = a / 2
>>> c
4.0
>>> |
```

Ln: 11 Col: 4

Para cerrar esta sección, digamos que si un programa o script Python está ya desarrollado y almacenado en un archivo externo, entonces se puede ejecutar ese programa *directamente desde la línea de órdenes del sistema operativo huésped*. Los detalles de cómo hacer esto están resumidos en sección siguiente. Dejamos para el alumno el estudio y aplicación de estos temas.

## 8.] Ejecución por línea de órdenes en Python.

Cuando se escribe un programa o script Python, se grabará en un archivo que puede tener el nombre que prefiera el programador, pero con extensión .py (que como ya indicamos en la Ficha, es la extensión típica de un archivo que contiene código fuente en Python). Un archivo fuente normalmente se crea a través del editor de textos del *IDE* que esté usando el programador (que en nuestro caso será *PyCharm*) y el mismo *IDE* se encargará de la extensión del archivo. Pero si no dispone de ningún *IDE*, o está practicando la forma de trabajo directa con el *SDK* (como ahora...) obviamente puede crear el fuente *con cualquier editor de textos* (por ejemplo, con el *Notepad*, con *Wordpad*, o con el que prefiera) siempre y cuando el programador recuerde colocar la extensión .py al archivo que cree.

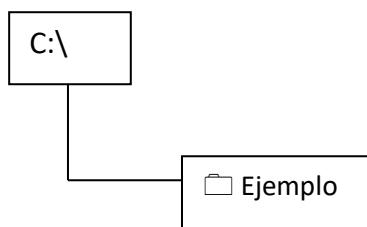
Una vez creado el fuente, el programador querrá verificar si tiene errores de sintaxis, depurar esos errores y finalmente ejecutarlo. Ese proceso en Python se lleva a cabo mediante el programa que en la plataforma Windows se llama *python.exe* del *SDK* de Python. Este programa es el *intérprete de Python*: toma un código fuente e intenta ejecutar una a una las instrucciones del mismo, a menos que alguna contenga un error de sintaxis (en cuyo caso, interrumpe la ejecución y avisa del error con un mensaje en la consola de salida). En caso de lanzarse un error de compilación, el programador debe volver al código fuente con el editor de textos, corregir el error, volver a grabar el fuente y finalmente intentar ejecutar otra vez el programa.

Aun cuando en la práctica un programador usará un *IDE* sofisticado para facilitar su trabajo, es importante que ese programador entienda a la perfección el proceso de interpretación y ejecución, ya sea para dominar todos los aspectos técnicos y poder comprender los problemas que podrían

surgir, como para poder salir delante de todos modos si le toca trabajar en un contexto donde no tenga acceso a un *IDE*. En este último caso, deberá poder hacer todo el proceso usando *directamente* los programas del SDK.

Mostraremos entonces aquí el proceso completo que debe llevarse a cabo para ejecutar un programa o script Python desde la *línea de órdenes*. Los pasos generales son los siguientes:

- i. Es conveniente, a los efectos de simplificar el proceso, que se cree primero una carpeta o directorio para contener todos los archivos asociados al programa que se está desarrollando. Esa carpeta en general se conoce como *carpeta del proyecto* y un *IDE* la crea en forma automática. En este caso, para mayor simplicidad, se puede pensar en crear esa carpeta a nivel del *directorío raíz* del disco local principal. En este ejemplo, crearemos una carpeta con el nombre *Ejemplo* en el directorio raíz del disco C, quedando así:



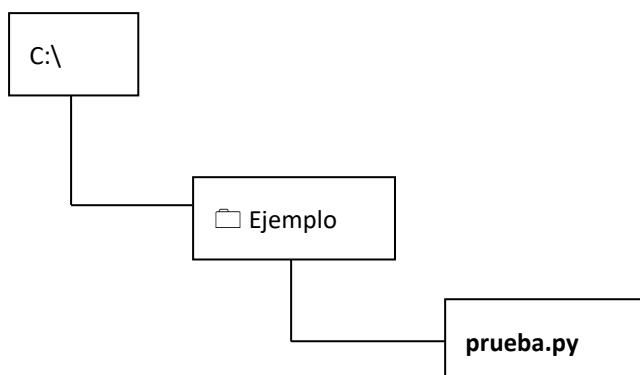
- ii. Ahora debe crear y editar un archivo fuente Python sencillo, usando cualquier editor de textos (el Block de Notas, el Wordpad, o cualquiera que tenga a mano). En este caso, plantearemos el típico ejemplo de un script sencillísimo que sólo muestra en pantalla el mensaje "Hola Mundo". El código fuente podría verse así (asumiendo que el archivo se llamará *prueba.py*)

```

# Ejemplo de script muy simple en Python
print('Hola Mundo!!!!...')

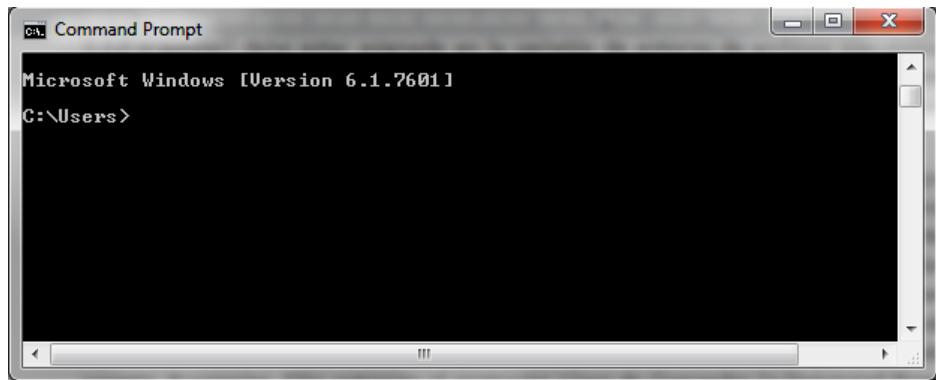
  
```

- iii. Cuando grabe el fuente que acaba de editar, recuerde hacerlo dentro de la carpeta *C:\Ejemplo* que ha creado antes (en el punto i.), y también recuerde grabar el fuente con el nombre *prueba.py*. Si está usando el Block de Notas, recuerde poner el nombre del archivo entre comillas cuando grabe, para evitar que se agregue la extensión ".txt" a su archivo. En nuestro caso, así debería verse todo en el explorador de Windows al terminar este paso:



- iv. Salga ahora a la ventana de la línea de órdenes del sistema operativo. Si está trabajando con Windows 10, una forma de hacerlo es mediante el botón *Inicio* del escritorio de Windows. Busque allí el ítem *Windows System* y elija entonces el programa *Línea de Comandos* (o *Command Prompt*). Con eso se abrirá una pequeña ventana de fondo negro: esa es la

ventana de *línea de órdenes* de Windows, y su aspecto es similar al de la siguiente captura de pantalla:



- v. Es posible que los nombres de los directorios o carpetas mostrados no coincidan necesariamente con los que verá usted. En el caso del ejemplo, la última línea que se muestra en la ventana (que se designa con el nombre general de *prompt*) tiene este aspecto:

C:\Users>\_

- vi. El prompt le está indicando en qué directorio o carpeta está ubicado el sistema de línea de órdenes de Windows en ese momento. En el estado indicado antes, se podrá ejecutar cualquier archivo con extensión .exe que se encuentre dentro de la carpeta *Users* del disco *C*, pero no se podrán ejecutar archivos .exe de otras carpetas, a menos que se ajuste el valor de la variable de entorno llamada *PATH*, de forma que contenga los nombres de los directorios que contienen a los .exe que se desea ejecutar. Como nos interesa ejecutar el programa *python.exe* (o sea, el intérprete del SDK), debemos probar si ese programa está *visible* para la línea de órdenes. Simplemente pruebe a escribir lo siguiente en la línea del prompt (sólo la palabra *python* que se remarcó en azul en el ejemplo: el resto es el contenido del prompt que ya está visible):

C:\Users>*python*

- vii. Si al presionar <Enter> aparece un mensaje de error similar a este:

"python" is not recognized as an internal or external command

entonces el intérprete no es *visible* para la línea de órdenes (recuerde que el programa *python.exe* se encuentra en el directorio del SDK, posiblemente en la carpeta llamada *C:\Program Files\Python 3.8* o similar. Verifique el nombre del directorio del SDK en su equipo). Debemos ajustar el valor de la variable *PATH* del sistema operativo, lo cual puede hacerse así (recuerde: los nombres de los directorios pueden ser diferentes... usted debe usar la ruta de la carpeta que contenga al SDK de Python EN **SU** COMPUTADORA PERSONAL):

C:\Users>set PATH=C:\Program Files\Python 3.8

- viii. Lo anterior debe dejar visible el intérprete para la línea de órdenes. Si ahora vuelve a probar con:

C:\Users>*python*

debería ver algunas líneas de ayuda para usar el intérprete, y no un mensaje de error. El siguiente paso es cambiar el directorio activo para entrar al *directorio del proyecto donde*

está el archivo *prueba.py* que se quiere interpretar y ejecutar. En nuestro caso puede hacerlo con el comando *cd* (por "change directory") en forma similar a esta secuencia:

```
C:\Users>cd \[al presionar <Enter> cambia al directorio raíz]
C:\>cd Ejemplo \[al presionar <Enter> cambia al directorio Ejemplo]
C:\Ejemplo>_
```

- ix. Con el *prompt* apuntando al directorio *C:\Ejemplo*, puede ya probar a ejecutar con el siguiente comando (sólo escriba lo que figura **en azul**... el resto es el *prompt*):

```
C:\Ejemplo>python prueba.py
```

- x. El proceso puede demorar un par de segundos. Si aparecen mensajes de error, entonces deberá volver al editor de textos, abrir nuevamente el archivo fuente, corregir el error que pudiera tener, grabar los cambios, y luego volver a ejecutar hasta que logre que el programa se ejecute sin errores. Si todo anduvo bien y sin errores, el programa será ejecutado y en este caso debería aparecer a renglón seguido el mensaje *Hola Mundo!!!...* como se ve a continuación (el *prompt* volverá a aparecer inmediatamente debajo del mensaje):

```
C:\Users>python prueba.py
Hola Mundo!!!...
C:\Users>_
```

Con esto el proceso se completa. Sugerimos encarecidamente al estudiante que intente aplicar estos pasos hasta lograr hacerlo en forma correcta, y asegurarse de poder entenderlos. Intente cometer algún error a propósito en el script contenido en el archivo *prueba.py* para comenzar a familiarizarse con el proceso de corrección de errores y el uso del intérprete. Y no se desespere: tiene que saber manejarse con el SDK en forma directa, pero cuando haya aprendido a hacerlo pasará a usar un IDE profesional (como *PyCharm*) y muchos de estos detalles quedarán automatizados.

## Bibliografía

- [1] G. Beekman, Introducción a la Informática, Madrid: Pearson Education, 2006.
- [2] C. Stephenson and J. N. Petterson Hume, Introduction to Programming in Java, Toronto: Holt Software Associates Inc., 2000.
- [3] R. Kurzweil, La Era de las Máquinas Espirituales, Buenos Aires: Editorial Planeta Argentina, 2000.
- [4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [5] S. Lipschutz, Matemáticas para Computación, México: McGraw-Hill / Interamericana, 1993.
- [6] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [7] M. Pilgrim, Dive Into Python - Python from novice to pro, Nueva York: Apress, 2004.

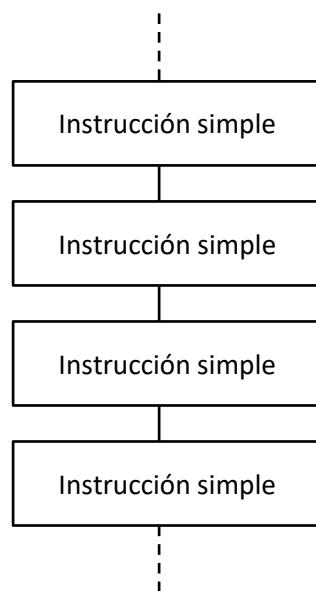
# Ficha 2

## Estructuras Secuenciales

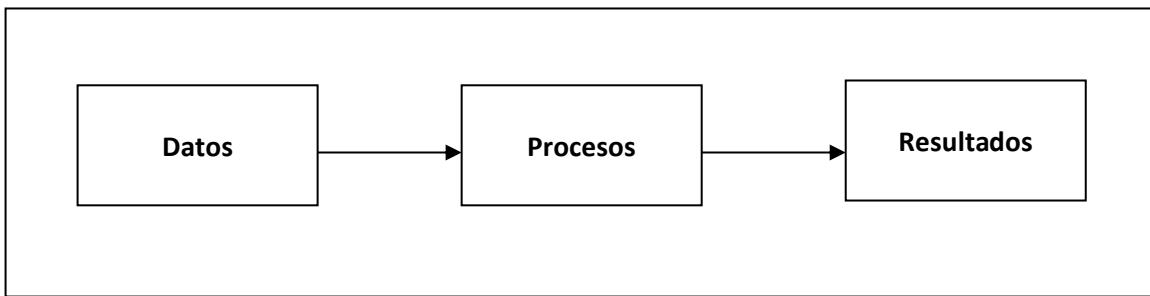
### 1.] Resolución de problemas simples. Estructuras secuenciales.

Con lo visto hasta ahora estamos en condiciones de analizar *problemas simples* de forma de poder desarrollar programas que los resuelvan. Como veremos en una ficha posterior, un problema se dice *simple* si no admite (o no justifica) ser dividido en problemas menores o *subproblemas* [1]. Por ahora, nos concentraremos en problemas simples de *lógica lineal* que pueden ser resueltos mediante la aplicación de *secuencias de instrucciones simples* (recuerde que básicamente, una *instrucción simple* es una asignación, o una instrucción de visualización o una instrucción de carga por teclado) una debajo de la otra, de tal manera que cada instrucción se ejecuta una después de la otra. Un bloque de instrucciones lineal de esta forma, se conoce en programación como un *bloque secuencial de instrucciones* o también como una *estructura secuencial de instrucciones*. La Figura 1 muestra un esquema general de una estructura secuencial:

Figura 1: Esquema general de una *Estructura Secuencial* de instrucciones.



Como vimos en la Ficha 01, antes de escribir un programa debemos ser capaces de plantear un *algoritmo* que muestre la forma de resolver el problema que se analiza. Recordemos que un *algoritmo* es la secuencia de pasos que permite resolver un problema, y por lo tanto su planteo es el paso previo a un *programa*. En ese sentido, *resolver algorítmicamente un problema* significa plantear el mismo de forma tal que queden indicados los pasos necesarios para obtener los resultados pedidos a partir de los datos conocidos. Y como también vimos en la Ficha 01, esto implica que en un algoritmo básicamente intervienen tres elementos: *datos*, *procesos* y *resultados* (ver Figura 2):

**Figura 2: Estructura de un Algoritmo**

Dado un problema, entonces, para plantear un algoritmo que permita resolverlo es siempre conveniente *entender correctamente el enunciado* y tratar de deducir y caracterizar a partir del mismo los elementos ya indicados (datos, procesos y resultados). Lo normal es:

1. Comenzar identificando los **resultados** pedidos, porque así quedan claros los objetivos a cumplir.
2. Luego individualizar los **datos** con que se cuenta y determinar si con estos es suficiente para llegar a los resultados pedidos.
3. Finalmente si los datos son completos y los objetivos claros, se intentan plantear los **procesos** necesarios para convertir esos datos en los resultados esperados.

Cabe aclarar que cuando se pide identificar resultados y datos, no se trata simplemente de enumerarlos sino, además, de *caracterizarlos*: esto es, poder indicar de qué *tipo* de datos y resultados se está hablando (números enteros o flotantes, cadenas, valores lógicos, etc.), que rango de valores son aceptables (¿negativos? ¿sólo positivos? ¿vale el cero? ¿pueden venir en una cadena caracteres que representen números mezclados con otros que representen letras? etc.) así como de asociar a cada uno un identificador genérico que luego se convierta en el nombre de una variable. En cuanto a los procesos, en esta fase de comprensión del problema se espera que el programador sepa al menos identificar el contexto del problema (¿incluye alguna noción matemática esencial? ¿el volumen de los datos a procesar requerirá de algoritmos sofisticados? ¿es necesario algún tipo de desarrollo gráfico?, ¿es similar a otros problemas ya analizados?, etc.) para que pueda comenzar a explorar posibles soluciones y/o interiorizarse sobre técnicas especiales que podría requerir.

Es completamente cierto que un programador experimentado lleva a cabo este proceso de comprensión del problema en forma mental, sin tanto formalismo, pero en el caso de un curso introductorio conviene, al menos en los primeros ejercicios, plantearlo en forma rigurosa para fijar las ideas y facilitar la discusión.

Una vez cumplido el paso anterior, se procede al planteo del algoritmo (o al menos, a intentarlo). En este punto, el programador buscará determinar el conjunto de instrucciones o pasos que permitan pasar del conjunto de datos al conjunto de resultados. En ese sentido, el algoritmo actúa como un proceso que *convierte* esos datos en los resultados pedidos, como una función que opera sobre las variables de entrada. El programador combina los diferentes tipos de instrucciones que sabe que son soportadas por los lenguajes de programación (es decir, el conjunto de *operaciones primitivas* válidas) y plantea el esquema de pasos a aplicar. Cuando el problema es complejo y el algoritmo es extenso y difícil de plantear, se suelen usar técnicas auxiliares que ayudan al programador a poner sus ideas en claro. Entre esas técnicas se cuentan los *diagramas de flujo* y el *pseudocódigo*, que analizaremos más adelante.

Los tres ejemplos de problemas que siguen serán resueltos aplicando *dos fases*, con el objetivo de poner en práctica los conceptos vistos hasta aquí: primero, se identificarán y caracterizarán *resultados, datos y procesos*; y luego se concretará un algoritmo para resolver el problema. Como en los tres casos el problema es de naturaleza *muy* sencilla, no se requerirá ninguna técnica auxiliar para plantear el algoritmo y por lo tanto el mismo será plasmado directamente como un *programa en Python* (esto no es incorrecto ni inconsistente, ya que como sabemos, un *programa es un algoritmo*).

Consideremos el enunciado del siguiente problema básico, como primer ejemplo:

**Problema 1.)** *Dado el valor de los tres lados de un triángulo, calcular el perímetro del triángulo.*

**a.) Identificación de componentes:** Para cada componente identificado, se asocia un nombre, que luego será usado como variable en el programa correspondiente.

- **Resultados:** El perímetro de un triángulo. (*p*: coma flotante)
- **Datos:** Los tres lados del triángulo. (*lad1, lad2, lad3*: coma flotante)
- **Procesos:** Problema de *geometría elemental*. El perímetro *p* de un triángulo es igual a la suma de los valores de sus tres lados *lad1, lad2* y *lad3*, por lo que la fórmula o expresión a aplicar es:  $p = \text{lad1} + \text{lad2} + \text{lad3}$ . En este caso, podemos suponer que los valores de los lados vendrán dados de forma que efectivamente puedan conformar un triángulo, pero en una situación más realista el programador debería *validar* (es decir, controlar y eventualmente volver a cargar) los valores ingresados en *lad1, lad2* y *lad3* de forma que al menos se cumplan las relaciones o propiedades elementales de los lados de un triángulo: *la longitud de cualquiera de los lados de un triángulo es siempre menor que la suma de los valores de los otros dos, y mayor que su diferencia*.

**b.) Planteo del algoritmo (como script o programa en Python):** Más adelante en esta misma Ficha (Sección 6, página 46) se muestra la forma de usar el entorno de programación *PyCharm* para Python. Puede leer esa sección antes de continuar, si lo cree necesario. Asegúrese de poder crear un proyecto nuevo en *PyCharm*, abrir el editor de textos y escribir el siguiente programa para poder probarlo y modificarlo si lo cree conveniente:

```
__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 1 - Perímetro de un triángulo')
lad1 = float(input('Longitud del primer lado: '))
lad2 = float(input('Longitud del segundo lado: '))
lad3 = float(input('Longitud del tercer lado: '))

# procesos...
p = lad1 + lad2 + lad3

# visualización de resultados...
print('El perímetro es:', p)
```

En cuanto a la *lógica general*, el programa anterior es el reflejo directo del análisis de procesos que se hizo en el paso anterior. Se cargan por teclado los valores de los tres lados (sin lo cual no se puede hacer la suma). Luego se aplica la fórmula del perímetro que se puso en claro en el mismo análisis del proceso y el resultado se *asigna* en la variable *p*. Y finalmente se muestra el resultado final en la consola de salida. Note que las variables

usadas se designaron con los mismos nombres que los que se usaron al identificar resultados y datos, y que en el planteo del script se respetaron los *tipos de datos* generales que fueron caracterizados en esa misma fase. Como dijimos, en una situación realista el programador debería incluir algún tipo de control para validar que los lados que se cargan cumplan con las propiedades que se indicaron en el análisis de procesos, pero en este ejemplo introductorio podemos obviar ese detalle y asumir que los datos serán cargados correctamente (de hecho, también se debería validar que los valores de los lados no sean negativos ni cero).

En cuanto a detalles de estilo, observe que se mantuvo en el script la variable `__author__` asignada con el nombre del autor del programa (que *PyCharm* inserta por default, aunque eso depende de la versión de *PyCharm* que esté usando). También note el uso de líneas de *comentarios de texto* en distintos lugares del script para hacer más clara su lectura y su comprensión, así como el uso de líneas en blanco para separar bloques de instrucciones afines (aportando también claridad). El script arranca mostrando un título básico que hace más amigable y comprensible su uso por parte del usuario. **Y por supuesto, todo el programa está correctamente indentado (encolumnado): no podría ejecutarse si no fuese así.**

Veamos ahora el segundo ejemplo:

**Problema 2.)** *Se conoce la cantidad de horas que trabaja un empleado en una fábrica, más el importe que percibe por cada hora trabajada, además del nombre del empleado. Se pide calcular el importe final del sueldo que el empleado deberá cobrar y mostrar el nombre del empleado y el importe final del sueldo que se calculó.*

a.) Identificación de componentes:

- **Resultados:** Nombre del empleado. (*nom*: cadena de caracteres)  
Sueldo final. (*sueldo*: número en coma flotante)
- **Datos:** Nombre del empleado. (*nom*: cadena de caracteres)  
Horas trabajadas. (*horas*: número entero)  
Monto ganado por hora. (*monto*: número en coma flotante)
- **Procesos:** Problema de *gestión administrativa/contable*. El nombre del empleado debe ser informado en la consola de salida tal como se cargue en la variable *nom*. En cuanto al sueldo final (variable *sueldo*), se deduce claramente la operación a realizar: multiplicar la cantidad de horas trabajadas (variable *horas*) por el importe que se le paga al empleado por cada hora trabajada (variable *monto*). Por lo tanto, la fórmula o expresión será: *sueldo = horas \* monto*. De nuevo, en una situación real el programador debería validar que los valores cargados en *horas* y *monto* sean *mayores a cero*, pero por ahora no nos preocuparemos por ese detalle: asumiremos que el usuario cargará correctamente ambos datos.

b.) Planteo del algoritmo (como script o programa en *Python*): Asegúrese de poder agregar este programa como parte del *mismo proyecto que creó para el Problema 1*... ¡No necesitas crear un nuevo proyecto para cada programa o script que desarrolle!

```
__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 2 - Cálculo del sueldo de un empleado')
nom = input('Ingrese el nombre del empleado: ')
```

```

horas = int(input('Ingrese la cantidad de horas trabajadas: '))
monto = float(input('Ingrese el monto a cobrar por hora: '))

# procesos...
sueldo = horas * monto

# visualización de resultados...
print('Empleado: ', nom, '- Sueldo a cobrar:', sueldo, 'pesos')

```

En este script nuevamente tenemos una traducción directa a Python del análisis de procesos que se hizo en el paso anterior. El enunciado no dice formalmente que el sueldo final es el resultado de la multiplicación entre las horas trabajadas y el monto a pagar por hora, pero se deduce fácilmente del contexto. Por otra parte, no es extraño que se solicite mostrar en la pantalla de salida algún dato original (como el nombre) sin cambios. Otra vez, note el uso de comentarios de texto, líneas en blanco separadoras, la visualización de un título general, el uso de mensajes claros al cargar datos y al mostrar los resultados, y la **correcta indentación del script para evitar errores de intérprete**.

Nuestro tercer ejemplo es el que sigue:

**Problema 3.)** *Se tiene registrada la temperatura ambiente medida en tres momentos diferentes en un depósito de químicos y se necesita calcular el valor promedio entre las temperaturas medidas, tanto en formato entero (sin decimales) como en formato real (con decimales).*

**a.) Identificación de componentes:**

- **Resultados:** Promedio entero. *(prom1: número entero)*  
Promedio real. *(prom2: número en coma flotante)*
- **Datos:** Tres temperaturas. *(t1, t2, t3: números enteros)*
- **Procesos:** Problema de *aritmética elemental*. El enunciado no especifica si las tres temperaturas serán números enteros o de coma flotante, por lo que asumimos que serán enteros. El cálculo del promedio es simplemente la suma de las tres temperaturas  $t_1$ ,  $t_2$  y  $t_3$ , dividido por 3. Como se piden dos promedios, podemos almacenar la suma en una variable auxiliar ( $suma = t_1 + t_2 + t_3$ ) y luego dividir  $suma$  por 3. El cociente entero puede calcularse en Python con el operador  $//$  (y entonces sería  $prom1 = suma // 3$ ) mientras que el cociente real (o de coma flotante) puede calcularse con el operador  $/$  (o sea,  $prom2 = suma / 3$ ). Note que en este problema, los valores de las temperaturas podrían ser negativos, cero o positivos, por lo que incluso en una situación real no habría tanto problema con la validación de los datos que se carguen.

**b.) Planteo del algoritmo (como script o programa en Python):** Agregue este programa como parte del mismo proyecto que creó para el *Problema 1*:

```

__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 3 - Cálculo de la temperatura promedio')
t1 = int(input('Temperatura 1: '))
t2 = int(input('Temperatura 2: '))
t3 = int(input('Temperatura 3: '))

# procesos...

```

```

suma = t1 + t2 + t3
prom1 = suma // 3
prom2 = suma / 3

# visualización de resultados...
print('Promedio entero:', prom1, '- Promedio real:', prom2)

```

Este script sigue siendo elemental, pero es un poco más extenso que los dos ejemplos anteriores. Aquí, la secuencia de instrucciones que constituye el proceso de los datos, incluye tres operaciones: la suma y los dos cocientes. Otra vez, note el uso de comentarios de texto, líneas en blanco separadoras, la visualización de un título general, el uso de mensajes claros al cargar datos y al mostrar los resultados, y la **correcta indentación del script para evitar errores de intérprete**.

Observe cómo el uso de la variable auxiliar *suma* (no necesariamente prevista en el paso de identificación de componentes) ayuda a un planteo más claro y sin redundancia de operaciones: sin esa variable, el programador debería calcular la suma dos veces para calcular los dos promedios. Por otra parte, a modo de adelanto del tema que veremos en la próxima sección, consideremos solamente el cálculo del promedio real entre los valores de las variables *t1*, *t2* y *t3* pero sin el uso de la variable auxiliar. El promedio es la suma de los tres valores, dividido por 3 y en principio el cálculo en Python podría escribirse (*ingenuamente...*) así:

$$p = t1 + t2 + t3 / 3$$

*Lo anterior es un error*: así planteada la expresión, lo que se divide por 3 es sólo el valor de la variable *t3* y **no** la suma de las tres variables. Así escrita, la asignación anterior es equivalente a:

$$p = t1 + t2 + (t3 / 3)$$

que de ninguna manera calcula el promedio pedido. La forma correcta de escribir la instrucción, debió ser:

$$p = (t1 + t2 + t3) / 3$$

## 2.] Técnicas de representación de algoritmos: Diagramas de flujo.

Hemos indicado que el paso previo para el desarrollo de un programa es el planteo de un algoritmo que muestre la forma de resolver el problema que se está enfrentando. Los problemas que hemos analizado eran lo suficientemente simples como para que el algoritmo fuese directo: unas pocas consideraciones respecto del proceso a realizar y luego se pudo escribir el programa sin dificultades adicionales.

Sin embargo, en la gran mayoría de los casos, los problemas o requerimientos que enfrenta un programador no son tan sencillos ni tan directos. La lógica detrás del algoritmo puede ser compleja, con demasiadas ramas lógicas derivadas del uso de condiciones (que veremos) o demasiado cargada con situaciones especiales. Incluso un programador experimentado podría tener dificultades para captar la estructura de esa lógica en casos complejos, o bien, podría querer transmitir aspectos del algoritmo a otros programadores sin entrar en demasiado detalle de complejidad, y en este caso el idioma hablado resulta ambiguo.

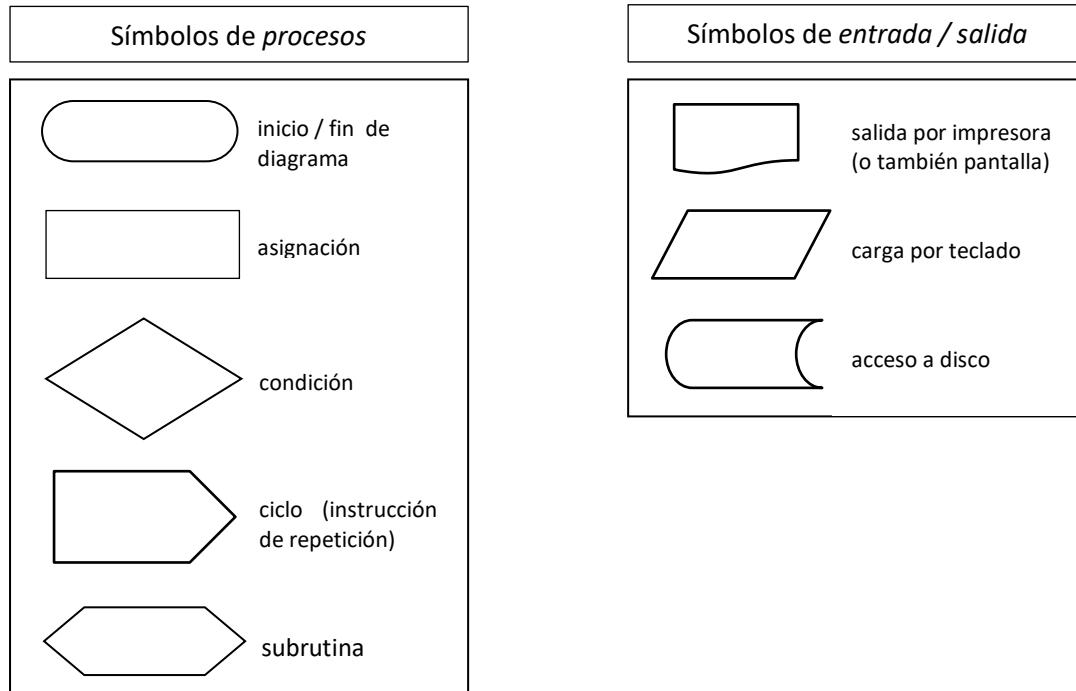
En casos así suele ser útil contar con herramientas de representación general de algoritmos, tales como gráficos, que permitan que un programador pueda rescatar y reflejar con más claridad la lógica general del algoritmo, sin tener que preocuparse (por ejemplo) de los elementos de sintaxis específica de un lenguaje [1].

Los *diagramas de flujo* y el *pseudocódigo* son dos de estas técnicas que ayudan a hacer unívoca la representación del algoritmo. Cada una de ambas, a su vez, admite innumerables variantes que no deben preocupar demasiado al estudiante: siempre recuerde que se trata de *técnicas auxiliares* en el trabajo de un programador, que normalmente serán usadas a modo de borrador o ayuda circunstancial, y podrá adaptarlas a sus preferencias (salvo en casos en que el programador trabaje en un contexto en el cual se hayan acordado reglas especiales para el planteo de un diagrama o de un pseudocódigo, y aun así, esas reglas no significarán un problema).

Un *diagrama de flujo* es un gráfico que permite representar en forma clara y directa el algoritmo para resolver un problema. Se basa en el uso de unos pocos símbolos unidos por líneas rectas descendentes. Los símbolos de un *diagrama de flujo* pueden clasificarse en dos grupos, y los más comunes de ellos se pueden ver en la *Figura 3*:

- ✓ **Símbolos de representación de procesos:** cada uno representa una operación para transformar datos en resultados. A modo de ejemplo, el símbolo usado para representar instrucciones de asignación es un rectángulo y forma parte de este grupo.
- ✓ **Símbolos de representación de operaciones de entrada y/o salida:** cada símbolo representa una acción o instrucción mediante la cual se lleva a cabo alguna operación de carga de datos o salida de resultados a través de algún dispositivo. En nuestros diagramas, se usarán para representar instrucciones de carga o entradas por teclado (mediante la función `input()` de Python 3) y también para las invocaciones a la función de salida por consola estándar `print()` de Python 3.

**Figura 3: Principales símbolos usados en un diagrama de flujo.**



El símbolo que hemos indicado como *carga por teclado* originalmente se usa tanto para indicar una *entrada genérica* como una *salida genérica*, en forma indistinta. Aquí, *genérica* significa que no es necesario en ese momento indicar en forma precisa el tipo de dispositivo a usar y sólo importa rescatar que se espera una operación de entrada o de salida. Sin

embargo, para evitar ambigüedad, aquí usaremos este símbolo *sólo para indicar una entrada* (no una salida), normalmente desde el teclado.

Y en el mismo orden de cosas, el símbolo que hemos indicado como *salida por impresora* originalmente sólo se usa para eso. Pero en este contexto lo admitiremos también para indicar una salida por pantalla o consola estándar. Con este par de convenciones, tratamos de evitar confusiones en cuanto al uso de símbolos que representen carga o visualización en un diagrama, pero no lo olvide: un diagrama de flujo es una herramienta auxiliar y adaptable a las necesidades del programador. Si se trabaja en un contexto colaborativo, simplemente deben pactarse las convenciones de representación y facilitar así el trabajo del equipo [1].

A modo de ejemplo de aplicación, considere el siguiente problema simple:

**Problema 4.)** *Se conoce la cantidad total de personas que padecen cierta enfermedad en todo el país, y también se sabe cuántas de esas personas viven en una ciudad determinada. Se desea saber qué porcentaje representan estas últimas sobre el total de enfermos del país.*

**a.) Identificación de componentes:**

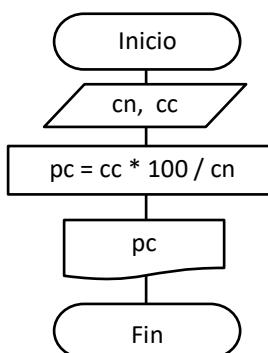
- **Resultados:** Porcentaje de enfermos de la ciudad con respecto al total nacional. ( $pc$ : número de coma flotante (o *real*))
- **Datos:** Cantidad de enfermos (nacional). ( $cn$ : número entero)  
Cantidad de enfermos (ciudad). ( $cc$ : número entero)
- **Procesos:** Problema de *cálculo de porcentaje en contexto de salud humana*. El cálculo del porcentaje surge de una regla de tres: La cantidad de enfermos de todo país ( $cn$ ) equivale al 100% del total y la cantidad de enfermos en la ciudad ( $cc$ ) equivale al porcentaje pedido  $pc$ , que entonces sale de:  $pc = cc * 100 / cn$ .

La expresión anterior para el cálculo del porcentaje no incluye sumas ni restas, y los dos operadores que aparecen (*producto* y *división real*) son de la misma precedencia, por lo que primero se aplicará el producto, y luego la división. Pero en términos del resultado final, sería lo mismo dividir primero y multiplicar después, lo que podría expresarse así:  $pc = cc * (100 / cn)$ .

Note que en una situación real, el programador debería incluir algún tipo de validación al cargar los valores  $cn$  y  $cc$ , para evitar que se ingresen negativos.

**b.) Planteo del algoritmo:** Si bien este problema es tan sencillo que sólo con lo anterior ya bastaría para pasar directamente al programa, mostramos aquí el algoritmo en forma de *diagrama de flujo* para comenzar a ejercitarse con esa herramienta:

**Figura 4: Diagrama de flujo del problema de cálculo del porcentaje.**



Como se ve, el diagrama se construye y se lee *de arriba hacia abajo*. Los símbolos que lo componen se unen entre sí mediante *líneas rectas verticales*, y como la lectura es descendente, se asume que esas líneas de unión también lo son: no es necesario hacer que terminen en punta de flecha (aunque podría hacerse si el programador lo desea).

El diagrama comienza y termina con el símbolo de *inicio / fin de diagrama* (ver *Figura 3*). En el inicial se escribe la palabra *Inicio* (o alguna equivalente) y en el de cierre se escribe la palabra *Fin* (o equivalente). Para indicar que se espera la carga por teclado de los valores de las variables *cn* y *cc* se usa el paralelogramo indicado como símbolo de carga por teclado (ver *Figura 3*), escribiendo dentro de él los nombres de ambas variables. El propio cálculo del porcentaje se escribe dentro del rectángulo que representa una asignación. Y el *símbolo de salida por impresora o pantalla* se usa para indicar que se espera mostrar el valor final de la variable cuyo nombre se inscribe dentro de él (en este caso, la variable *pc*).

Note que un diagrama de flujo *es genérico*: se plantea de forma que (en lo posible) no se haga referencia a lenguaje de programación alguno. La idea es que una vez planteado el diagrama, el programador escriba el programa usando el lenguaje que quiera. No hay (no debería haber...) diagramas de flujo para Python ni diagramas de flujo para Java o Basic o Pascal... *Un único diagrama debe servir de base para un programa en cualquier lenguaje*<sup>1</sup>.

En general, un diagrama de flujo *no debe incluir indicaciones de visualización de mensajes en pantalla*, salvo que eso de alguna manera ayude a entender la lógica general. Por caso, note que en el diagrama anterior se indica la carga por teclado las variables *cn* y *cc*, pero no se incluyen los mensajes que acompañan a la carga (como "*Ingrese el total nacional:*" o "*Ingrese el total de la ciudad:*"). Lo mismo vale en la visualización del resultado final *pc*: sólo se indica el nombre de la variable a mostrar, y se omiten los mensajes auxiliares [1].

- c.) **Desarrollo del programa:** Si se tiene un diagrama de flujo bien estructurado, el script o programa se deduce en forma prácticamente directa:

```
__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 4 - Cálculo del porcentaje de enfermos')
cn = int(input('Cantidad de enfermos en el país: '))
cc = int(input('Cantidad de enfermos en la ciudad: '))

# procesos...
pc = cc * 100 / cn

# visualización de resultados...
print('Porcentaje de la ciudad sobre el total país:', pc, '%')
```

### 3.] Técnicas de representación de algoritmos: Pseudocódigo.

Otra técnica muy usada por los programadores para representar genéricamente un algoritmo, es la que se conoce como *pseudocódigo*. Se trata de una herramienta por medio de la cual se plantea el algoritmo sin usar gráficas, escribiendo cada acción o paso en *lenguaje natural o cotidiano*, sin entrar en detalles sintácticos de ningún lenguaje aunque

---

<sup>1</sup> No obstante, los programadores pueden tomarse algunas licencias y relajar esta regla. Si se sabe a ciencia cierta que el lenguaje será Python, por ejemplo, entonces el diagrama de flujo podría incluir instrucciones con sintaxis específica de Python (como *y = x \*\* 2*) y nadie debería molestararse o sorprenderse por ello.

respetando ciertas pautas básicas (como la indentación). A diferencia de un programa terminado, el *pseudocódigo* está pensado para ser leído y entendido por una persona, y no por un computador.

Dado que el programa fuente terminado en un lenguaje específico se conoce como el *código fuente*, la palabra *pseudocódigo* hace referencia a una forma de *código incompleta o informal* (o sea, un código que no llega a ser completo ni riguroso).

El nivel de profundidad que se refleja en el *pseudocódigo* depende de la necesidad del programador, así como el nivel de formalismo o estructuración en el planteo de cada paso. A modo de ejemplo, el algoritmo para el mismo *problema 4* de la sección anterior (que ya resolvimos con el diagrama de flujo de la *Figura 4*) podría replantearse mediante el siguiente esquema de *pseudocódigo*:

Algoritmo:

1. Cargar en *cn* la cantidad de enfermos del país
2. Cargar en *cc* la cantidad de enfermos de la ciudad
3. Calcular y asignar en *pc* el porcentaje:  $pc = cc * 100 / cn$
4. Mostrar el porcentaje *pc*

Está claro que tratándose de un problema tan sencillo, es posible que no sea necesaria tanta minuciosidad o detalle en la descripción informal de cada paso. Si se desea claridad en el significado de cada variable o cálculo, lo anterior es aceptable. Pero en muchos casos, el siguiente replanteo más reducido puede ser suficiente (si los programadores conocen el contexto del problema):

Algoritmo:

1. Cargar *cn*
2. Cargar *cc*
3. Sea  $pc = cc * 100 / cn$
4. Mostrar *pc*

Como se ve, quien plantea el pseudocódigo dispone de mucha libertad para expresar cada paso. *No hay un estándar general* y las convenciones y reglas de trabajo pueden variar mucho entre programador y programador, pero en muchos ámbitos se suelen aceptar las siguientes (que son las que básicamente emplearemos en este curso cuando se requiera el planteo de pseudocódigo)<sup>2</sup>:

- ✓ Comenzar indicando en la primera línea el *nombre del proceso* que se está describiendo, seguido de *dos puntos* (como veremos, esto es típico de Python). Si no se conoce el nombre del proceso, escriba un descriptor genérico (*Algoritmo* en nuestro caso).
- ✓ Enumerar cada paso en forma correlativa. Si hubiese *subpasos*, enumerarlos en base al paso principal<sup>3</sup>.

---

<sup>2</sup> Obviamente, existen muchas otras convenciones y reglas y tanto el estudiante como sus profesores pueden aplicar las que deseen.

<sup>3</sup> Veremos en una Ficha posterior que una instrucción puede estar compuesta por otras instrucciones incluidas en ella, lo cual la hace una *instrucción compuesta*. En contrapartida, una *instrucción simple* (como una asignación o una visualización o una carga) no contiene instrucciones incluidas.

Algoritmo:

1. Paso 1...
2. Paso 2...
  - 2.1. Subpaso 2.1...
  - 2.2. Subpaso 2.2...
3. Paso 3...

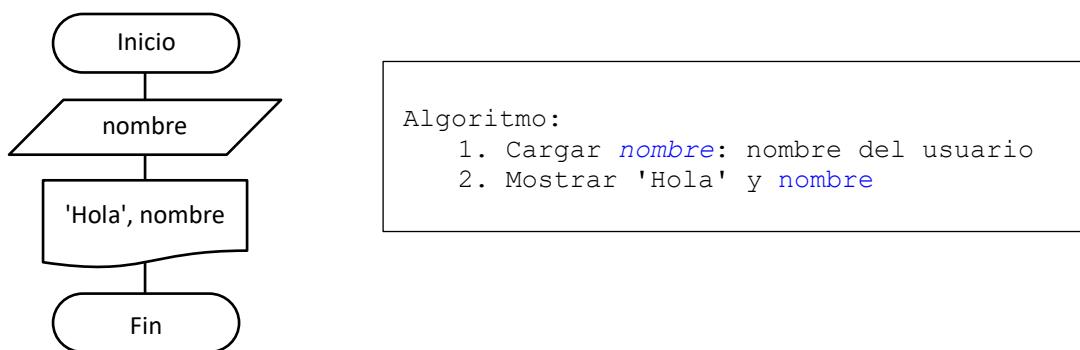
- ✓ Mantener el encolumnado o indentación general, tanto a nivel de pasos principales como a nivel de *subpasos* (ver ejemplo anterior).
- ✓ Mantener consistencia: si tiene (por ejemplo) dos pasos de carga, designe a ambos de la misma forma y con el mismo estilo. El siguiente replanteo del pseudocódigo que vimos, *no respeta la consistencia*:

1. Cargar la variable *cn*
2. Ingrese en *cc* un valor
3. Sea *pc* = *cc* \* 100 / *cn*
4. Mostrar *pc*

Como dijimos, cada programador decide la forma y el nivel de detalle y profundidad de su pseudocódigo. En muchos casos, se usa el criterio de plantear pseudocódigos usando una estructura análoga a la del lenguaje que se sabe que se usará luego para el desarrollo del programa. En general, mientras más se basa un pseudocódigo en las reglas y convenciones sintácticas de un lenguaje en particular, *más estructurado se dice que es ese pseudocódigo*. En nuestro caso, a lo largo del curso emplearemos pseudocódigo con bastante libertad, aunque siguiendo una estructura básica inspirada en Python.

Como ya se dijo, tanto los diagramas de flujo como el pseudocódigo son técnicas auxiliares de representación de algoritmos. En teoría, al plantear un diagrama o un pseudocódigo se hace de forma genérica, tratando de no apegarse a sintaxis ni convenciones de ningún lenguaje, aunque en la práctica esto se relaja, dado que los programadores saben en qué lenguaje trabajarán. Pero aun así, un mismo diagrama y/o un mismo pseudocódigo debe plantearse de forma que pueda servir como base para un programa en cualquier lenguaje. A modo de ejemplo, si se quiere escribir un programa que cargue por teclado el nombre de una persona y a continuación le muestre un saludo por consola estándar, podría tener la siguiente estructura algorítmica elemental planteada como diagrama y como pseudocódigo:

**Figura 5: Diagrama y pseudocódigo para el problema de cargar el nombre y mostrar un saludo**



El diagrama y el pseudocódigo que se ven arriba, sirven sin problemas para guiar el desarrollo de un programa en el lenguaje que prefiera el programador. Como ejemplo, los

siguientes son cuatro programas en cuatro lenguajes diferentes (*Pascal, Java, C++ y Python*), que implementan el algoritmo reflejado en el diagrama y el pseudocódigo [2]:

**Figura 6: El mismo algoritmo en cuatro programas con diferentes lenguajes.**

<u>Pascal:</u>	<u>Java:</u>
<pre>Program Hola; Uses CRT; Var   nombre: String[40]; Begin   WriteLn("Ingrese su nombre: ");   ReadLn(nombre);   WriteLn("Hola ", nombre); End.</pre>	<pre>import java.util.Scanner; public class Hola {     public static void main(String [] args)     {         Scanner sc = new Scanner(System.in);         String nombre;          System.out.println("Ingrese su nombre: ");         nombre = sc.nextLine();         System.out.println("Hola " + nombre);     } }</pre>
<u>C++:</u>	<u>Python:</u>
<pre>#include&lt;iostream&gt; using namespace std; int main() {     char nombre[20];      cout &lt;&lt; "Ingrese su nombre: ";     cin &gt;&gt; nombre;     cout &gt;&gt; "Hola " &gt;&gt; nombre &gt;&gt; endl;     return 0; }</pre>	<pre>nombre = input("Ingrese su nombre: ") print("Hola ", nombre)</pre>

#### 4.] Precedencia de operadores y uso de paréntesis en una expresión.

Hemos visto que una *expresión* es una fórmula que combina valores (constantes o variables) designados como *operando*s, y símbolos de operaciones (suma, resta, etc.) designados como *operadores*. Cuando el intérprete Python encuentra una expresión, *la evalúa* (es decir, obtiene el resultado de la fórmula) y deja disponible el resultado para ser almacenado en una variable o para ser visualizado en consola de salida o para el propósito que disponga el programador: no es obligatorio que el resultado de una expresión se asigne en una variable. Considere el siguiente ejemplo:

```
a = int(input('A: '))
b = int(input('B: '))

# 1.) expresión asignada en una variable...
c = a + 2*b
print('C:', c)

# 2.) expresión directamente visualizada (sin asignación previa)...
print('D:', 3*a - b/4)

# 3.) expresión libre... el resultado se pierde...
a + b + a*b
```

En el script anterior, el bloque 1.) muestra la forma clásica de usar y asignar una expresión: el intérprete evalúa primero el *miembro derecho* de la asignación ( $a + 2*b$ ), obtiene el resultado, y asigna el mismo en la variable *c*. En el bloque 2.) aparece otra expresión ( $3*a - b/4$ ) la cual también es evaluada por el intérprete, pero al obtener el resultado el mismo es enviado directamente a la función *print()* para que se muestre en consola de salida (una vez

visualizado, el valor se pierde: ninguna variable retiene ese valor). Y en el bloque 3.) se muestra una expresión correctamente escrita ( $a + b + a * b$ ) que será efectivamente evaluada por el intérprete sin producir error alguno, pero el valor obtenido no será visualizado y simplemente se perderá ya que no fue asignado en ninguna variable. Puede parecer que esto último no tiene sentido ni utilidad, pero por ahora lo rescatamos sólo como una posibilidad válida.

En general, si el resultado de una expresión es un número, entonces esa expresión se conoce como una *expresión aritmética*. Las tres expresiones que mostramos en el ejemplo anterior, son *aritméticas*. Note que es perfectamente posible que el resultado de una expresión sea un valor lógico (o *booleano*: *True* o *False*) en cuyo caso la expresión se conoce como *expresión lógica*; y aun más: una expresión podría dar como resultado una cadena de caracteres o algún otro tipo compuesto de resultado (en estos casos, no hay nombres específicos para ellas... son simplemente expresiones). Volveremos sobre las expresiones lógicas y las expresiones en general en Fichas posteriores.

Para resaltar la importancia del tema siguiente, mostramos ahora otros ejemplos de expresiones aritméticas. Suponga que las variables *a* y *b* son inicializadas con los valores 10 y 7 respectivamente, Tómese un par de minutos para hacer un análisis minucioso de cada una de las cuatro expresiones que siguen y trate de predecir el valor que terminará asignándose en cada una de las variables *c*, *d*, *e* y *f*:

```
a, b = 10, 7

c = a + 10 - 5 * b + 4
d = 3 * a + 1 - b // 4 + a % 2
e = 2 * a + b - 1 + a ** 2
f = a + b / a - b
```

La respuesta correcta es  $c = -11$ ,  $d = 30$ ,  $e = 126$  y  $f = 3$  (lo que puede fácilmente verificarse ejecutando las instrucciones una por una en el shell y mostrando sus resultados). Si no llegó a estos mismos resultados, muy posiblemente se deba a que aplicó en forma incorrecta lo que se conoce como *precedencia de ejecución* de los operadores [3] [4]. En todo lenguaje de programación, los operadores de cualquier tipo tienen diferente prioridad de ejecución en una expresión, y a esa prioridad se la llama *precedencia de ejecución*.

Cuando el intérprete Python analiza una expresión (en este caso aritmética) los operadores de *multiplicación* (\*), *división* (//, /), *resto* (%) y *potencia* (\*\*) se aplican antes que los operadores de *suma* (+) y *resta* (-) y se dice entonces que los operadores \*, //, /, % y \*\* tienen *mayor precedencia* que los operadores + y -. Obtenidos todos los resultados de los operadores de precedencia mayor, se aplican sobre esos resultados los operadores de precedencia menor. Por lo tanto, si  $a = 10$  y  $b = 7$  entonces la expresión

```
c = a + 10 - 5 * b + 4
```

aplicará primero la multiplicación  $5 * b$  (con resultado igual a 35) y luego hará las sumas y las restas, llegando al resultado final de -11 (obviamente, la suma  $a + 10$  se ejecuta directamente ya que no hay operadores de precedencia mayor en ella):

```
c = 20 - 35 + 4
c = -11
```

Exactamente el mismo resultado se obtendría si el producto  $5 * b$  fuese encerrado entre paréntesis (y por lo tanto, si eso es lo que quería obtener el programador, los paréntesis no son necesarios en esta expresión):

```
c = a + 10 - (5 * b) + 4
c = 20 - 35 + 4
c = -11
```

Si dos operadores tienen el mismo nivel de precedencia, se ejecutan y aplican en orden de aparición, de izquierda a derecha (y esto se conoce como *precedencia izquierda*). En el ejemplo que sigue, asumamos las variables  $a = 2$ ,  $b = 4$ ,  $c = 5$  y  $d = 3$ :

$$e = a * b // c * d$$

Como los operadores `*` y `//` tienen el mismo nivel de precedencia y aparecen en la misma expresión, *se aplican de izquierda a derecha*: primero se hace  $a * b$ , con resultado 8. Ese 8 se divide en forma entera por  $c$ , con resultado 1, y ese 1 se multiplica por  $d$ , con resultado final (guardado en  $e$ ) de 3.

La excepción a esta regla es el operador `**` (para exponenciación). Este operador tiene *precedencia derecha*. Si aparece en la misma expresión que otros operadores del mismo nivel (`*`, `/`, `//`, `%`) entonces se aplica primero `**` y luego los demás. Veamos la siguiente expresión (asuma los mismos valores que en el ejemplo anterior):

$$e = a * b * c ** 2 * d$$

Aquí se calcula primero  $c ** 2$  (que da 25), y luego se aplican las multiplicaciones, *todas de izquierda a derecha*, con resultado  $e = 600$ .

Los paréntesis siempre pueden usarse para cambiar la precedencia de ejecución de algún operador [3] y lograr resultados diferentes según se necesite. *Cualquier expresión encerrada entre paréntesis se ejecutará primero*, y al resultado obtenido se le aplicarán los operadores que queden según sus precedencias.

Si se usan en la forma indicada en el ejemplo anterior, los paréntesis no son necesarios ya que en forma natural el producto  $5 * b$  se ejecutaría primero de todos modos. Pero si el programador hubiese querido multiplicar la suma  $b + 4$  por 5, entonces **debería usar paréntesis para alterar la precedencia**, en la forma siguiente:

```
c = a + 10 - 5 * (b + 4)
c = 20 - 5 * 11
c = 20 - 55
c = -35
```

Del mismo modo, si las expresiones originales:

```
d = 3 * a + 1 - b // 4 + a % 2
e = 2 * a + b - 1 + a ** 2
f = a + b // a - b
```

se reescriben así:

```
d = (3 * a + 1 - b) // (4 + a) % 2
e = 2 * (a + b) - (1 + a) ** 2
f = (a + b) // (a - b)
```

entonces con  $a = 10$  y  $b = 7$  los resultados obtenidos serían:

```
d = (3 * 10 + 1 - 7) // (4 + 10) % 2
d = (30 + 1 - 7) // 14 % 2
d = 24 // 14 % 2    # primero el de más a la izquierda...
d = 1 % 2
d = 1
```

```

e = 2 * (10 + 7) - (1 + 10) ** 2
e = 2 * 17 - 11 ** 2
e = 34 - 121
e = -87

f = (10 + 7) // (10 - 7)
f = 17 // 3
f = 5

```

Asegúrese de comprender cada secuencia de cálculo que hemos mostrado. Recuerde que la suma y la resta tienen precedencia menor pero que esa precedencia puede alterarse en forma arbitraria si se usan paréntesis.

En base a lo expuesto hasta aquí, debe entenderse que el conocimiento acerca de la precedencia de los operadores y el correcto uso de paréntesis resulta imprescindible para evitar cometer errores en el planteo de una expresión: en muchas ocasiones los programadores escriben una fórmula *que no querían* por aplicar mal estos conceptos y sus programas entregan resultados totalmente diferentes de los que se esperaría.

Para cerrar esta sección, mostramos la forma de escribir en Python las instrucciones que corresponden a las siguientes fórmulas (o ecuaciones) generales:

**Figura 7: Uso de paréntesis para el planteo en Python de distintas expresiones aritméticas comunes.**

Fórmula general	Expresión en Python	Observaciones
$p = \frac{c1 + c2 + c3}{3}$	$p = (c1 + c2 + c3)/3$	<ul style="list-style-type: none"> <li>Promedio real <b>p</b> de los valores <b>c1</b>, <b>c2</b> y <b>c3</b>.</li> </ul>
$s = \frac{n * (n + 1)}{2}$	$s = (n*(n+1))/2$	<ul style="list-style-type: none"> <li>Suma <b>s</b> de todos los números naturales del 1 al <b>n</b>.</li> <li>Puede demostrarse <b>por inducción</b> [5].</li> </ul>
$h = \sqrt{a^2 + b^2}$	$h = (a**2 + b**2)**0.5$	<ul style="list-style-type: none"> <li>Longitud de la hipotenusa <b>h</b> de un triángulo rectángulo con catetos <b>a</b> y <b>b</b>.</li> <li>Recuerde que calcular <math>\sqrt{n}</math> es lo mismo que calcular <math>n^{0.5}</math> que es <math>n**0.5</math> en Python.</li> </ul>
$p = \frac{y2 - y1}{x2 - x1}$	$p = (y2 - y1)/(x2 - x1)$	<ul style="list-style-type: none"> <li>Pendiente <b>p</b> de la recta que pasa por los puntos <b>(x1, y1)</b> y <b>(x2, y2)</b>.</li> </ul>
$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$	$x = (-b + (b**2 - 4*a*c)**0.5) / (2*a)$	<ul style="list-style-type: none"> <li>Primera raíz <b>x</b> de la ecuación de segundo grado, con coeficientes <b>a</b>, <b>b</b> y <b>c</b> (con <b>a != 0</b>).</li> <li>Recuerde que calcular <math>\sqrt{n}</math> es lo mismo que calcular <math>n^{0.5}</math> que es <math>n**0.5</math> en Python.</li> </ul>

## 5.] Aplicaciones del resto de la división entera - Fundamentos de Aritmética Modular.

Hemos visto que Python provee el operador % (llamado *operador módulo* u *operador resto*) que permite calcular el resto de la división entre dos números. El estudio de las propiedades del resto

forma parte de una rama de las matemáticas designada con el nombre general de *aritmética modular*, y esas propiedades tienen aplicaciones especialmente útiles en muchos campos (entre ellos, la programación y las ciencias de la computación). En este anexo de *Temas Avanzados* para la Ficha 2, expondremos brevemente algunas definiciones y notaciones propias de la aritmética modular, para pasar luego a aplicaciones de utilidad inmediata en ciertos problemas de programación.

Como primera medida, enumeraremos algunas propiedades fundamentales del resto de la división entera (el conjunto de los *números enteros* se denota aquí como  $\mathbb{Z}$ , y recordemos que incluye a los naturales, el cero, y los negativos de los naturales):

- El *resto de la división entera* entre  $a$  (llamado el *dividendo*) y  $b$  (llamado el *divisor*) es el primer número entero  $r$  que queda como residuo de la división parcial, tal que  $0 \leq r < b$ . Este valor indica que la división ya no puede proseguir en forma entera: el residuo parcial obtenido no alcanza a ser dividido nuevamente en forma entera por  $b$ . Todos los lenguajes de programación proveen algún tipo de operador para calcular el resto. Como vimos, en Python es el operador % [3] [4].

Ejemplos:  $14 \% 3 = 2$        $15 \% 2 = 1$        $18 \% 5 = 3$

- Si se divide por  $n$  (con  $n \in \mathbb{Z}$ ) entonces pueden obtenerse exactamente  $n$  posibles restos distintos, que son los números en el intervalo  $[0, n-1]$ . El resto 0 se obtiene si la división es exacta. Pero si no es exacta, se obtendrá un resto que por definición es menor a  $n$  (pues de otro modo la división podría continuar un paso más). El valor del mayor resto posible es  $n - 1$ .

Ejemplos:

- ✓ El conjunto  $S$  de todos los restos posibles que se obtienen al dividir por  $n = 5$  contiene 5 valores:

$$S = \{0, 1, 2, 3, 4\}$$

- ✓ El conjunto  $T$  de todos los restos posibles que se obtienen al dividir por  $n = 100$  contiene 100 valores:

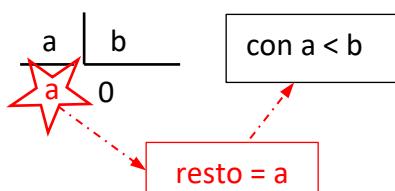
$$T = \{0, 1, 2, \dots, 98, 99\} = \{\forall r \in \mathbb{Z} \mid 0 \leq r \leq 99\}$$

- Si el número  $a$  es divisible por  $b$  (o lo que es lo mismo, si  $a$  es múltiplo de  $b$ ) entonces el resto de dividir  $a$  por  $b$  es cero.

Ejemplos:  $14 \% 2 = 0 \Rightarrow 14$  es divisible por 2  
 $17 \% 3 = 2 \Rightarrow 17$  *no* es divisible por 3

- Si el *dividendo*  $a$  es menor que el *divisor*  $b$  entonces el resto es *igual al dividendo*. Esto surge de la propia mecánica de la división entera. Si  $a < b$ , entonces el cociente  $a // b$  es 0. Para calcular el resto parcial, se multiplica ese cociente 0 por  $b$  (lo cual es 0), y el resultado se resta de  $a$ ... con lo cual es  $a - 0 = a$ . La división entera debe terminar cuando se encuentre el primer resto parcial que sea menor que  $b$ , y como  $a$  es efectivamente menor que  $b$ , aquí termina el proceso y el resto es igual al valor  $a$ , que era el dividendo.

Ejemplos:  $3 \% 5 = 3$   
 $13 \% 20 = 13$



La aritmética modular se construye a partir del concepto de *relación de congruencia* entre números enteros: Sean los números enteros  $a$ ,  $b$  y  $n$ . El número  $a$  se dice congruente a  $b$ , módulo  $n$ , si ocurre que  $(b - a)$  es múltiplo de  $n$  (o lo que es lo mismo, si  $(b - a)$  es divisible por  $n$ ). A su vez, esto es lo mismo que decir que los números  $a$  y  $b$  tienen el mismo resto al dividirlos por  $n$ . En símbolos, la relación de congruencia se denota de la siguiente forma [5]:

$$a \equiv b \pmod{n} \quad (\text{se lee: } a \text{ es congruente a } b, \text{ módulo } n)$$

#### Ejemplos:

- **$4 \equiv 8 \pmod{2}$**

Se lee: 4 es congruente a 8, módulo 2. Esto es efectivamente así, ya que  $(8 - 4) = 4$ , y 4 es divisible por 2. En forma alternativa, el resto de dividir por 2 tanto al 4 como al 8 es el mismo: 0(cero).

- **$17 \equiv 12 \pmod{5}$**

Se lee: 17 es congruente a 12, módulo 5. Efectivamente,  $(12 - 17) = -5$ , y -5 es divisible por 5. Alternativamente, el resto de dividir a 17 y a 12 por 5 es el mismo: 2(dos).

En base a las relaciones de congruencia, la aritmética modular permite clasificar a los números enteros en subconjuntos designados como *clases de congruencia (módulo  $n$ )*. Dos números  $a$  y  $b$  pertenecen a la misma clase de congruencia (módulo  $n$ ), si se obtiene el mismo resto al dividir los números  $a$  y  $b$  por  $n$ .

#### Ejemplos:

- ✓ Sea  $Z$  el conjunto de los números enteros. Entonces el subconjunto  $Z_{2_0}$  de todos los números pares es una clase de congruencia (módulo 2): contiene a todos los enteros que dejan un resto de 0 al dividirlos por 2 (es decir, todos los números enteros divisibles por 2):

$$Z_{2_0}: \{-6, -4, -2, 0, 2, 4, 6, \dots\} = \{2*k + 0 \mid k \in Z\}$$

- ✓ A su vez, el subconjunto  $Z_{2_1}$  de todos los números impares es otra clase de congruencia (módulo 2): contiene a todos los enteros que dejan un resto de 1 al dividirlos por 2 (o sea, todos los números enteros que no son divisibles por 2):

$$Z_{2_1}: \{-5, -3, -1, 1, 3, 5, 7, \dots\} = \{2*k + 1 \mid k \in Z\}$$

- ✓ Las posibles clases de congruencia (módulo 3) son las tres que siguen:

$$Z_{3_0}: \{-6, -3, 0, 3, 6, 9, 12, \dots\} = \{3*k + 0 \mid k \in Z\} \quad (\text{resto } = 0 \text{ al dividir por } 3)$$

$$Z_{3_1}: \{-4, -1, 1, 4, 7, 10, \dots\} = \{3*k + 1 \mid k \in Z\} \quad (\text{resto } = 1 \text{ al dividir por } 3)$$

$$Z_{3_2}: \{-2, 2, 5, 8, 11, 14, \dots\} = \{3*k + 2 \mid k \in Z\} \quad (\text{resto } = 2 \text{ al dividir por } 3)$$

Como se dijo (*propiedad ii del resto*) los posibles restos de dividir por  $n$  son los  $n$  valores del intervalo  $[0, n-1]$ . De allí que existan  $n$  clases de congruencia (módulo  $n$ ) distintas, dado el valor  $n$ .

De todo lo anterior, podemos ver que los diferentes restos de dividir por  $n$  se repiten cíclicamente: al dividir por  $n = 3$  (por ejemplo) y comenzando desde el 0, tenemos:

$0 \% 3 = 0$	$3 \% 3 = 0$	$6 \% 3 = 0$	... (en clase de congruencia $Z_{3_0}$ )
$1 \% 3 = 1$	$4 \% 3 = 1$	$7 \% 3 = 1$	... (en clase de congruencia $Z_{3_1}$ )
$2 \% 3 = 2$	$5 \% 3 = 2$	$8 \% 3 = 2$	... (en clase de congruencia $Z_{3_2}$ )

El nombre de *aritmética modular* proviene justamente de esta característica: el proceso de tomar el resto de dividir por  $n$  produce resultados cíclicamente iguales. El valor  $n$  se denomina *módulo* del proceso, ya que por definición, un módulo es una medida que se usa como norma para valorar objetos del mismo tipo (y en este caso, los objetos del mismo tipo son los números que pertenecen a la misma clase de congruencia).

La aritmética modular está presente en la vida cotidiana en al menos una situación muy conocida: la forma de interpretar un reloj analógico (de agujas). El cuadrante de un reloj de agujas está dividido en 12 secciones que indican las horas principales. Pero un día tiene 24 horas (y no 12). Por lo tanto, las primeras 12 horas del día pueden leerse en forma directa, pero las siguientes 12 (esto es, desde la 13 a la 24) deben interpretarse con *relación de congruencia módulo 12*. Así, la hora 13 corresponde (*o es equivalente*) a la hora 1 del reloj, ya que 13 y 1 son congruentes módulo 12 (o sea que la relación es  $13 \equiv 1 \pmod{12}$ ): ambos tienen el mismo resto (que es 1) al dividir por 12).

Entonces, ahora sabemos que 13 y 1 pertenecen a  $Z_{12_1}$  (la clase de congruencia módulo 12 con resto 1). Y cada par de horas del reloj (de agujas) separadas por 12, pertenece a una clase de congruencia módulo 12 diferente: 12 y 24 pertenecen a  $Z_{12_0}$ ; 13 y 1 pertenecen a  $Z_{12_1}$ ; 14 y 2 pertenecen a  $Z_{12_2}$  y así sucesivamente. Y el resto de la división por 12 es la hora que se debe mirar en el reloj.

Por otra parte, y siempre en el ámbito del manejo de unidades de tiempo<sup>4</sup>, una aplicación práctica simple del operador resto y las relaciones de congruencia es la de convertir una cierta cantidad inicial de segundos (*is*), a su correspondiente cantidad de horas (*ch*), minutos (*cm*) y segundos restantes (*cs*). Dado que el sistema horario internacional no tiene una subdivisión de base decimal (unidades de 10, como el sistema métrico) sino sexagesimal (unidades de 60, para los minutos y los segundos), la conversión requiere algo de trabajo con operaciones de regla de 3 y aplicaciones del resto. Analicemos esquemáticamente el proceso en base a un ejemplo:

1. Supongamos que la cantidad inicial de segundos es *is* = 8421 segundos.
2. Dado que una hora tiene 60 minutos, y cada minuto tiene 60 segundos, entonces una hora contiene  $60 * 60 = 3600$  segundos.
3. Por lo tanto, la cantidad de horas completas *ch* que hay en *is* segundos, surge del cociente entero entre *is* y 3600. En nuestro ejemplo, el cociente entero entre 8421 y 3600 es *ch* = 2 horas.
4. Ahora necesitamos saber cuántos segundos nos quedaron de los que inicialmente teníamos. En el ejemplo, teníamos *is* = 8421 segundos inicialmente, pero hemos tomado el equivalente a *ch* = 2 horas, que son  $3600 * 2 = 7200$  segundos. Es fácil ver que nos quedarían  $8421 - 7200 = 1221$  segundos para continuar el cálculo.
5. Sin embargo, notemos que no es necesario multiplicar *ch* \* 3600 para luego restar ese resultado desde *is*: si se observa con atención, el proceso que se acaba de describir está basado en una *relación de congruencia (módulo 3600)*: los múltiplos de 3600 (resto 0 al dividir por 3600) equivalen a las cantidades de horas completas disponibles. Y los totales de segundos que no son múltiplos de 3600 (como 8421) dejan un resto al dividir por 3600 que es justamente igual a la cantidad de segundos en exceso...
6. Por lo tanto, se puede saber cuál es la cantidad de horas *ch* que hay en *is* tomando el cociente entero de la división entre *is* y 3600 (como vimos), y podemos saber cuántos segundos quedan para seguir operando (*ts*) simplemente tomando el resto de la misma división. En Python puede hacer esas operaciones con el operador // para el cociente entero y el operador % para el resto. En nuestro caso, quedaría *ch* = 2, y *ts* = 1221.
7. Para saber la cantidad de minutos completos *cm* que pueden formarse con *ts* segundos (aquí *cm* sería la cantidad de minutos que no llegaron a formar una hora), se procede en forma similar: un

<sup>4</sup> En la Ficha 1 hicimos una referencia del mundo del cine para la historia de Alan Turing, y ahora que estamos tratando con unidades de tiempo hacemos otra, pero del campo de la ciencia ficción: la película *In Time* del año 2011 (conocida en Hispanoamérica como el "El Precio del Mañana"), dirigida por Andrew Niccol e interpretada por Justin Timberlake y Amanda Seyfried, trata sobre una imaginaria sociedad futura en la que el tiempo se ha convertido en la moneda de cambio, en lugar del dinero. Las personas entregan parte del tiempo que les queda de vida para comprar un café o para comprar ropa o un viaje... ¿Cuánto valdría un segundo de tu vida en una realidad como esa? ¿Qué harías si en tu "billetera" temporal quedasen sólo 5 o 6 segundos? ¿Cómo sería tu vida si sólo tuvieras 2 o 3 minutos disponibles en cada momento del día?

- minuto tiene 60 segundos, por lo que el cociente entero entre  $ts$  y 60 entrega la cantidad de minutos buscada. En nuestro caso, el cociente entero 1221 y 60 es  $cm = 20$  minutos.
8. La cantidad de segundos que queden después de este último cálculo, es la cantidad de segundos residuales  $cs$  que falta para completar el resultado: son los segundos remanentes que no alcanzaron para formar ni otra hora ni otro minuto. Y dado que este segundo proceso muestra una *relación de congruencia (módulo 60)*, entonces la cantidad remanente de segundos es el *resto de la división entre  $ts$  y 60* (en nuestro caso, el resto de la división entre 1221 y 60, que es  $cs = 21$  segundos).
  9. Resultado final para nuestro ejemplo: si se tiene una cantidad inicial de 8421 segundos, eso equivale a 2 horas, 20 minutos y 21 segundos.

Dejamos para el estudiante la tarea de implementar estas ideas en un script o programa Python completo.

## 6.] Uso de Entornos Integrados de Desarrollo (IDEs) para Python: El IDE PyCharm Edu.

Si bien está claro que se puede usar el shell de Python para editar, testear y ejecutar scripts y programas, el hecho es que el uso directo del shell tiene muchas limitaciones en cuanto a comodidad para el programador. El *IDLE Python GUI* que se usó a lo largo de la Ficha 01 permite escribir y ejecutar instrucciones una por una, y eventualmente también ejecutar un script de varias líneas, pero la tarea resulta incómoda (sobre todo si se está pensando en programas mucho más extensos y complejos).

En ese sentido, más temprano que tarde los programadores comienzan a usar lo que se conoce como un *Entorno Integrado de Desarrollo* (o *IDE*, tomando las siglas del inglés *Integrated Development Environment*), que no es otra cosa que un programa más sofisticado y profesional, que incluye herramientas que facilitan muchísimo el trabajo de desarrollar programas en cualquier lenguaje. Un *IDE* incluye un *editor de textos completo* (que en muchos casos viene provisto con *asistentes inteligentes para predicción de escritura y para ayuda contextual*), opciones de configuración de todo tipo, y herramientas de compilación, depuración y ejecución del programa que se haya escrito.

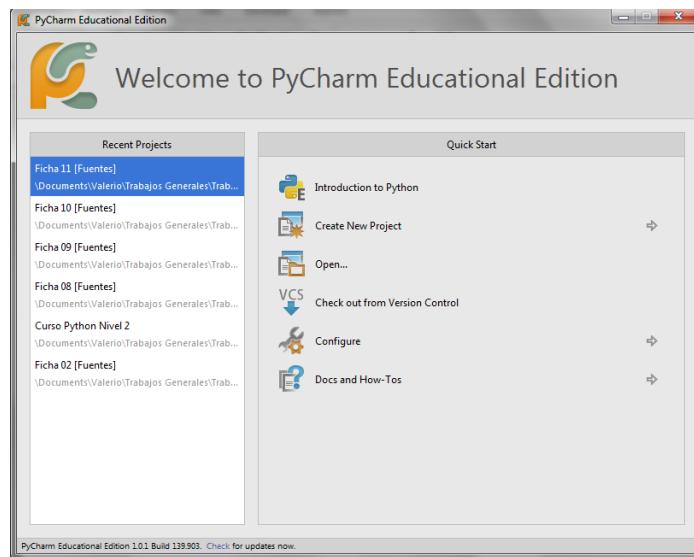
Existen numerosísimos *IDEs* para Python. Muchos de ellos son de uso libre y otros requieren la compra del producto y/o su licencia de uso. Algunos *IDEs* son simplemente editores de texto especializados para programación en múltiples lenguajes. En lo que respecta al desarrollo de la asignatura AED, y luego de un proceso de exploración y prueba que llevó cierto tiempo, nos hemos inclinado por el *IDE PyCharm* en su *versión educativa (PyCharm Edu)*, desarrollado por la empresa *JetBrains*. Las directivas de descarga e instalación han sido oportunamente provistas a los alumnos en un documento separado, dentro de los materiales subidos al aula virtual del curso en la primera semana (ver *Zona Cero* del aula virtual: *Instructivo de Instalación: Python – PyCharm*).

La versión *PyCharm Edu* es de uso libre, y aunque tiene limitaciones en cuanto a disponibilidad de recursos (con respecto a las versiones *Professional* y *Community*), el hecho es que dispone de algunos módulos y prestaciones que la hacen muy aplicable al contexto de un curso de introducción a la programación en Python.

El uso de *PyCharm Edu* es intuitivo, aunque llevará un poco de tiempo dominar la herramienta y lograr experiencia. Para comenzar a trabajar, veamos paso a paso la forma de escribir y ejecutar un programa simple a través de *PyCharm Edu*. Arranque el programa *PyCharm* desde el escritorio de su computadora. Si suponemos que es la primera vez que se ejecuta ese programa en esa computadora, usted verá una ventana de bienvenida similar a la que se muestra en la captura de pantalla de la *Figura 8*. Si no aparece esta ventana y en su lugar se ve directamente el escritorio de trabajo de *PyCharm*, se debe a que el *IDE* fue utilizado recientemente para desarrollar algún programa, y ese programa está cargado en ese momento (*PyCharm* automáticamente carga el último proyecto en el que se estaba trabajando antes de cerrar el *IDE*). Si este fue el caso, y sólo por esta vez para que

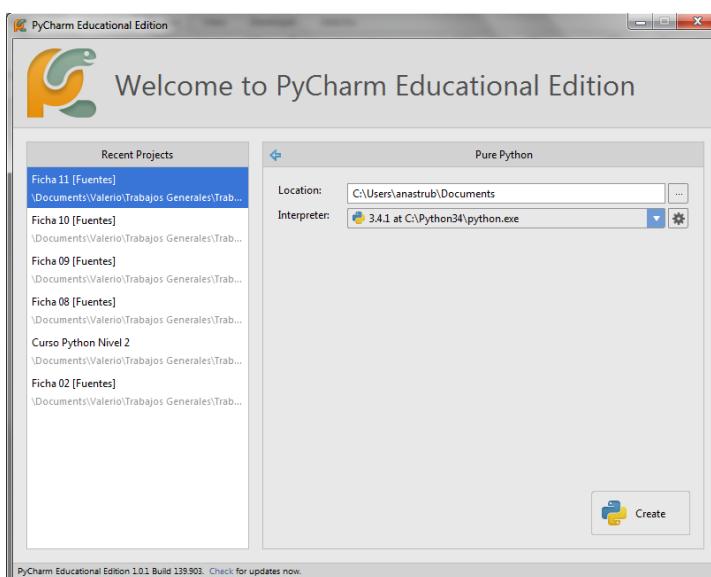
pueda seguir el proceso completo desde el inicio, busque en la barra de opciones del menú de *PyCharm* la opción *File*, y dentro del menú desplegable que se abre elija ahora *Close Project*. Esto cerrará el proyecto y luego de un instante verá la ventana de bienvenida.

**Figura 8:** Ventana de bienvenida de PyCharm.



En el panel de la izquierda (*Recent Projects*) de esta ventana verá una lista de los proyectos recientemente abiertos, y en el panel de la derecha (*Quick Start*) se muestra un conjunto de opciones para comenzar a trabajar. Por ahora, la única que nos interesa es la opción *Create New Project*. Al hacer click en ella, el panel de la derecha cambia para mostrar los diferentes *tipos de proyectos* que puede crear, que normalmente son *Pure Python*, *Course creation* y *Educational*. Seleccione *Pure Python* y en ese momento el panel derecho vuelve a cambiar, mostrando un aspecto similar al de la captura de pantalla en la *Figura 9*.

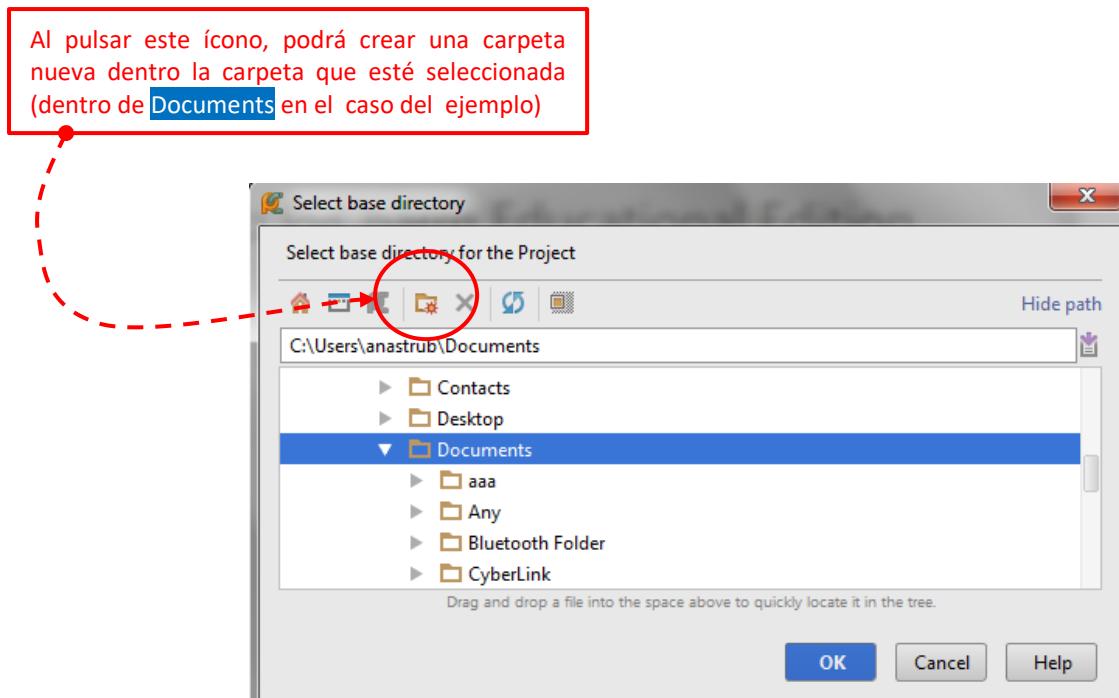
**Figura 9:** Ventana de bienvenida de PyCharm [selección de intérprete y carpeta del proyecto].



El cuadro de texto llamado *Location* le permite crear y/o elegir la *carpeta donde será alojado el proyecto* que se quiere crear. A la derecha del cuadro *Location* hay un pequeño botón con el ícono

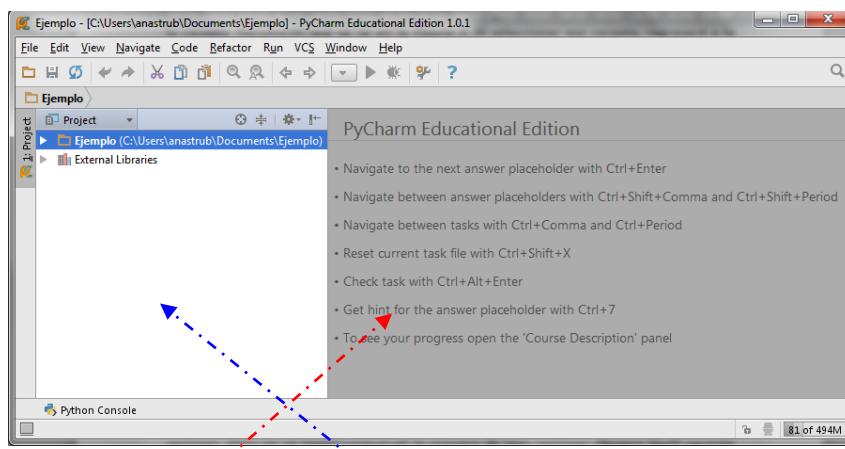
"...". Presione este botón y verá una ventana para navegar en el sistema de archivos de su disco local. Elija la carpeta donde usted querrá que se aloje el proyecto, y seleccione el ícono que se indica en la gráfica de la *Figura 10* para crear una carpeta específica para su proyecto.

**Figura 10:** Creación de una carpeta específica para el proyecto.



Supongamos entonces que hemos creado una nueva carpeta llamada *Ejemplo* dentro de la carpeta *Documents* que se ve en la *Figura 10*. Al seleccionar esa carpeta, regresará a la ventana que muestra en la *Figura 9*, y verá que el cuadro *Location* contiene ahora la ruta de esa carpeta. Presione el botón *Create* que se encuentra debajo y a la derecha de la ventana de bienvenida y se abrirá el escritorio de trabajo de *PyCharm*, mostrando un aspecto semejante al de la *Figura 11*.

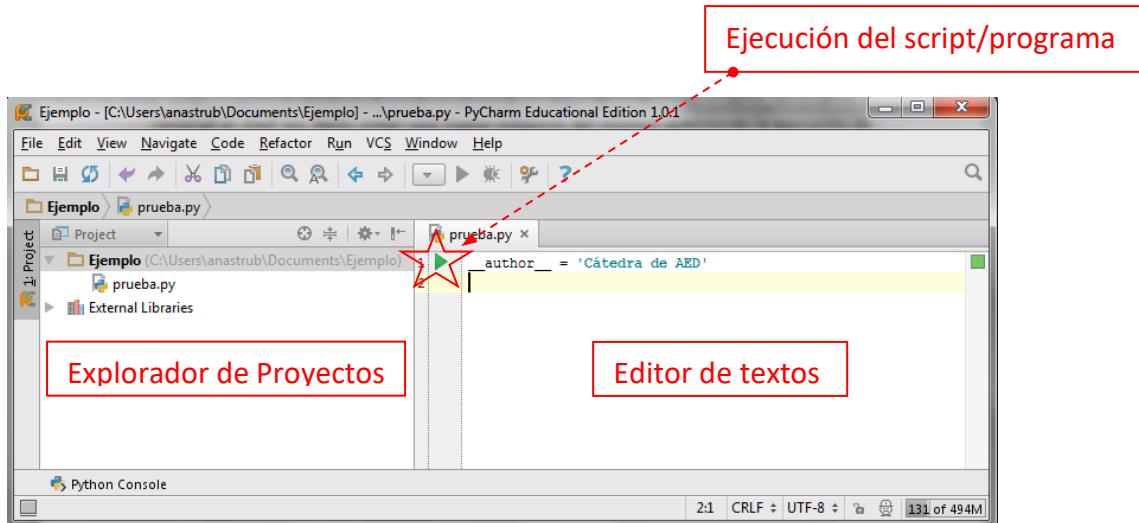
**Figura 11:** Escritorio de trabajo de PyCharm [con un proyecto vacío]



El bloque que se ve en *color gris* es la zona donde estará ubicado el editor del texto de *PyCharm* (mediante el cual podrá escribir sus programas), pero para acceder al editor debe tener al menos un programa en desarrollo dentro del proyecto. Para abrir el editor y comenzar a escribir su primer programa, apunte con el mouse a cualquier lugar del *panel blanco* (conocido como *explorador de proyectos*) que se encuentra a la izquierda de la zona gris del editor, haga *click con el botón derecho*,

y en el menú contextual que aparece seleccione la opción *File*, y luego el ítem *Python File*. Se le pedirá que escriba el nombre del archivo donde será grabado el código fuente de su programa: escriba (por ejemplo) el nombre *prueba* y presione el botón *Ok*. En ese momento, se abrirá el editor de textos (el bloque de color gris cambiará a color blanco) quedando a la espera de que el programador comience a escribir (ver *Figura 12*).

**Figura 12:** Escritorio de trabajo de PyCharm [con el editor abierto y listo para usar]



En este momento se puede comenzar a escribir el programa en el editor. Note que *PyCharm* (dependiendo de la versión que esté usando) puede colocar una primera línea comenzando con la palabra `'__author__'` en el archivo para documentar el nombre del autor del programa. Esta es una convención de trabajo del lenguaje Python: no se preocupe por ahora de esa línea (si es que aparece). A continuación escriba un script muy simple, para cargar por teclado el nombre del usuario y mostrarle un saludo. El código fuente debería quedar así (incluyendo la línea `'__author__'` que *PyCharm* coloca por default, aunque el nombre del autor que aparezca seguramente será diferente al del ejemplo):

```
__author__ = 'Cátedra de AED'

nom = input('Ingrese su nombre: ')
print('Hola', nom)
```

Las dos líneas en blanco que hemos dejado en el ejemplo entre la primera línea y la segunda línea están allí por razones de claridad (y en determinadas circunstancias se espera que figuren por convención de trabajo de Python). Puede dejarlas así o eliminarlas si lo desea ya que no afectará al correcto funcionamiento del script. Ahora puede ejecutar el script, simplemente presionando el ícono en forma de punta de color verde (▶) que se resalta en la *Figura 12*. Observará que en la parte inferior del escritorio de *PyCharm* se abre un nuevo marco (conocido como el marco de la *consola de salida* de *PyCharm*) y allí irán apareciendo los mensajes del programa. También allí deberá el usuario escribir un nombre cuando aparezca el cursor titilante. La *Figura 13* (página 50) muestra la forma en que todo se vería cuando termine de ejecutar el script, suponiendo que el nombre cargado fue *Ana*.

Un detalle muy importante en Python, es que el programador **debe** respetar el *correcto encolumnado de instrucciones* (o *indentación de instrucciones*) en su programa [3] [4], pues de otro modo el intérprete lanzará un error. La indentación o encolumnado le indican al programa qué instrucciones están en el mismo nivel o *línea de flujo de ejecución*; y en Python es **obligatorio**

respetar ese encolumnado. El script anterior funcionará sin problemas pues además de no tener errores de sintaxis, está correctamente indentado. Sin embargo, observe el siguiente ejemplo:

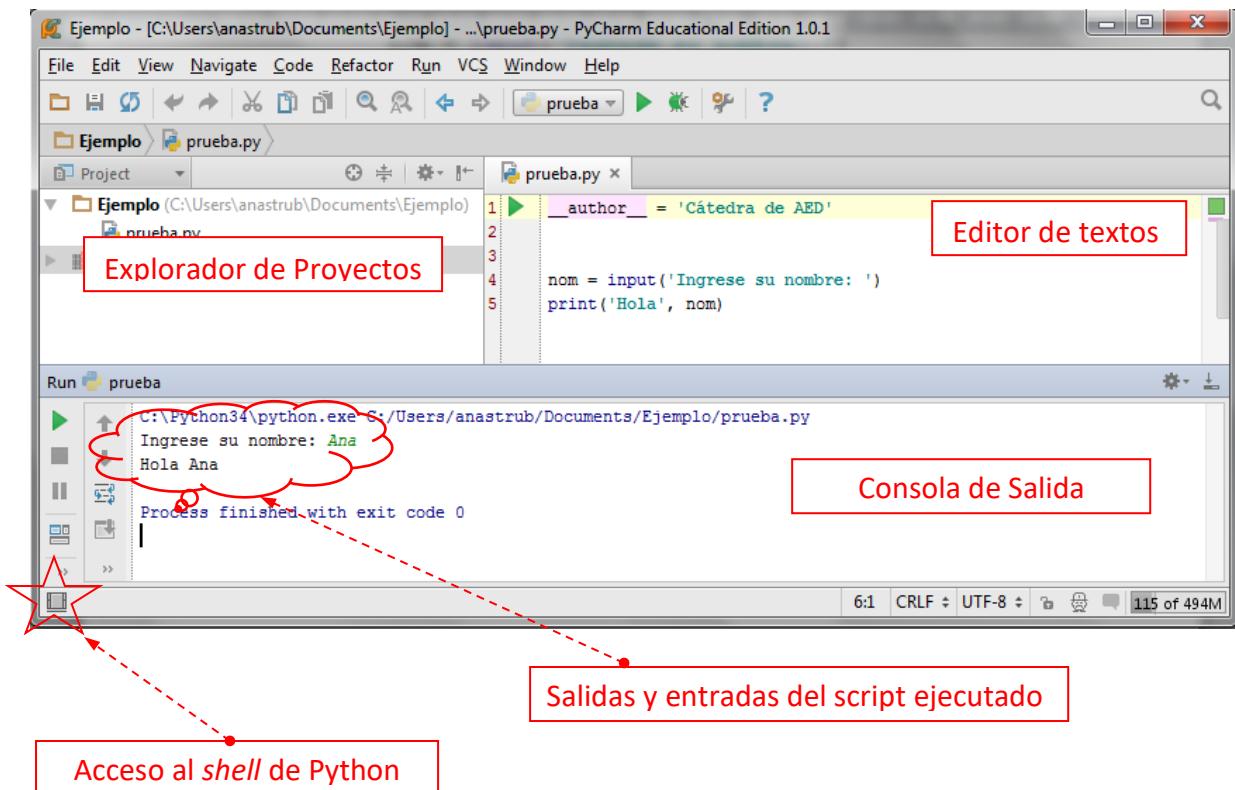
```
__author__ = 'Cátedra de AED'

nom = input('Ingrese su nombre: ')
print('Hola', nom)
```

Se trata exactamente del mismo script original, pero ahora la tercera instrucción está fuera de encolumnado (se dice que está *mal indentada*): la instrucción `print('Hola', nom)` está corrida unos tres espacios hacia la derecha, y ya no está en la misma columna que la instrucción anterior `nom = input('Ingrese su nombre: ')`. Así como está escrito, este script no podrá ejecutarse: el intérprete Python lanzará el siguiente error en la consola de salida cuando se presione el botón de ejecución:

IndentationError: unexpected indent

Figura 13: Escritorio de trabajo de PyCharm [luego de ejecutar un script].



De hecho, incluso antes de intentar ejecutar el script, el editor de textos de *PyCharm* detectará el error y lo marcará, subrayando con una cejilla de color rojo la letra p de la función `print()`. Además, le mostrará un mensaje aclaratorio cuando pase el puntero del mouse sobre esa instrucción. En general, el editor de textos de *PyCharm* dispone de esta característica de *predicción de errores*, de forma que el programador puede saber que está cometiendo uno antes de intentar ejecutar.

Para finalizar esta introducción al procedimiento de uso de la herramienta *PyCharm*, vuelva a la Figura 13 y observe el pequeño ícono ubicado en el ángulo inferior izquierdo del escritorio de trabajo de *PyCharm*. Este ícono da acceso a la función rápida de activación de marcos en *PyCharm*: cuando pase el puntero del mouse sobre él, aparecerá una pequeña caja de menú contextual, indicando los

nombres de los marcos de trabajo que están disponibles en ese momento; y seleccionando cualquiera de ellos activará el que deseé. Esto es útil cuando el marco que se quiere acceder no está visible en un momento dado y uno de esos marcos es el propio shell de comandos de Python. Si pasa el mouse sobre el ícono de activación rápida y selecciona luego el ítem Python Console, notará que el marco inferior (en donde hasta ahora se mostraban las entradas y salidas del script ejecutado) cambia y se muestra en su lugar el shell de comandos de Python (lo cual puede comprobar porque aparece el prompt ">>>" característico del ese shell). Si el shell está abierto, obviamente, puede escribir y ejecutar órdenes directas de la misma en que lo hacía con el IDLE GUI que analizamos en la Ficha 01 (de hecho, es exactamente el mismo programa, pero ahora incrustado dentro del sistema de ventanas (o interfaz de usuario) de PyCharm. Puede volver a ver la consola de salida normal, volviendo a pasar el mouse sobre el ícono de activación rápida y seleccionando ahora el ítem *Run*. Dejamos para el alumno la tarea de explorar el funcionamiento general del shell "incrustado", repitiendo ejemplos vistos en la Ficha 01 o introduciendo las instrucciones que deseé para que se convenza que se trata del mismo viejo shell de la Ficha 01.

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] V. Frittelli, D. Serrano, R. Teicher, F. Steffolani, M. Tartabini, J. Fenández and G. Bett, "Uso de Python como Lenguaje Inicial en Asignaturas de Programación", in *Libro de Artículos Presentados en la III Jornada de Enseñanza de la Ingeniería - JEIN 2013*, Bahía Blanca, 2013.
- [3] Python Software Foundation, "Python Documentation", 2021. [Online]. Available: <https://docs.python.org/3/>.
- [4] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [5] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.

# Ficha 3

## Tipos Estructurados Básicos

### 1.] Tipos estructurados (o compuestos) básicos en Python: secuencias de datos.

Hemos visto que en Python una variable puede ser asignada con un valor y luego cambiar ese valor por otro (perdiendo el valor anterior) cuando el programador lo requiera. En general, aquellos tipos de datos que sólo admiten que una variable pueda contener *un único valor de ese tipo*, se llaman *tipos simples*. Los tipos *int*, *float* y *bool* que ya conocemos, son ejemplos de tipos simples. Una variable de tipo simple, entonces, puede representarse como un recipiente con un único compartimiento para almacenar un único elemento:



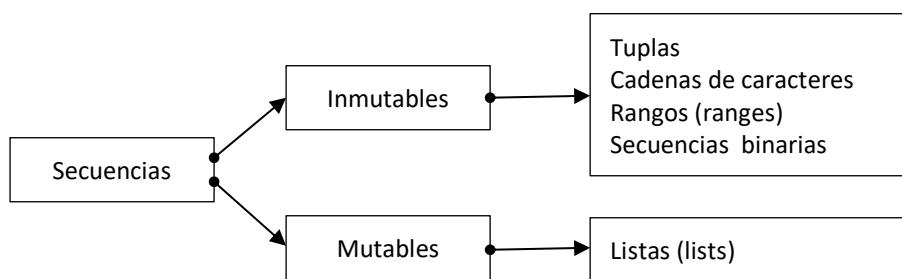
Sin embargo, tanto en Python como en todo otro lenguaje, existen tipos de datos que permiten que *una misma variable pueda contener varios valores al mismo tiempo*. Estos tipos se designan genéricamente como *tipos compuestos* o *tipos estructurados* y por esta razón, una *variable de tipo estructurado* se suele designar también como una *estructura de datos*.

Hemos visto ya un ejemplo concreto de tipo estructurado: las *cadenas de caracteres* (o *strings*), que son casos particulares de una gran familia de tipos compuestos de Python llamados *secuencias* (o *secuencias de datos*), las que a su vez pueden ser *inmutables* o pueden ser *mutables*.

Una *secuencia de datos inmutable* es simplemente un conjunto finito de datos cuyos valores y orden de aparición **no pueden modificarse una vez asignados**. Las ya citadas *cadenas de caracteres* son un ejemplo de *secuencias inmutables*.

Por otra parte, las *secuencias de datos mutables* son aquellas secuencias en las que los valores individuales **pueden modificarse** luego de ser asignados. La *Figura 1* muestra un esquema general de los tipos de secuencias disponibles en *Python 3* (aunque considere que este esquema podría ampliarse en el futuro, ya que Python continuamente incorpora mejoras y agregados en sus nuevas versiones [1]):

**Figura 1: Tipos de secuencias en Python 3.**



Hasta aquí sólo hemos visto ejemplos muy simples de uso de *cadenas de caracteres*, pero el resto de los tipos de secuencias de la figura anterior irán siendo presentadas y estudiadas a lo largo del curso.

A diferencia de las variables de tipo simple, se puede pensar que una *estructura de datos* (o sea, una *variable de tipo compuesto*) es un recipiente que contiene muchos compartimientos internos.

Así, una *cadena de caracteres* puede entenderse como un contenedor dividido en tantas casillas como caracteres tenga esa cadena. Y si el programador necesita almacenar varios números (o varios elementos de cualquier tipo que no sean necesariamente caracteres) en el mismo contenedor, puede en Python usar *tuplas* (entre otras alternativas).

Una *tupla* es una *secuencia inmutable* de datos que pueden ser de tipos diferentes. En Python hay varias formas de definir una tupla *t* que contenga ninguno, uno o varios valores iniciales [1]:

1. Usando un par de paréntesis vacíos para denotar una tupla vacía: `t = ()`
2. Usando una coma al final, para denotar una tupla de un solo valor: `t = a,` o bien `t = (a,)`
3. Separando los ítems con comas: `t = a, b, c` o bien `t = (a, b, c)`
4. Usando la función predefinida `tuple()`: `t = tuple()` o bien `t = tuple(iterable)` (donde *iterable* es cualquier secuencia que pueda recorrerse con un *ciclo for iterador*<sup>1</sup>: una cadena, otra tupla, un rango, una lista, etc. Por ejemplo, la siguiente instrucción crea una tupla *t* con cada uno de los caracteres de la cadena 'Mundo': `t = tuple('Mundo')`).

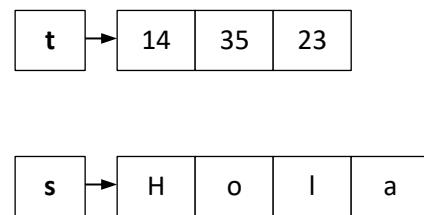
Note que el uso de paréntesis para encerrar una tupla es opcional (ver forma 3), aunque a veces es exigible (forma 1) para evitar ambigüedades.

Se dice que la tupla *t* del script que sigue tiene tres elementos o componentes (uno por cada número), y la cadena *s* tiene cuatro (uno por cada carácter de la cadena). Los esquemas gráficos que acompañan al script ilustran la idea:

Figura 2: Ejemplos gráficos de secuencias inmutables de tipo *cadena* y *tupla*:

```
# una tupla...
t = 14, 35, 23
print(t)

# una cadena de caracteres...
s = 'Hola'
print(s)
```



Cualquiera sea el tipo de secuencia que se esté usando (tupla, cadena, lista, etc.) la función interna `len()` de Python puede usarse para determinar la *cantidad de elementos que tiene esa secuencia*:

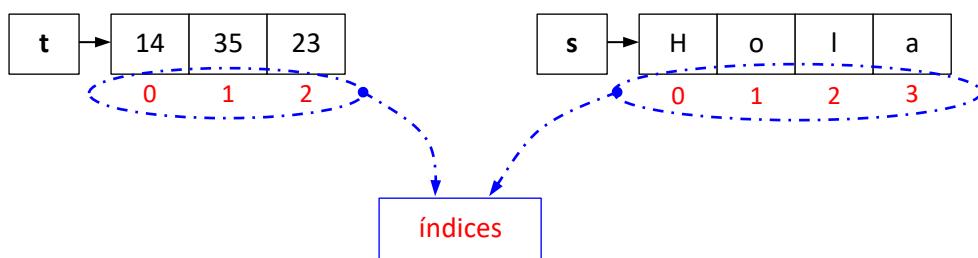
```
t = 14, 35, 23
n = len(t)
print(n)      # muestra: 3
```

<sup>1</sup> Veremos el concepto de *ciclo for iterador* en una ficha posterior, en ocasión de estudiar el funcionamiento de una *instrucción repetitiva*.

```
s = 'Hola'
m = len(s)
print(m)      # muestra: 4
```

Un detalle importante que por ahora sólo veremos a modo de presentación (y sobre el cual volveremos en profundidad más adelante en el curso) es que cada casillero de una secuencia de cualquier tipo está asociado a un número de orden llamado **índice**, mediante el cual se puede acceder a ese casillero en forma individual. En Python, la **secuencia de índices** que identifica a los casilleros de una secuencia de datos comienza desde **0(cero)** y sigue luego en forma correlativa. La *Figura 3* ayuda a entender la idea (suponga la tupla *t* y la cadena *s* del ejemplo anterior):

**Figura 3: Dos secuencias y sus casilleros identificados por **índices**.**



Para acceder a un elemento individual de una secuencia, basta con escribir *el identificador de la misma y luego agregar entre corchetes el índice del casillero a acceder*:

```
t = 14, 35, 23
print(t[1])      # muestra: 35
x = t[2]
print(x)         # muestra: 23

s = 'Hola'
print(s[0])      # muestra: H
c = s[3]
print(c)         # muestra: a
```

Cuando se trabaja con secuencias de tipo **inmutable** (ver *Figura 1*) como las tuplas o las cadenas de caracteres, **cualquier intento de modificar el valor de un casillero individual accediéndolo por su índice producirá un error de intérprete**, similar a los que se muestran aquí:

```
t = 14, 35, 23
t[2] = 87      # error...
TypeError: 'tuple' object does not support item assignment
```

```
s = 'Hola'
s[2] = 'a'      # error...
TypeError: 'str' object does not support item assignment
```

La modificación del valor de un elemento de una secuencia está permitida siempre y cuando esa secuencia sea del tipo *mutable*, como es el caso de las *listas* en Python que veremos oportunamente [2].

Es perfectamente válido que una *tupla* contenga elementos de tipos diferentes: cada casillero se define en el momento que se asigna, y dado que Python es de tipado dinámico, nada impide que esos casilleros sean de diferentes tipos:

```
t2 = 'Juan', 24, 3400.57, 'Córdoba'
print(t2)    # muestra: ('Juan', 24, 3400.57, 'Córdoba')
```

Si se desea procesar por separado cada elemento, se usan los índices como se indicó y se pueden introducir variables temporales a criterio y necesidad del programador:

```
nombre = t2[0]
edad = t2[1]
print('Nombre:', nombre, '- Edad:', edad, '- Sueldo:', t2[2], '- Ciudad:', t2[3])
# muestra: Nombre: Juan - Edad: 24 - Sueldo: 3400.57 - Ciudad: Córdoba
```

Habiendo introducido el uso de *tuplas* es muy interesante observar que en Python una *expresión de asignación* puede estar formada por *tuplas en ambos lados de la expresión*: una secuencia de variables (separadas por comas) en el lado izquierdo del operador de asignación, y otra secuencia con la misma de cantidad de valores, variables y/o expresiones del lado derecho del operador de asignación, también separados por comas. En el siguiente ejemplo sencillo, las variables *x* e *y* forman una tupla en el lado izquierdo, y toman respectivamente los valores 10 y 20 (que surgen de las dos expresiones que forman otra tupla en el lado derecho de la asignación):

```
a = 5
x, y = 2*a, 4*a
```

En la expresión de asignación el miembro derecho es evaluado primero, antes que el miembro izquierdo, y la evaluación procede de izquierda a derecha. Los valores obtenidos de la *tupla* derecha son provisoriamente asignados en variables temporales y luego esos valores son asignados a las variables de la *tupla* del lado izquierdo, también en orden de izquierda a derecha (por este motivo el 10 se asignó en *x* y no en *y*: el primer valor del lado derecho, se asigna en la primera variable del lado izquierdo, y así sucesivamente). La gráfica que se muestra en la *Figura 4* (página 56) muestra la forma en que todo ocurre.

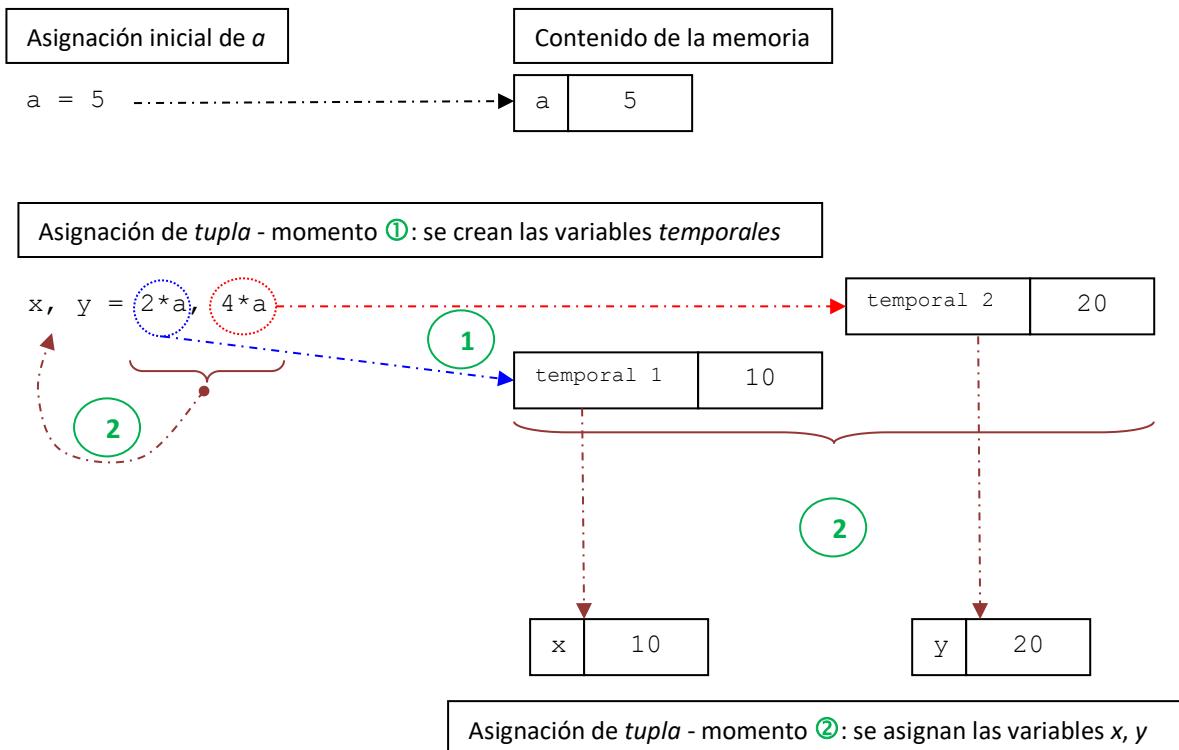
En una expresión de asignación de múltiples operandos (como la de una *tupla*), el operador *coma* (,) actúa como un *empaquetador* cuando está a la derecha de la asignación, produciendo la *tupla* (o *secuencia de expresiones*) con los valores obtenidos. Y actúa como un *desempaquetador* si está a la izquierda de la expresión [1]. La siguiente instrucción es válida en Python, y crea una *tupla* de dos números (5 y 8) almacenada como tal en la variable *sec*:

```
sec = 5, 8
print('Secuencia: ', sec)
# muestra: Secuencia: (5, 8)
```

Y como era de esperarse, la instrucción siguiente asigna los valores 5 y 8 en las variables *a* y *b*:

```
a, b = sec
print('a: ', a)
print('b: ', b)
```

Figura 4: Asignación de una tupla en Python.



Lo anterior tiene al menos una aplicación práctica inmediata: es muy común que el programador tenga que realizar lo que se conoce como un *intercambio de valores entre dos variables*: se tienen dos variables `a` y `b`, cada una con un valor previo, y se necesita intercambiar esos valores, de forma que `a` tome el valor de `b`, y que `b` tome el valor original de `a`. Según sabemos, sería un error intentar simplemente lo siguiente:

```

a = 12
b = 4
a = b
b = a
print('Valor final de a: ', a)
print('Valor final de b: ', b)

```

ya que la secuencia mostrada hará que ambas variables queden valiendo finalmente el valor 4, perdiendo el 12. Asegúrese de entender la veracidad de esta afirmación, haciendo un gráfico de los valores de cada variable en cada momento del esquema.

¿Cómo hacer entonces el intercambio? En general, en la mayoría de los lenguajes de programación, el intercambio de valores de dos variables requiere el uso de una tercera variable a modo de auxiliar, y usar (obviamente...) ingenio y sentido común. La siguiente secuencia de instrucciones en Python produce correctamente el intercambio de valores entre `a` y `b`, usando una tercera variable `c`, en la forma en que normalmente se hace en todo lenguaje:

```

# valores iniciales...
a = 12
b = 4
# secuencia de intercambio al estilo clásico...
c = a
a = b
b = c

```

```
# visualización de valores intercambiados...
print('Valor final de a: ', a)
print('Valor final de b: ', b)
```

Aun cuando lo anterior funciona, el hecho es que en Python se puede hacer el intercambio de valores recurriendo a la asignación de *tuplas* en forma directa. La siguiente instrucción en Python, hace el intercambio de valores entre las variables *a* y *b* sin tener que recurrir en forma explícita a una variable auxiliar (ya que en realidad, *la variable auxiliar es la tupla que Python crea al evaluar el miembro derecho*) (Asegúrese de comprender la secuencia que sigue, haciendo una gráfica del estado de las variables similar al de la *Figura 4* en cada momento):

```
# valores iniciales...
a = 12
b = 4
# intercambio al estilo Python...
a, b = b, a
# visualización de valores intercambiados...
print('Valor final de a: ', a)
print('Valor final de b: ', b)
```

## 2.] Cadenas de caracteres: Elementos adicionales.

Las *cadenas de caracteres* o *strings* constituyen un tipo de dato fundamental en todo lenguaje de programación. Como sabemos, en Python un *string* se define como una *secuencia inmutable de caracteres* y por lo tanto, le son aplicables todos los operadores, funciones y métodos propios del manejo de secuencias [1].

Un *literal* (o sea, una *constante*) de tipo *string* es una secuencia de caracteres delimitada por comillas dobles o por comillas simples (o *apóstrofos*). Hemos visto que se pueden usar indistintamente las comillas dobles o las comillas simples, pero si se comenzó con un tipo, se debe cerrar con el mismo tipo de comilla:

```
nombre = "Pedro"
provincia = 'Córdoba'
```

Lo anterior es útil, por ejemplo, si se quiere almacenar *literalmente* una comilla doble o simple como parte de una cadena constante: si el delimitador usado fue comilla doble, entonces la comilla simple puede usarse dentro de la cadena como carácter directo, y viceversa:

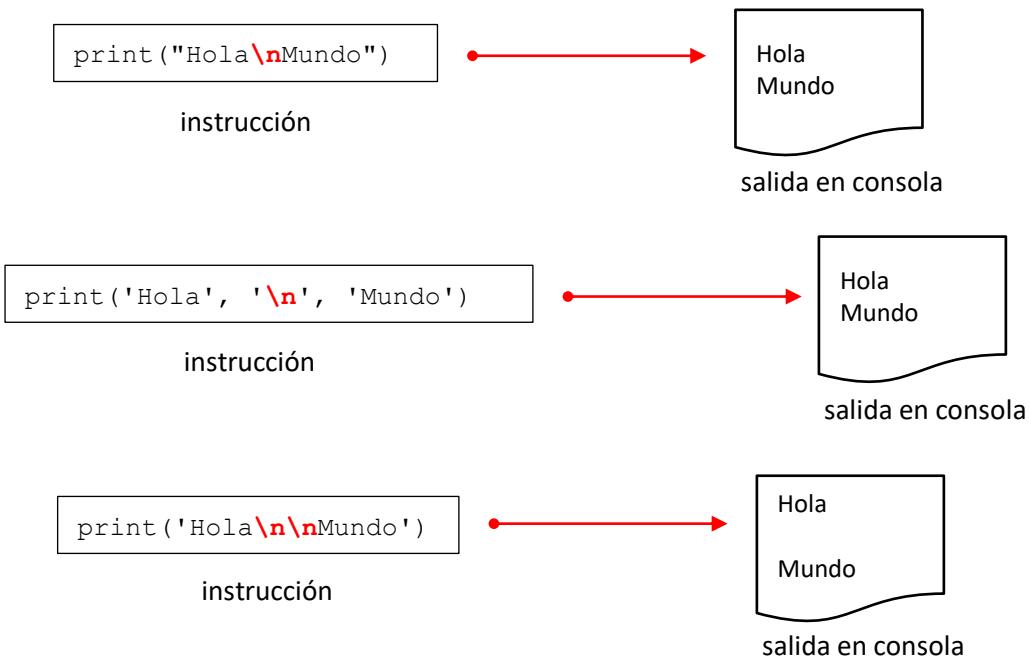
```
apellido = "O'Donnell"
autor = 'Anthony "Tony" Hoare'
```

Tanto en Python como en otros lenguajes (Java, C, C++, por ejemplo) se pueden usar caracteres especiales llamados *caracteres de escape* o *caracteres de control*. Un carácter de escape se forma con un símbolo de barra (\) seguido de algún carácter especial que actúa en forma de código: cuando Python encuentra una secuencia que corresponde a un carácter de escape dentro de un literal de tipo *string*, interpreta a la secuencia \caracter como un solo y único carácter.

En algunos casos, el carácter de escape provoca algún efecto en la salida por consola (y no exactamente la visualización del carácter). Por ejemplo, el carácter "\n" produce el efecto de

un salto de línea en la salida por consola cuando aparece dentro de una cadena (vea los ejemplos en la gráfica siguiente):

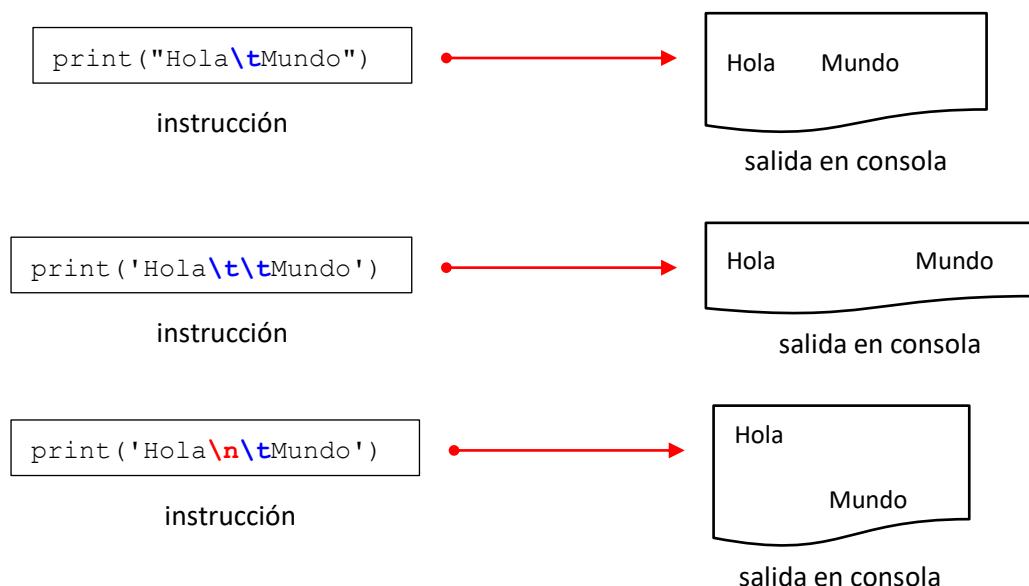
**Figura 5: Efectos de uso del carácter de salto de línea.**



Como se puede ver en el tercer ejemplo de la *Figura 5*, los caracteres de control pueden combinarse: dos `\n` seguidos, producen dos saltos de línea en la consola de salida.

Por otra parte, el carácter de control `\t` provoca un espaciado en la consola de salida, equivalente a una "sangría" (o "espaciado horizontal" o "Tab") (normalmente, 4 espacios en la misma línea, a partir del cursor). Y todos pueden combinarse, repitiendo varios del mismo tipo o agrupando dos o más de tipos diferentes (ver *Figura 6*):

**Figura 6: Efectos de uso del carácter de espaciado horizontal.**



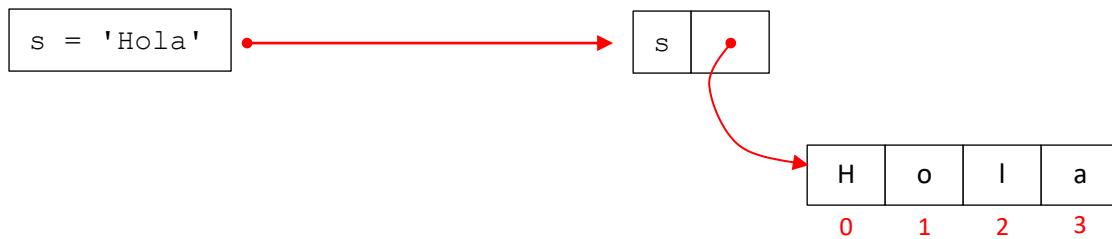
Los caracteres de escape `\'` y `\\"` representan *literalmente* comillas simples o comillas dobles y permiten entonces su uso dentro de cadenas en forma directa, sin importar si estas cadenas fueron abiertas con el mismo tipo de delimitador [1]:

```
apellido = 'O\'Donnell' # almacena: O'Donell
autor = "Anthony \\"Tony\\" Hoare" # almacena: Anthony "Tony" Hoare
```

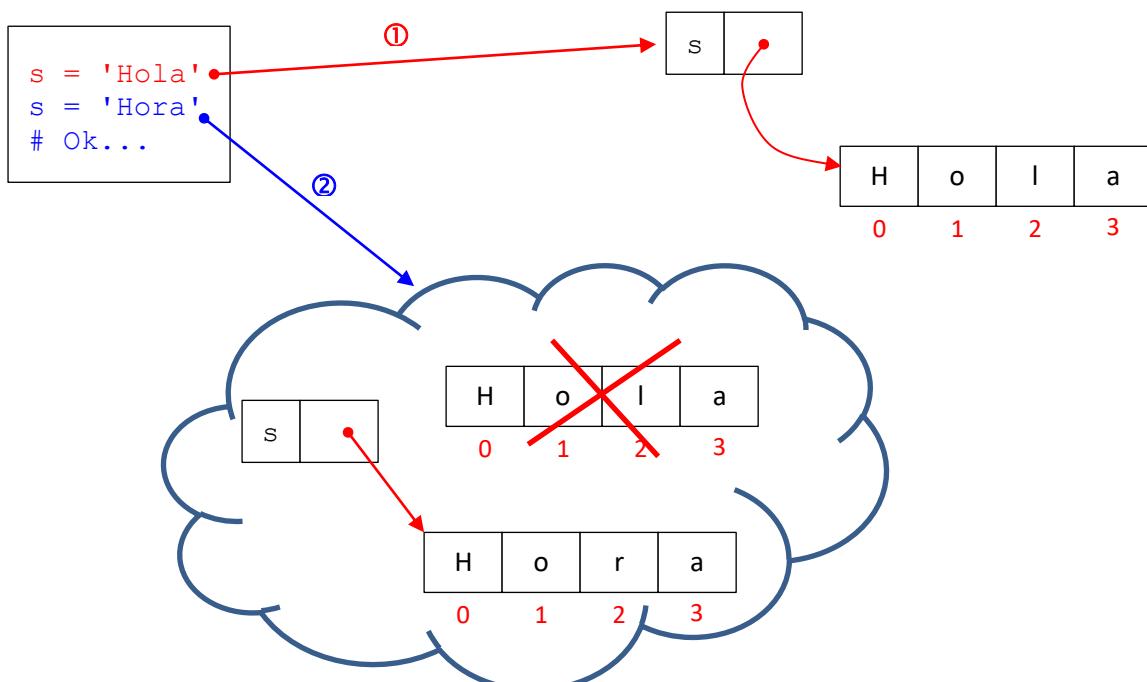
También sabemos que los caracteres individuales de una cadena se almacenan como una secuencia con índices numerados desde cero en adelante, y que *el índice puede usarse para acceder a esos caracteres, sin modificarlos* (recuerde que una cadena es una secuencia *inmutable*):

```
s = 'Hola'
print('Primeras dos letras: ', s[0], s[1])
s[2] = 'r' # error...
```

Una variable de tipo cadena (y cualquier variable en general) contiene en realidad la *dirección del byte de memoria donde la cadena comienza* (y se dice entonces que la variable *apunta* a esa dirección o a esa cadena):



Debe quedar claro que lo que **no** puede hacerse es *cambiar el valor de un casillero de una cadena ya creada y almacenada en memoria*. En el ejemplo anterior, el intento de cambiar la "l" en la casilla `s[2]` por una "r" (o sea, `s[2] = "r"`) provoca un error de intérprete. *Pero no hay ningún inconveniente en que una variable que apunta a una cadena, pase a apuntar a otra diferente* (en este caso, se crea la nueva cadena y la anterior se pierde):



Para finalizar esta sección, resaltamos un hecho interesante: los operadores *suma (+)* y *producto (\*)* en Python también pueden usarse con cadenas de caracteres. El primero permite lo que se conoce como *la concatenación o unión de dos o más cadenas*: se crea una *nueva cadena* que contiene a todas las cadenas originales, en el mismo orden en que se unieron y una detrás de la otra [1]:

```
nombre = 'Guido'
apellido = "van Rossum"
completo = nombre + " " + apellido
print("Nombre completo: ", completo)
# muestra: Guido van Rossum
```

Y el segundo permite *repetir* una cadena varias veces concatenando el resultado:

```
replicado = nombre * 4
print("Nombre repetido: ", replicado)
# muestra: GuidoGuidoGuidoGuido
```

### 3.] Funciones comunes de la librería estándar de Python.

Los operadores aritméticos de Python que hasta aquí hemos visto permiten el planteo directo de muchos cálculos y fórmulas comunes y generales. No obstante, y en forma esperable, Python provee un amplio conjunto de *funciones internas*, listas para usar, que están siempre disponibles en un script o programa sin necesidad de declaraciones ni instrucciones adicionales. La lista completa de estas funciones puede verse en la documentación de ayuda de Python [3] que se instala junto con el lenguaje, pero aquí extraemos algunas de las más comunes, para facilitar el avance:

**Figura 7: Tabla de las principales funciones de la librería estándar de Python.**

Función	Descripción	Ejemplo de uso
<b>abs(x)</b>	Retorna el valor absoluto del parámetro <b>x</b> .	x = -23 y = abs(x) print(y) # muestra: 23
<b>bin(x)</b>	Obtiene la conversión a binario (base 2) del valor <b>x</b> , en formato de cadena cuyos dos primeros caracteres son '0b', indicando que lo que sigue es una secuencia en binario.	x = 8 s = bin(x) print(s) # muestra: 0b1000
<b>chr(i)</b>	Retorna el carácter (en formato de string) que corresponde al valor Unicode número <b>i</b> .	i = 65 c = chr(i) print(c) # muestra: A
<b>divmod(a, b)</b>	Retorna el cociente y el resto de la división entera entre los parámetros <b>a</b> y <b>b</b> .	a, b = 14, 4 c, r = divmod(a,b) print(c, r) # muestra: 3 2
<b>float(x)</b>	Convierte la cadena <b>s</b> a un número en coma flotante. Hemos usado esta función combinada con la función <i>input()</i> para cargar números flotantes desde teclado.	s = '23.45' y = float(s) print(y) # muestra: 23.45
<b>hex(x)</b>	Obtiene la conversión a hexadecimal (base 16) del valor <b>x</b> , en formato de cadena cuyos dos primeros caracteres son '0x', indicando que lo que sigue es una secuencia en hexadecimal.	x = 124 s = hex(x) print(s) # muestra: 0x7c
<b>input(p)</b>	Obtiene una cadena de caracteres desde la entrada estándar y la retorna. La cadena <b>p</b> es visualizada a modo de "prompt" mientras se espera la carga.	Ver los ejemplos de uso de carga por teclado en las fichas 1 y 2 del curso.
<b>int(x)</b>	Convierte la cadena <b>s</b> a un número entero. Hemos usado esta función combinada con la función <i>input()</i> para cargar	s = '1362' y = int(s)

	números enteros desde teclado.	<code>print(y)</code> # muestra: 1362
<b>len(s)</b>	Retorna la longitud del objeto <b>t</b> tomado como parámetro (un string, una lista, una tupla o un diccionario). La longitud es la <i>cantidad de elementos</i> que tiene el objeto <b>t</b> .	<code>t = 12, 45, 73</code> <code>n = len(t)</code> <code>print(n)</code> # muestra: 3
<b>max(a, b, *args)</b>	Retorna el mayor de los parámetros tomados. Notar que admite una <i>cantidad arbitraria</i> de parámetros: <b>max(a,b)</b> retorna el mayor entre <b>a</b> y <b>b</b> , pero <b>max(a,b,c,d)</b> retorna el mayor entre <b>a, b, c y d</b> .	<code>a, b = 6, 3</code> <code>my = max(a, b)</code> <code>print(my)</code> # muestra: 6
<b>min(a, b, *args)</b>	Retorna el menor de los parámetros tomados. Notar que admite una <i>cantidad arbitraria</i> de parámetros: <b>min(a,b)</b> retorna el menor entre <b>a</b> y <b>b</b> , pero <b>min(a,b,c,d)</b> retorna el menor entre <b>a, b, c y d</b> .	<code>a, b = 6, 3</code> <code>mn = min(a, b)</code> <code>print(mn)</code> # muestra: 3
<b>oct(x)</b>	Obtiene la conversión a octal (base 8) del valor <b>x</b> , en formato de cadena cuyos dos primeros caracteres son '0o', indicando que lo que sigue es una secuencia en octal.	<code>x = 124</code> <code>s = oct(x)</code> <code>print(s)</code> # muestra: 0o174
<b>ord(c)</b>	Retorna el entero Unicode que representa al carácter <b>c</b> tomado como parámetro.	<code>c = 'A'</code> <code>i = ord(c)</code> <code>print(i)</code> # muestra: 65
<b>pow(x, y)</b>	Retorna el valor de <b>x</b> elevado a la <b>y</b> . Note que lo mismo puede hacerse con el operador "doble asterisco" (**). En Python: <b>r = pow(x, y)</b> es lo mismo que <b>r = x**y</b> .	<code>x, y = 2, 3</code> <code>r = pow(x, y)</code> <code>print(r)</code> # muestra: 8
<b>print(p)</b>	Muestra el valor <b>p</b> en la consola estándar.	Ver los ejemplos de uso de salida por consola estándar en las fichas 1 y 2 del curso.
<b>round(x, n)</b>	Retorna el número flotante <b>x</b> , pero redondeado a <b>n</b> dígitos a la derecha del punto decimal. Si <b>n</b> se omite retorna la parte entera de <b>x</b> (como int). Si <b>n</b> es cero, retorna la parte entera de <b>x</b> (pero como float): <b>round(3.1415)</b> retornará 3 (int), pero <b>round(3.1415, 0)</b> retornará 3.0 (float).	<code>x = 4.1485</code> <code>r = round(x, 2)</code> <code>print(r)</code> # muestra: 4.15
<b>str(x)</b>	Retorna una versión en forma de cadena de caracteres del objeto <b>t</b> (es decir, retorna la <i>conversión a string</i> de <b>t</b> ).	<code>t = 2, 4, 1</code> <code>s = str(t)</code> <code>print(s)</code> # muestra: (2, 4, 1)

#### 4.] Aplicaciones y casos de análisis.

Toda esta sección está dedicada al estudio y resolución de problemas simples basados sólo en estructuras secuenciales de instrucciones. Cada problema será planteado en la forma que hemos sugerido hasta ahora: comprensión general del enunciado, planteo de un algoritmo mediante un diagrama de flujo y mediante un pseudocódigo, y finalmente el desarrollo del programa en Python. La numeración de cada problema, sigue en forma correlativa a los ya planteados en la Ficha 2 y en lo que va de esta Ficha 3.

**Problema 5.)** La fuerza de atracción entre dos masas **m1** y **m2** separadas por una distancia **d**, está dada por la siguiente fórmula<sup>2</sup> de la mecánica clásica:

$$F = G * \left( \frac{m1 * m2}{d^2} \right)$$

con  $G = 6.673 * 10^{-8}$  (constante de gravitación universal)

<sup>2</sup> Este problema fue sugerido por la Ing. Marina Cardenas, docente de esta Cátedra, quien también desarrolló el script en Python para resolverlo.

*Escribir un programa que cargue las masas  $m_1$  y  $m_2$  de dos cuerpos y la distancia  $d$  entre ellos y obtenga y muestre el valor de la intensidad de la fuerza de atracción entre esos cuerpos.*

a.) Identificación de componentes:

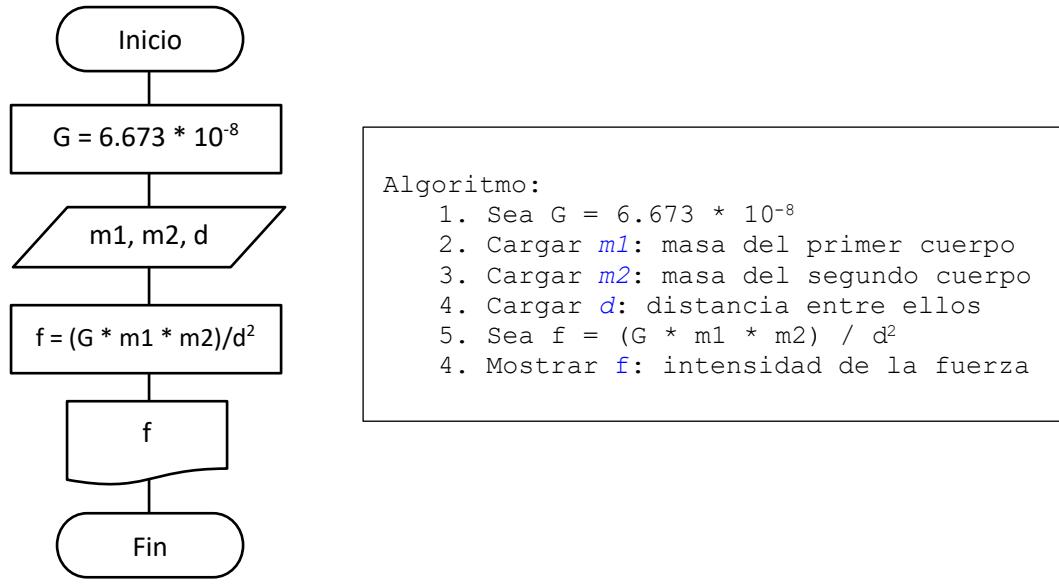
- **Resultados:** Intensidad de la fuerza de atracción entre dos cuerpos<sup>3</sup>.  
( $f$ : número real)
- **Datos:** Masa de los dos cuerpos. ( $m_1, m_2$ : números reales)  
Distancia entre ellos. ( $d$ : número real)
- **Procesos:** Problema de *física*. El enunciado mismo del problema ya brinda la fórmula necesaria para el cálculo, la que sólo debe plantearse adecuadamente para ser ejecutada en un programa. El valor de la constante  $G$  viene dado, y por lo tanto no debe cargarse por teclado: simplemente se asigna el valor en una variable auxiliar  $G$  y se usa sin más. Esa asignación y la fórmula replanteada (con sintaxis de Python) sería:

$$\begin{aligned} G &= 6.673 * \text{pow}(10, -8) \\ F &= (G * m_1 * m_2) / d ** 2 \end{aligned}$$

Por razones de sencillez, se supone aquí que los datos serán ingresados de forma que las unidades de cada magnitud sean las correctas (por ejemplo, las masas en kilogramos y la distancia en metros) y por lo tanto el programador no deberá incluir fórmulas de conversión. También suponemos que las masas y la distancia serán cargadas como positivas, pero como siempre hemos indicado, en una situación real el programador debería incluir algún tipo de validación al cargar esos datos y para evitar que se ingresen negativos o cero.

b.) Planteo del algoritmo: Mostramos tanto el *diagrama de flujo* como el *pseudocódigo* a continuación:

Figura 8: Diagrama de flujo y pseudocódigo del problema de cálculo de la fuerza de atracción.



<sup>3</sup> Respecto de la gravedad, la forma en que decrece a medida que nos alejamos de la Tierra, y los problemas que causa su ausencia en un viaje al espacio, podemos sugerir la muy taquillera película *Gravity* (*Gravedad*) del año 2013, dirigida por *Alfonso Cuarón* y protagonizada por *Sandra Bullock* y *George Clooney*. Un poco más vieja, pero muy vigente, es la famosísima *Apollo 13* de 1995, dirigida por *Ron Howard* y protagonizada por *Tom Hanks*.

c.) **Desarrollo del programa:** En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'

# asignación de constantes...
G = 6.673 * pow(10, -8)

# título general y carga de datos...
print('Ejemplo 4 - Cálculo de la fuerza de atracción')
m1 = float(input('Masa del primer cuerpo: '))
m2 = float(input('Masa del segundo cuerpo: '))
d = float(input('Distancia entre ambos: '))

# procesos...
f = (G * m1 * m2) / d**2

# visualización de resultados...
print('Fuerza de atracción:', f)
```

En este programa aparecen cinco variables, y en cuatro de ellas ( $m1$ ,  $m2$ ,  $d$  y  $f$ ) hemos usado nombres en minúscula, mientras que la restante ( $G$ ) tiene su nombre en mayúscula. De acuerdo al lenguaje que se use, existen muchas convenciones en cuanto al estilo de designación de nombres de variables. En Python no hay una recomendación especial, y sólo se pide que sea cual sea la que siga el programador sea claramente distinguible. En algunos lenguajes se recomienda que todo nombre de variable comience con minúscula. En otros, que comience con mayúscula. En otros, que se usen siempre minúsculas... Y así por el estilo.

En el anexo de *Temas Avanzados* (página 67) de esta Ficha 3 (con la que se cierra la Unidad 1 del programa de AED), hemos incluido una descripción más profunda del tema de los estilos de desarrollo de código fuente. Por ahora, y sólo para cerrar el análisis del programa del cálculo de la intensidad de la fuerza de atracción, digamos que en el desarrollo de las Fichas de Estudio de este curso, y en los ejemplos de ejercicios y programas resueltos que se entreguen a los estudiantes, aplicaremos las siguientes convenciones en cuanto a nombres de variables<sup>4</sup>:

- El nombre una variable se escribirá siempre en minúsculas (Ejemplos: *sueldo*, *nombre*, *lado*).
- Cuando se quiera que el nombre de una variable agrupe dos o más palabras, usaremos el *guión bajo* como separador (Ejemplos: *mayor\_edad*, *horas\_extra*).
- Excepcionalmente designaremos en *mayúsculas sostenidas* a alguna variable que represente algún valor muy conocido y normalmente escrito en mayúsculas (Ejemplos: *G* (por la constante de gravedad universal que vimos en el programa anterior), o *PI* (por el número *pi*), o *IVA* (por nuestro bendito impuesto al valor agregado)).

Veamos otro ejemplo de problema simple:

**Problema 6.)** *La dirección de la carrera de Ingeniería en Sistemas de la UTN Córdoba necesita un programa que permita cargar el nombre de un estudiante de quinto año, el nombre del profesor responsable de la Práctica Profesional Supervisada de ese estudiante, y el promedio general (con decimales) del estudiante en su carrera. Una vez cargados los datos, se pide simplemente mostrarlos en la consola de salida a modo de verificación, pero de forma que el nombre del practicante vaya precedido de la cadena "est." y el nombre del profesor supervisor se preceda con "prof.". El promedio del alumno debe mostrarse redondeado, sin decimales, en formato entero.*

---

<sup>4</sup> No obstante, los estudiantes (e incluso sus profesores) podrán usar otras convenciones reconocidas si las prefieren. Sólo se sugiere que se mantengan consistentes con ellas.

**a.) Identificación de componentes:**

- **Resultados:** "est. " + Nombre del practicante. (*nom\_est\_final*: string)  
"prof. " + Nombre del supervisor. (*nom\_pro\_final*: string)  
Promedio entero redondeado del practicante. (*prom\_final*: int)
- **Datos:** Nombre del practicante. (*nom\_est\_orig*: string)  
Nombre del supervisor. (*nom\_pro\_orig*: string)  
Promedio real del practicante. (*prom\_orig*: número real)
- **Procesos:** Problema de *procesamiento de cadenas y redondeo de números, en un contexto de gestión académica*. Esencialmente, se trata de mostrar los mismos datos que se cargaron, pero modificándolos ligeramente para obtener el formato de salida pedido.

En principio, agregar el prefijo "est. " al nombre del estudiante es una simple operación de *concatenación* o *unión* de cadenas, cuyo resultado podría volver a almacenarse en la misma variable donde viene el nombre original del estudiante:

```
nom_est_orig = 'est. ' + nom_est_orig
```

Sin embargo, esto haría que el dato original se pierda. En este problema en particular no parece un inconveniente mayor ya que el enunciado no pide volver a usar ese dato en un proceso posterior. Pero en general será recomendable que el programador preserve los datos originales, de forma de poder volver a usarlos más adelante si los necesitase. Por lo tanto, se deben almacenar los resultados de las concatenaciones en dos variables nuevas (una para el practicante y otra para el profesor supervisor):

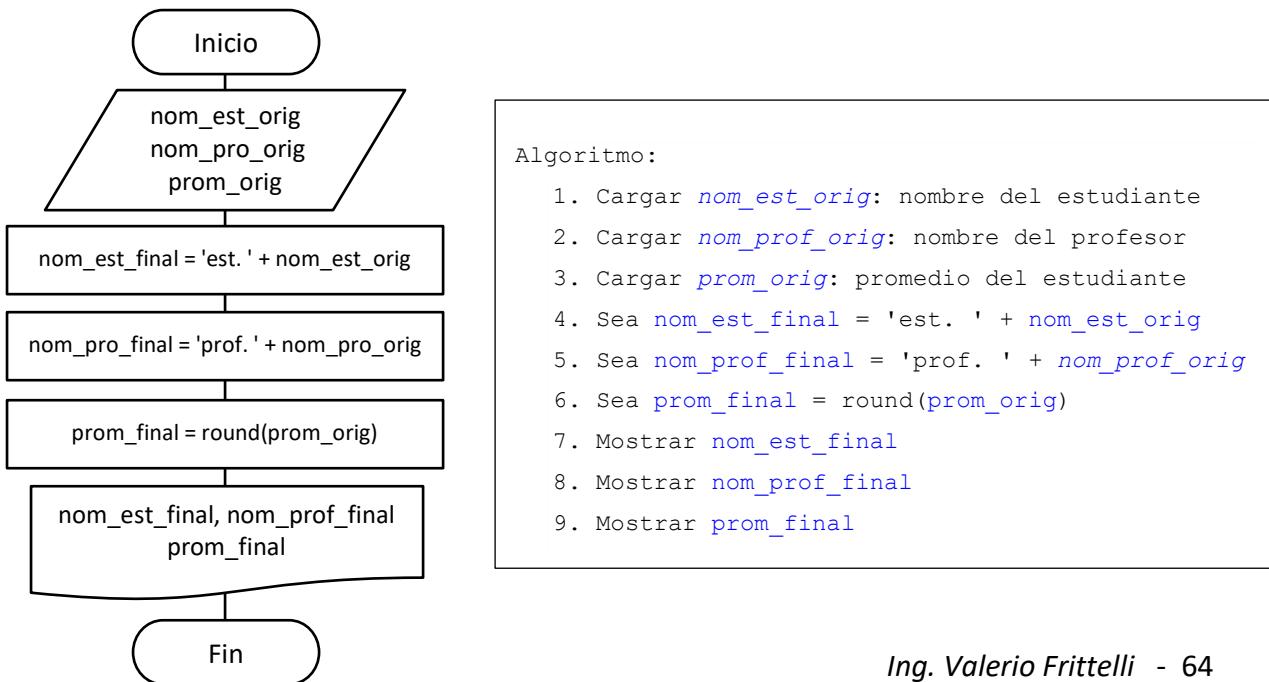
```
nom_est_final = 'est. ' + nom_est_orig  
nom_pro_final = 'prof. ' + nom_pro_orig
```

Y en cuanto al promedio, el dato viene como número real y se está pidiendo mostrarlo como entero redondeado, lo cual puede hacerse en forma directa con la función *round()* provista por Python y que presentamos en la Ficha 2:

```
prom_final = round(prom_orig)
```

**b.) Planteo del algoritmo:** Mostramos tanto el *diagrama de flujo* como el *pseudocódigo* a continuación:

Figura 9: Diagrama de flujo y pseudocódigo del problema de cálculo de la fuerza de atracción.



c.) **Desarrollo del programa:** En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'

# título general y carga de datos...
print('Ejemplo 6 - Datos de Práctica Profesional Supervisada')
nom_est_orig = input('Nombre del estudiante: ')
prom_orig = float(input('Promedio general: '))
nom_pro_orig = input('Nombre del profesor: ')

# procesos...
nom_est_final = 'est. ' + nom_est_orig
nom_pro_final = 'prof. ' + nom_pro_orig
prom_final = round(prom_orig)

# visualización de resultados...
print('Practicante:', nom_est_final, '\t-\tPromedio:', prom_final)
print('Supervisor:', nom_pro_final)
```

Todas las variables de este script han sido designadas con nombres que siguen el estilo indicado al final del problema anterior: todo el nombre en letras minúsculas y el uso del guión bajo como separador.

El siguiente problema se sugiere para aplicar propiedades del resto de la división:

**Problema 7.) Una pequeña oficina de correos dispone de cinco casillas o boxes para guardar las cartas que debe despachar. Cada casilla (que puede contener muchas cartas) está numerada con números correlativos del 0 al 4. Cada carta que se recibe tiene un código postal numérico, y en base a ese código el despachante debe determinar en qué box guardarán la carta, pero de tal forma que el mismo sistema sirva luego para saber en qué caja buscar una carta, dado su código postal. Diseñe un esquema de distribución que permita cumplir este requerimiento, cargando por teclado el código postal de tres cartas y mostrando en qué casilla será almacenada cada una.**

a.) **Identificación de componentes:**

- **Resultados:** Números de las casillas donde serán almacenadas las tres cartas recibidas. ( $n_1, n_2, n_3$ : números enteros)
- **Datos:** Códigos postales de tres cartas ( $c_1, c_2, c_3$ : números enteros)
- **Procesos:** Problema de *procesamiento y aplicación de relaciones de congruencia, en un contexto de correo postal*. Está claro que el enunciado propone un nuevo desafío: diseñar un esquema de distribución de cartas, pero sin dar muchas pistas respecto de cómo hacerlo. No hay fórmulas, ni pasos evidentes indicados en ese enunciado. Sólo hay un requerimiento. Y este tipo de situaciones son las que más típicamente enfrentan los programadores en su trabajo: entienda que las personas que soliciten sus servicios no saben nada de programación ni de algoritmos... sólo pueden decirle qué datos le darán y qué tipo de resultados esperan obtener... y el resto es tarea del profesional de la programación.

Veamos: El problema es tratar de asignar a cada código postal posible un número de casillero, sabiendo que estos últimos están todos en el rango del intervalo entero  $[0, 4]$ . Una primera idea (errónea...) podría ser tomar el último dígito del código postal y usar ese dígito como número de casillero. Pero puede verse fácilmente que esta estrategia no funciona: si el código postal termina con

cualquier dígito mayor a 4 (como 5347, o 2238), no habrá un casillero identificado con ese dígito y el sistema de distribución fallará<sup>5</sup>.

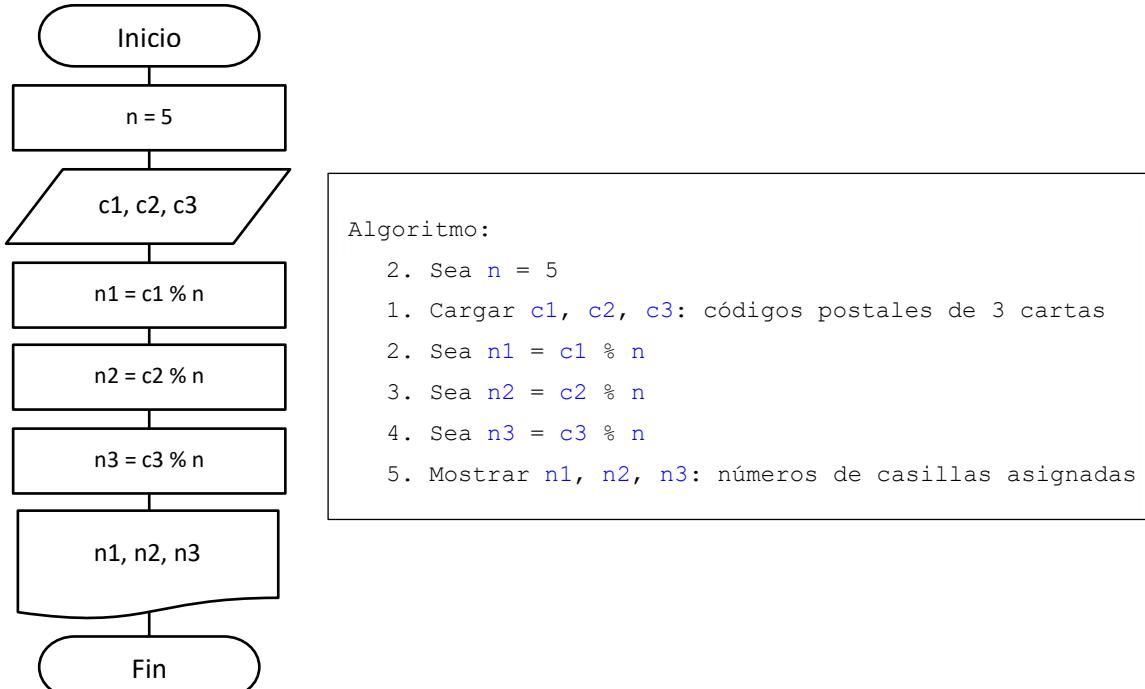
Una idea mejor surge de observar que el rango de posibles números de casilleros es exactamente igual al conjunto de posibles restos de la división por 5. Entonces, si para cada código postal simplemente se toma el resto de dividir por 5, se obtendrá inexorablemente un número entre 0 y 4 que puede ser asignado como número de casillero para la carta en cuestión. Así, la carta con código postal 5347 será asignada al casillero 2 (ya que  $5347 \% 5 = 2$ ) y también irá al casillero 2 una carta cuyo código postal sea, por ejemplo, 3672 (ya que  $3672 \% 5 = 2$ ).

En definitiva, y matemáticamente hablando [3], cada casillero puede ser equiparado a  $\mathbf{Z5}_n$ : la clase de congruencia (módulo 5) con el subíndice  $n$  igual al resto de dividir por 5, como vimos en la Ficha 2. Los números 5347 y 3672 pertenecen a  $\mathbf{Z5}_2$  y por ello las cartas con esos códigos postales van al box 2. Los números 8761 y 9236 pertenecen a  $\mathbf{Z5}_1$  y por eso las cartas con esos códigos irán al box 1.

Conclusión: nuestro algoritmo sólo debe tomar el resto de dividir por  $n = 5$  a cada código postal y mostrar ese resto. El número 5 (la cantidad de casilleros) puede asignarse previamente como una constante inicial  $n$ , lo cual le da al código fuente algo de flexibilidad si luego el programador debe modificarlo: si el número de casillas pasa de 5 a 13, por ejemplo, el programador sólo deberá cambiar el valor de  $n$  y el resto del programa seguirá funcionando correctamente.

**b.) Planteo del algoritmo:** Mostramos tanto el *diagrama de flujo* como el *pseudocódigo* a continuación:

Figura 10: Diagrama de flujo y pseudocódigo del problema de distribución de cartas.



<sup>5</sup> Aceptamos que sólo funcionará si el programador tuviese la suerte de contar con exactamente 10 casilleros en lugar de 4.

c.) **Desarrollo del programa:** En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'

# asignación de constantes...
n = 5

# título general y carga de datos...
print('Ejemplo 7 - Distribución de cartas')
c1 = int(input('Primer código postal: '))
c2 = int(input('Segundo código postal: '))
c3 = int(input('Tercer código postal: '))

# procesos...
n1 = c1 % n
n2 = c2 % n
n3 = c3 % n

# visualización de resultados...
print('Código postal:', c1, '\tasignado al box: ', n1)
print('Código postal:', c2, '\tasignado al box: ', n2)
print('Código postal:', c3, '\tasignado al box: ', n3)
```

La idea general usada en este problema para calcular el número del box que corresponde a cada carta es la base de una técnica de organización de datos para búsqueda rápida *muy eficiente*, conocida como *búsqueda por dispersión de claves* (o como se la designa en inglés: *hashing*<sup>6</sup>). El estudio de esta técnica está fuera del alcance de este curso: se lleva a cabo en asignaturas o cursos avanzados de estructuras de datos, más adelante en la Carrera.

## 5.] Convenciones de estilo de escritura de código fuente en Python: la guía PEP.

Llegados a este punto parece prudente introducir algunos conceptos referidos a *convenciones y reglas de estilo* cuando se programa en Python. Entre la muy extensa documentación de ayuda del lenguaje, se incluyen algunos documentos oficiales que contienen sugerencias para que los programadores escriban código fuente consistente, entendible y claro para otros programadores. Entre esos documentos se destaca claramente el conocido como *PEP* (iniciales de *Python Enhancement Proposal* o *Propuesta de Mejora de Python*)<sup>7</sup>, que contiene muchos capítulos dependiendo de cuál sea el tema o aspecto del lenguaje sobre el que se están fijando convenciones [4].

En particular, el documento o capítulo *PEP 8 – Style Guide for Python Code*<sup>8</sup> (o *Guía de Estilos para Código Python*) se centra específicamente en recomendaciones de estilo de escritura de código fuente en Python, y desde *PEP 8* tomamos las recomendaciones que siguen [5]. Por supuesto, la guía *PEP 8* es mucho más extensa, y lo que mostramos en este resumen es una selección de las principales convenciones de estilo aplicables a un curso que lleva sólo

<sup>6</sup> En este contexto, la palabra *hashing* se puede entender como "desmenuzamiento" o "hacer picadillo" o "convertir en migajas". Justamente, la acción de tomar un número o clave original y obtener a partir de él un resto simple, da la idea de haber *desmigajado* ese número para tomar sólo una pequeña parte del mismo (el resto) para continuar el proceso.

<sup>7</sup> Disponible en forma completa en <https://www.python.org/dev/peps/>.

<sup>8</sup> Disponible en <https://www.python.org/dev/peps/pep-0008/>.

tres semanas de desarrollo. Por supuesto, recomendamos encarecidamente al estudiante que se acostumbre a consultar las guías *PEP* por su propia cuenta, para mantenerse actualizado en cuanto a recomendaciones, reglas y convenciones.

Todo lenguaje tiene sus convenciones y recomendaciones de estilo. Entre los programadores Python se habla de *Filosofía Python* para referirse a la forma de hacer las cosas en Python. En ese contexto, si un código fuente respeta las convenciones de legibilidad y sencillez de la *Filosofía Python*, se dice que es un *código pythónico* (por *pythonic code*), mientras que lo opuesto (complejo y poco claro) sería un *código no pythónico* (por *unpythonic*). Por lo pronto, entonces, comenzamos transcribiendo una famosa tabla informal de principios pythónicos que fueron presentados graciosamente en forma de aforismos por el desarrollador *Tim Peters* en el documento *PEP 20* (conocido como *El Zen de Python*<sup>9</sup>) [6]:

Figura 11: El Zen de Python...

- ✓ Bello es mejor que feo.
- ✓ Explícito es mejor que implícito.
- ✓ Simple es mejor que complejo.
- ✓ Complejo es mejor que complicado.
- ✓ Plano es mejor que anidado.
- ✓ Disperso es mejor que denso.
- ✓ La legibilidad cuenta.
- ✓ Los casos especiales no son tan especiales como para quebrantar las reglas.
- ✓ Aunque lo práctico le gana a la pureza.
- ✓ Los errores nunca deberían dejarse pasar silenciosamente.
- ✓ A menos que hayan sido silenciados explícitamente.
- ✓ Frente a la ambigüedad, rechaza la tentación de adivinar.
- ✓ Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- ✓ Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- ✓ Ahora es mejor que nunca.
- ✓ Aunque *nunca* es a menudo mejor que *ya mismo*.
- ✓ Si la implementación es difícil de explicar, es una mala idea.
- ✓ Si la implementación es fácil de explicar, puede que sea una buena idea.
- ✓ Los espacios de nombres (*namespaces*) son una gran idea ¡Hagamos más de esas cosas!

Finalmente, presentamos aquí (de la guía *PEP 8*) un subconjunto de las convenciones de estilo de escritura de código en Python más aplicables en este momento del curso (insistimos: no dude en consultar la guía *PEP 8* completa para ampliar estas referencias) [5]:

- Para indentar use 4(cuatro) espacios en lugar de *tabs* (tabuladores). El uso de tabuladores introduce confusión, y con 4 espacios de indentación tendrá el equilibrio justo entre "poco" y "demasiado".

---

<sup>9</sup> Por extraño que parezca, *PEP 20* sólo contiene la tabla de aforismos que hemos transcripto (aunque, por supuesto, en inglés). Disponible en <https://www.python.org/dev/peps/pep-0020/>. Y aún más curioso: estos aforismos están incluidos en el SDK de Python a modo de *Easter Egg* (*Huevo de Pascua*): un tipo de mensaje oculto en un programa que sólo aparece si se sabe cómo encontrarlo (normalmente con alguna combinación no documentada de teclas, o con alguna instrucción especial en un programa). En el caso del *Zen*, lo verá si abre el shell de Python y simplemente ejecuta la instrucción >>> *import this* (no incluya los signos >>> que conforman el prompt del shell).

- Corte sus líneas de código en no más de 79 caracteres por línea. Esto le ayudará a mantener visible gran parte del código fuente en pantallas pequeñas, y le permitirá mostrar varios archivos de código fuente uno al lado del otro en pantallas grandes.
- Use líneas en blanco para separar funciones, clases y largos bloques de código dentro de una función o una secuencia (veremos la forma de desarrollar funciones más adelante en el curso).
- Cuando sea posible, coloque sus comentarios en líneas específicas para esos comentarios (de ser posible, no los agregue en la misma línea de una instrucción).
- Utilice cadenas de documentación (*dostrings*) [2]: mantenga actualizada la documentación de sus programas (veremos esto más adelante el curso).
- Coloque espacios alrededor de los operadores en un expresión, y después de las comas en una enumeración, pero no inmediatamente a los lados de cada paréntesis cuando los use: `a = f(1, 2) + g(3, 4)`.
- No utilice codificaciones de caracteres demasiado extrañas si está pensando en que su código fuente se use en contextos internacionales. En cualquier caso, el default de Python (que es UTF-8) o incluso el llamado *texto plano* (ASCII) funciona mejor.
- Del mismo modo, evite el uso de caracteres no-ASCII en el nombre de un identificador (por ejemplo, en el nombre de una variable) si existe aunque sea una pequeña posibilidad de que su código fuente sea leído y mantenido por personas que hablen en otro idioma.
- En cuanto al uso de comillas dobles o simples para delimitar cadenas de caracteres, no hay una recomendación especial. Simplemente, mantenga la consistencia: si comenzó con una forma de hacerlo, apéguese a ella y no la cambie a cada momento. Cuando necesite incluir un tipo de comilla (simple o doble) en una cadena, use la otra para delimitarla (en lugar de caracteres de control que restan legibilidad).
- Si tiene operadores con diferentes prioridades, considere agregar espacios alrededor de los que tengan menor prioridad, y eliminar los espacios en los que tengan prioridad mayor. Use el sentido común. Sin embargo, nunca coloque más de un espacio y siempre coloque la misma cantidad de espacios antes y después del mismo operador.

**Sí:**

```
m = p + 1
t = x*2 - 1
h = x*x + y*y
c = (a+b) * (a-b)
```

**No:**

```
p=p+1
t= x*2-1
h = x * x + y * y
c = (a + b) * (a - b)
```

- Como ya se indicó, en Python no hay una recomendación especial en cuanto a convenciones de nombres o identificadores. En general, use sentido común: sea cual sea la convención que siga el programador, debe ser claramente distinguible y mantenerse en forma coherente. Otra vez, digamos que en el desarrollo de las Fichas de Estudio de este curso y en los ejemplos de ejercicios y programas resueltos que se entreguen a los estudiantes, aplicaremos las siguientes convenciones en cuanto a nombres de variables (y de nuevo, aclaramos que los estudiantes e incluso sus profesores podrán usar otras convenciones reconocidas si las prefieren. Sólo se sugiere que se mantengan consistentes con ellas):
  - El nombre una variable se escribirá siempre en minúsculas (Ejemplos: *sueldo*, *nombre*, *lado*).

- Cuando se quiera que el nombre de una variable agrupe dos o más palabras, usaremos el *guión bajo* como separador (Ejemplos: *mayor\_edad*, *horas\_extra*).
  - Excepcionalmente designaremos en *mayúsculas sostenidas* a alguna variable que represente algún valor muy conocido y normalmente escrito en mayúsculas (Ejemplos: *PI* (por el número *pi*), o *IVA* (por nuestro bendito impuesto al valor agregado)).
- Nunca utilice la I ('letra ele' minúscula) ni la O ('letra o' mayúscula) ni la I ('letra i' mayúscula) como nombre simple de una variable: en algunas fuentes de letras esos caracteres son indistinguibles de los números 1 y 0 y obviamente causaría confusión.

## Bibliografía

---

- [1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [3] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.
- [4] Python Software Foundation, "PEP 0 - Index of Python Enhancement Proposals (PEPs)," 2015. [Online]. Available: <https://www.python.org/dev/peps/>.
- [5] Python Software Foundation, "PEP 8 - Style Guide for Python Code," 2015. [Online]. Available: <https://www.python.org/dev/peps/pep-0008/>.
- [6] Python Software Foundation, "PEP 20 - The Zen of Python," 2015. [Online]. Available: <https://www.python.org/dev/peps/pep-0020/>.
- [7] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [8] V. Frittelli, D. Serrano, R. Teicher, F. Steffolani, M. Tartabini, J. Fenández and G. Bett, "Uso de Python como Lenguaje Inicial en Asignaturas de Programación," in *Libro de Artículos Presentados en la III Jornada de Enseñanza de la Ingeniería - JEIN 2013*, Bahía Blanca, 2013.

# Ficha 4

## Estructuras Condicionales

### 1.] Fundamentos.

Todos los problemas y casos de análisis que hasta aquí hemos presentado tenían en común el hecho de ser problemas simples de lógica lineal: el algoritmo para resolver a cualquiera de esos problemas consistía en una estructura secuencial de instrucciones o pasos (es decir, un conjunto de instrucciones simples como asignaciones, visualizaciones y/o cargas por teclado, una debajo de la otra). Sin embargo, como pronto veremos, esos casos son poco frecuentes en la práctica real de la programación.

En problemas que no sean absolutamente triviales, es muy común que en algún punto se requiera comprobar el valor de alguna condición, y en función de ello proceder a dividir la lógica del algoritmo en dos o más ramas o caminos de ejecución. Por ejemplo, en un programa de control de acceso a un lugar seguro se debe pedir a cada usuario que cargue su clave de identificación. El programa entonces debería controlar si la clave cargada es correcta y sólo en ese caso habilitar el paso a esa persona. Pero si la clave fuese incorrecta, el programa debería tomar alguna medida alternativa, como sacar un mensaje de alerta por la consola de salida, bloquear una puerta, dar aviso a un supervisor, etc. Pero el hecho es que si sólo se emplean estructuras secuenciales de instrucciones, la situación anterior no podría resolverse.

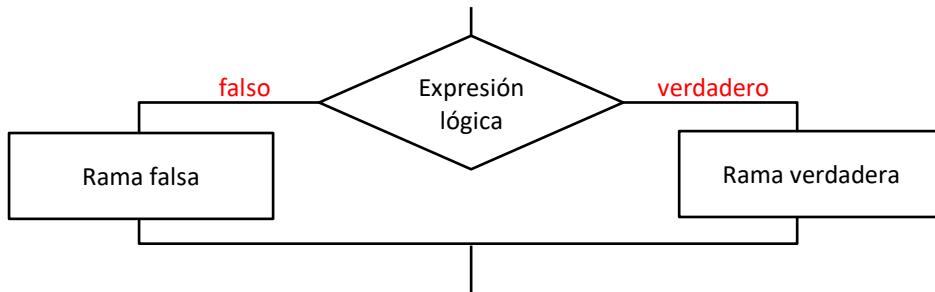
Para casos así, los lenguajes de programación proveen instrucciones específicamente diseñadas para el chequeo de una o más condiciones, permitiendo que el programador indique con sencillez lo que debe hacer el programa en cada caso. Esas instrucciones se denominan *estructuras condicionales*, o bien, *instrucciones condicionales*<sup>1</sup>.

En general, una *instrucción condicional* contiene una *expresión lógica* que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como *la salida o rama verdadera* y *la salida o rama falsa*. Si un programa alcanza una instrucción condicional y en ese momento la expresión lógica es verdadera, el programa ejecutará las instrucciones de la rama verdadera (y sólo esas). Pero si la expresión es falsa, el programa ejecutará las instrucciones de la rama falsa (y sólo esas). La forma de funcionamiento típica de una instrucción condicional se muestra en forma clara en su representación de diagrama de flujo (ver *Figura 1*):

<sup>1</sup> Programar un computador para que sea capaz de responder preguntas hace un poco más cercana la tarea de imitar el pensamiento humano con ese computador, lo cual fue el pilar del trabajo y del sueño de Ada Byron. Y si se trata de sueños combinados con preguntas, en 1968 el escritor Philip Dick publicó una novela corta de ciencia ficción llamada (en español): *¿Pueden los androides soñar con ovejas eléctricas?* en la cual se aborda el tema de la línea que separa a lo natural de lo artificial. Esta novela inspiró el guion de la película *Blade Runner* (de 1982, dirigida por Ridley Scott y protagonizada por Harrison Ford y Rutger Hauer)... en opinión de este profesor, la mejor película de ciencia ficción de todos los tiempos.

En el esquema de la *Figura 1*, se escribieron las palabras **verdadero** y **falso** para etiquetar cada rama, pero a lo largo del desarrollo de nuestro curso adoptaremos la convención de asumir que *siempre la rama verdadera estará a la derecha, y la falsa a la izquierda*. Por lo tanto, en nuestros diagramas de flujo *no etiquetaremos las ramas*.

**Figura 1: Diagrama general de una instrucción condicional típica.**



En el lenguaje Python, una instrucción condicional típica como la que mostramos en la figura anterior, se escribe (esquemáticamente) así [1]:

```
if expresión lógica:  
    instrucciones de la rama verdadera  
else:  
    instrucciones de la rama falsa
```

La palabra reservada *if* da inicio a la estructura condicional. La *expresión lógica* que se quiere evaluar por verdadero o falso se escribe a continuación, y se cierra la línea con el símbolo *dos puntos* (:). Recordemos en este contexto, que una expresión *lógica* es una fórmula cuyo resultado es un *valor lógico* (o *valor de verdad*) (que en Python son los valores *True* y *False*).

La secuencia de instrucciones que conforma la rama verdadera se escribe luego a renglón seguido, y *respetando la indentación o encolumnado*: no olvide que Python identifica a las instrucciones que pertenecen a un mismo bloque de acuerdo a su encolumnado o nivel de indentación. Por eso, las instrucciones que pertenecen al bloque de la rama verdadera deben escribirse encolumnadas hacia la derecha de la palabra *if*, y todas en la misma columna (de lo contrario, Python interpretará de forma incorrecta la estructura).

Una vez terminado de escribir el bloque de la rama verdadera, a renglón seguido se escribe la palabra reservada *else* (volviendo a la *misma* columna de la palabra *if*) seguida a su vez de *dos puntos* (:) y a partir de una nueva línea se escribe la rama falsa *respetando también el encolumnado*.

El siguiente ejemplo (más concreto) es un script que carga por teclado dos números *a* y *b* y muestra el mayor de ambos en la consola estándar:

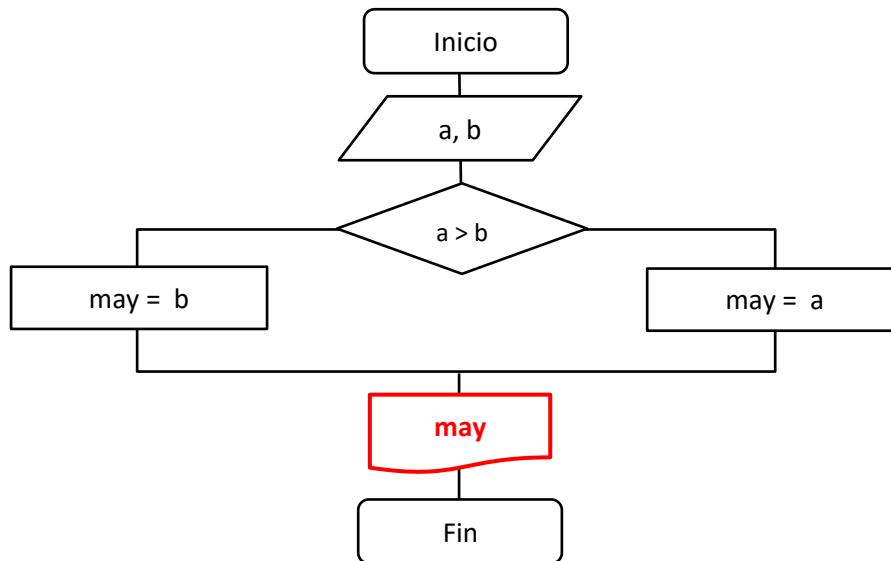
```
a = int(input('A: '))  
b = int(input('B: '))  
  
if a > b:  
    may = a  
else:  
    may = b  
  
print('El mayor es:', may)
```

Para determinar cuál es el mayor entre ambos números, se usa una *instrucción condicional* cuya estructura está remarcada en rojo en el ejemplo. En este caso, la *expresión lógica* que se evalúa es  $a > b$ , la cual puede ser verdadera o falsa dependiendo de los valores cargados en  $a$  y en  $b$ . Si efectivamente el valor de  $a$  fuese mayor que el de  $b$ , entonces se ejecutará la instrucción  $may = a$  (y sólo esa instrucción). Pero si  $a$  no fuese mayor que  $b$ , entonces el script *salteará la rama verdadera* y ejecutará la instrucción que figura en la rama *else* (la rama falsa):  $may = b$ .

Note que *nunca* se ejecutan ambas ramas de una instrucción condicional para el mismo lote de datos: si la expresión lógica es verdadera, se ejecuta la rama verdadera *y sólo la rama verdadera*, ignorando la rama falsa. Y si la expresión lógica fuese falsa, se ejecutará la rama falsa (*else*) *y sólo la rama falsa*, ignorando la rama verdadera.

Observe también la forma en que se escribió la instrucción *print()* de la última línea: se encolumnó *a la misma altura* de la palabra *if* con la que abrió la instrucción condicional, *y no* en la misma columna de las instrucciones de la rama falsa. De esta forma, el programador está indicando que ese llamado a *print()* *no pertenece* a la rama falsa de la condición, sino que está *completamente fuera* de la rama falsa y de la propia condición. De hecho, en el script anterior, el *print()* de la última línea será ejecutado *siempre*, sin importar si la condición entró por la rama verdadera o por la falsa (se dice por eso que esa instrucción es de *ejecución incondicional*). El diagrama de flujo del script anterior (ver *Figura 2*) permite aclarar la idea:

**Figura 2: Diagrama de flujo del script del cálculo del mayor**



El diagrama de la figura anterior muestra en *forma exacta* el funcionamiento lógico del script del cálculo del mayor: si la expresión  $a > b$  es cierta, se ejecutará la instrucción  $may = a$  y luego se mostrará el valor de  $may$ ; pero si la expresión es falsa, se ejecutará la instrucción  $may = b$  y luego también se mostrará el valor de  $may$ .

Recuerde que un diagrama de flujo se construye *de arriba hacia abajo*, tratando de incluir sólo un símbolo debajo de otro, y ahora que aparece la posibilidad de incluir alguna condición, es bueno incorporar nuevas convenciones de trabajo:

- ✓ Por lo pronto, un diagrama de flujo bien planteado *debería tener un único punto de comienzo y un único punto de final* (como el que se ve en la *Figura 2*). Sólo debería haber un único símbolo de *Inicio* al comienzo y arriba, y un único símbolo de *Fin* al final y abajo.
- ✓ Los distintos símbolos se conectan entre ellos con líneas rectas verticales. Si aparece una condición que abra la lógica en dos o más ramas, se traza una línea horizontal a cada lado, pero apenas se disponga de espacio el trazo debe volver a ser descendente.
- ✓ En cada rama el programador debe especificar los procesos a realizar, y apenas le sea posible debe volver a unir esas ramas en un único camino descendente, para llegar en algún momento a un único final.
- ✓ Y mantendremos la convención de que al aparecer una condición, haremos que la salida por verdadero esté siempre a la derecha, y la salida por falso a la izquierda.

Es muy importante que se comprenda el impacto en Python de escribir cada instrucción con la indentación o encolumnado correcto [\[2\]](#). En otros lenguajes, como C, C++, Pascal o Java, un bloque de instrucciones se delimita con símbolos o palabras reservadas especiales. Pero en Python, la pertenencia a un bloque se determina por su indentación. A modo de ejemplo, supongamos que el script del cálculo del mayor hubiese sido escrito (incorrectamente...) así:

```
a = int(input('A: '))
b = int(input('B: '))

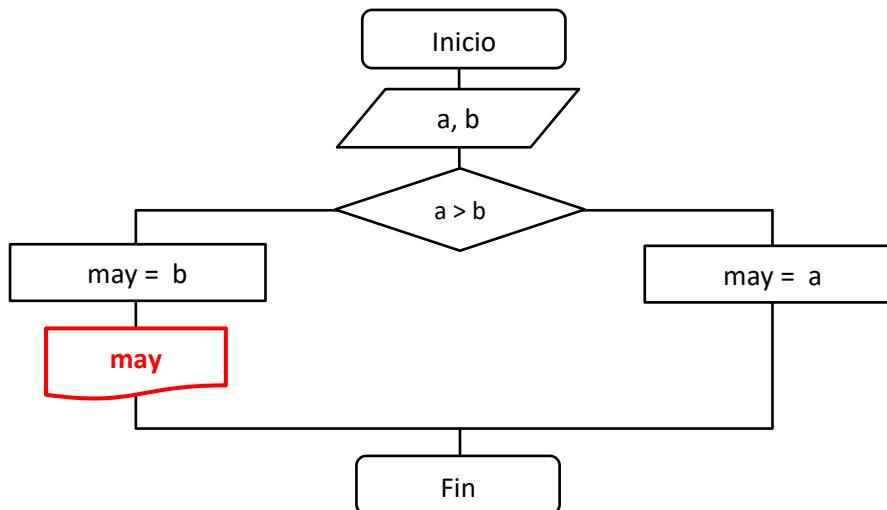
if a > b:
    may = a
else:
    may = b
print('El mayor es:', may)
```

En la forma en que está escrito ahora el script, Python asumirá que la instrucción *print()* para visualizar el valor de *may* pertenece al bloque de la *rama falsa* de la instrucción condicional, y por lo tanto, *sólo será mostrado el valor de may en caso de ser falsa la condición*. Si la condición fuese verdadera, el script terminará sin mostrar nada en la consola de salida. Para terminar de entenderlo, vea el diagrama de flujo que correspondería a este script incorrecto (ver *Figura 3*).

Como se puede ver en esa figura, la lógica es diferente: en la forma en que fue indentada la instrucción *print()* en la segunda versión del script, será entendida como parte del bloque de la rama falsa. Si la condición fuese verdadera, se ejecutaría la instrucción *may = a*, pero luego el script terminaría y no veremos el valor del mayor.

## 2.] Expresiones lógicas. Operadores relacionales y conectores lógicos.

En una ficha anterior hemos visto que en general, una *expresión* es una fórmula compuesta por variables y constantes (llamados *operando*s) y por símbolos que indican la aplicación de una acción (llamados *operadores*). Hemos analizado también el uso de los llamados *operadores aritméticos* básicos de Python (suma, resta, producto, etc.) y sabemos que en función de esto, una *expresión aritmética* es una expresión cuyo resultado es un número. Todos los problemas y ejemplos que se analizaron en las tres primeras fichas de estudio, incluían algún tipo de expresión aritmética.

Figura 3: Diagrama de flujo del script *incorrecto* del cálculo del mayor.

Ahora bien: el hecho de que una expresión entregue como resultado un número, se debe a que los operadores que aparecen en ella son operadores aritméticos y por lo tanto llevan a la realización de alguna operación cuyo resultado será numérico. Sin embargo, en todo lenguaje existen operadores cuya acción no implica la obtención de un número como resultado, sino, por ejemplo, *valores lógicos* de la forma *verdadero* o *falso* (*True* o *False* en Python) y algunos otros operadores entregarán resultados de otros tipos (cadenas de caracteres, por ejemplo). En ese sentido, como ya hemos indicado, una *expresión lógica* es una expresión cuyo resultado esperado es un valor de verdad (*True* o *False*).

Como vimos, las *instrucciones condicionales* (y otros tipos de instrucciones que veremos, como las *instrucciones repetitivas*) se basan *típicamente* en chequear el valor de una *expresión lógica* para determinar el camino que seguirá el programa en su ejecución. Pero veremos oportunamente que en Python es posible que la expresión a evaluar no sea necesariamente lógica y sería válida una expresión de cualquier tipo. Por razones de simplificación del análisis inicial, supondremos por ahora que la expresión a evaluar en una instrucción condicional será de tipo lógico.

Para el planteo de *expresiones lógicas*, todo lenguaje de programación provee operadores que implican la obtención de un valor de verdad como resultado. Los más elementales son los llamados *operadores relacionales* u *operadores de comparación*, que en Python son los siguientes [1]:

Figura 4: Tabla de operadores relacionales (o de comparación) en Python.

Operador	Significado	Ejemplo	Observaciones
<code>==</code>	igual que	<code>a == b</code>	retorna <i>True</i> si <i>a</i> es igual que <i>b</i> , o <i>False</i> en caso contrario
<code>!=</code>	distinto de	<code>a != b</code>	retorna <i>True</i> si <i>a</i> es distinto de <i>b</i> , o <i>False</i> en caso contrario
<code>&gt;</code>	mayor que	<code>a &gt; b</code>	retorna <i>True</i> si <i>a</i> es mayor que <i>b</i> , o <i>False</i> en caso contrario
<code>&lt;</code>	menor que	<code>a &lt; b</code>	retorna <i>True</i> si <i>a</i> es menor que <i>b</i> , o <i>False</i> en caso contrario
<code>&gt;=</code>	mayor o igual que	<code>a &gt;= b</code>	retorna <i>True</i> si <i>a</i> es mayor o igual que <i>b</i> , o <i>False</i> en caso contrario
<code>&lt;=</code>	menor o igual que	<code>a &lt;= b</code>	retorna <i>True</i> si <i>a</i> es menor o igual que <i>b</i> , o <i>False</i> en caso contrario

Los operadores de la tabla anterior permiten plantear instrucciones condicionales en Python para comparar de distintas formas dos valores. Hemos visto un ejemplo de aplicación en el

script para calcular el mayor de dos números. Pero note que en Python, los operadores relacionales también permiten *comparar en forma directa dos cadenas de caracteres*: los operadores == (igual que) y != (distinto de) harán lo que se espera: determinar si dos cadenas son iguales o distintas [1] [2]:

```
cad1 = 'Hola'
cad2 = 'Mundo'

if cad1 == cad2:
    print('Son iguales')
else:
    print('No son iguales')

if cad1 != 'Hello':
    print('No es la palabra Hello...')
else:
    print('Es la palabra Hello...')
```

Ahora bien: si se usan los operadores <, <=, >, o >= para comparar cadenas, Python hará lo que conoce como la *comparación lexicográfica*, que no es otra cosa que la forma de comparación de palabras que normalmente aplican las personas cuando intentan ordenar alfabéticamente un conjunto de palabras, o cuando buscan una palabra en un diccionario. En ese sentido, y para simplificar, si se tienen dos cadenas de caracteres *cad1* y *cad2* y se comparan en forma lexicográfica, entonces *cad1* será considerada *menor* que *cad2* si *cad1* *estuviese antes* que *cad2* en un diccionario. Considere a modo de ejemplo el siguiente script sencillo:

```
cad1 = 'mesa'
cad2 = 'silla'

if cad1 < cad2:
    print('Orden alfabético:', cad1, cad2)
else:
    print('Orden alfabético:', cad2, cad1)
```

En el ejemplo, la condición chequea si la cadena en *cad1* es menor que la cadena en *cad2* y lo hace automáticamente aplicando el criterio lexicográfico. Estrictamente hablando, Python comparará el primer carácter de cada cadena (los valores *cad1[0]* y *cad2[0]*, que son 'm' y 's' respectivamente). Si fuesen diferentes, tomará como *lexicográficamente menor a la cadena que comience con el carácter que aparezca primero en la tabla Unicode*: en este caso, es el carácter 'm', y por lo tanto, *cad1* será considerada menor que *cad2*, haciendo que la instrucción condicional entre por la rama verdadera.

Según en el criterio lexicográfico, si ambas cadenas tuviesen el mismo primer carácter, Python tomará el segundo de ambas (*cad1[1]* y *cad2[1]*) y aplicará la misma regla. Y si esos caracteres fuesen otra vez iguales, Python continuará con el par que sigue hasta encontrar dos diferentes. Obviamente, en el caso en que todos los caracteres de ambas cadenas fuesen iguales, las cadenas mismas serían iguales y la comparación del ejemplo daría un resultado de *False* (si son iguales, *cad1 no es menor* que *cad2*), activando la rama falsa.

En muchas ocasiones el programador necesitará hacer varias comprobaciones al mismo tiempo (por ejemplo, podría querer comprobar si *a > b* y al mismo tiempo *a > c*). La forma

más práctica de hacer esas comprobaciones múltiples consiste en aplicar los llamados *conectores lógicos* (en algunos contextos llamados también *operadores lógicos* o incluso *operadores booleanos*), que en Python son los que siguen [1]:

Figura 5: Tabla de conectores lógicos en Python.

Operador	Significado	Ejemplo	Observaciones
and	conjunción lógica ( <i>y</i> )	<code>a == b and y != x</code>	ver "Tablas de Verdad" en esta misma sección
or	disyunción lógica ( <i>o</i> )	<code>n == 1 or n == 2</code>	ver "Tablas de Verdad" en esta misma sección
not	negación lógica ( <i>no</i> )	<code>not x &gt; 7</code>	ver "Tablas de Verdad" en esta misma sección

Un conector lógico u operador booleano es un operador que permite encadenar la comprobación de dos o más expresiones lógicas y obtener un resultado único. En general, cada una de las expresiones lógicas encadenadas por un conector lógico se designa como una proposición lógica. En la columna *Ejemplo* de la tabla anterior, las expresiones `a == b`, `y != x`, `n == 2` y `x > 7` son proposiciones lógicas.

Si llamamos en general *p* y *q* a dos proposiciones lógicas cualesquiera, podemos mostrar la forma en que operan los conectores lógicos *and* y *or* mediante tablas que muestran qué resultado se obtiene para cada posible combinación de valores de *p* y de *q*. Esas tablas se suelen designar como *tablas de verdad* [3]. Mostramos aquí las tablas de verdad para los conectores *and* y *or*:

Figura 6: Tablas de verdad de los conectores *and* y *or*.

Tabla de verdad del conector <i>and</i>		
<i>p</i>	<i>q</i>	<i>p and q</i>
True	True	True
True	False	False
False	True	False
False	False	False

Tabla de verdad del conector <i>or</i>		
<i>p</i>	<i>q</i>	<i>p or q</i>
True	True	True
True	False	True
False	True	True
False	False	False

Podemos ver que en el caso del conector *and* la salida o respuesta obtenida sólo será verdadera si las proposiciones conectadas son verdaderas al mismo tiempo, y en todo otro caso, la salida será falsa. Por lo tanto, un *and* es muy útil cuando se quiere estar seguro que todas las condiciones impuestas sean ciertas en un proceso. Por caso, suponga que se cargaron por teclado dos variables *sueldo* y *antiguedad* con los datos de un empleado, y se quiere saber si ese empleado gana más de 15000 pesos y tiene al mismo tiempo una

antigüedad de por lo menos 10 años, para decidir si se le otorga o no un crédito. Como *ambas* condiciones son exigibles a la vez, la pregunta requiere un conector *and*, y en Python puede plantearse así:

```
if sueldo > 15000 and antiguedad >= 10:
    print('Crédito concedido')
else:
    print('Crédito rechazado')
```

Note que si alguna de las dos proposiciones fuese falsa, la condición completa sería falsa y se activaría la rama *else*, rechazando el crédito. Por supuesto, lo mismo ocurriría si ambas proposiciones fuesen falsas al mismo tiempo. *Sólo si ambas fuesen ciertas*, la condición sería cierta ella misma, y se activaría la rama verdadera.

El conector *or* opera de forma diferente: la salida será verdadera si al menos una de las proposiciones es verdadera. Sólo si *todas* las proposiciones al mismo tiempo son falsas, se obtendrá un falso como respuesta. El uso de un *or* es valioso (por ejemplo) cuando se quiere determinar si una variable que se acaba de cargar por teclado vale uno de varios posibles valores que se consideran correctos. Por ejemplo, supongamos que queremos saber si la variable *opcion* fue cargada con un 1, un 3, o un 5 (cualquiera de los tres valores es correcto en este ejemplo). En Python, podemos hacerlo así:

```
if opcion == 1 or opcion == 3 or opcion == 5:
    print('Opción correcta')
else:
    print('Opción incorrecta')
```

En este caso, la instrucción condicional se usa para comprobar si el valor de la variable *opcion* coincide con alguno de los números 1, 3 o 5. Cualquiera de las tres proposiciones que fuese cierta, haría cierta también la condición completa y se activaría la rama verdadera. Sólo si todas las proposiciones fuesen falsas, se activaría la rama *else* (por ejemplo, si el valor contenido en la variable *opcion* fuese el 7).

Como vimos, los conectores *and* y *or* se aplican sobre dos o más proposiciones (en general, si un operador aplica sobre dos operandos, se lo suele designar como un *operador binario*). Pero el operador *not* (negación lógica) aplica sobre una sola proposición (y por lo tanto se lo suele designar como un *operador unario*) y su efecto es obtener el *valor opuesto* al de la proposición negada. Por lo tanto, la tabla de verdad del *negador lógico* es trivial:

**Figura 7: Tabla de verdad del conector *not*.**

<b>Tabla de verdad del conector <i>not</i></b>	
<b>p</b>	<b>not p</b>
True	False
False	True

En general, el uso indiscriminado del negador lógico suele llevar a condiciones difíciles de leer y entender por parte de otros programadores, por lo que sugerimos se aplique con cuidado y sentido común. A modo de ejemplo, supongamos que se tiene una variable *edad* con la edad de una persona, y se quiere saber si esa persona tiene al menos 18 años para

decidir si puede acceder al permiso de conducir. Una forma de hacerlo en Python, sería preguntar si la edad ***no es*** menor que 18:

```
if not edad < 18:
    print('Puede acceder al permiso')
else:
    print('No puede acceder al permiso')
```

Si bien lo anterior es correcto y cumple efectivamente con el requerimiento, queda claro que la pregunta podría reformularse de forma de eliminar el negador, dejándola más clara. Sólo debemos notar que preguntar si *edad* no es menor que 18, es exactamente lo mismo que preguntar si *edad* es mayor o igual que 18. El script podría replantearse así:

```
if edad >= 18:
    print('Puede acceder al permiso')
else:
    print('No puede acceder al permiso')
```

### 3.] Precedencia de ejecución de los operadores relacionales y de los conectores lógicos.

En cuanto a la precedencia de ejecución, en Python todos los *operadores relacionales* (o *de comparación*) tienen la *misma precedencia*, y a su vez esta es *mayor que la de los conectores lógicos*, pero *menor que la de los operadores matemáticos* [1]. Obviamente, el uso de paréntesis permite al programador cambiar esas precedencias según sus necesidades. A modo de ayuda mnemotécnica, podemos usar la siguiente relación general:

precedencia(conectores) < precedencia(relacionales) < precedencia(matemáticos)

De esta forma, en cualquier expresión lógica sencilla o compleja el lenguaje agrupará los términos y resolverá *primero las operaciones aritméticas*. Luego, aplicará los *operadores de comparación*, y finalmente ejecutará los *conectores lógicos*.

Sabemos que los operadores aritméticos tienen distinta precedencia (por ejemplo, las sumas y las restas son de menor precedencia que los productos, los cocientes y los restos) y serán aplicados de acuerdo a ella, de izquierda a derecha (salvo el operador de exponentiación (\*\* ) que tiene precedencia mayor).

Los operadores relacionales o de comparación tienen la misma prioridad todos ellos, y serán aplicados de izquierda a derecha. En particular, en Python es especialmente notable el hecho de que estos operadores pueden aplicarse en forma idéntica a como se hace en la notación algebraica normal: una instrucción condicional como:

```
if a > b > c:
```

en Python ejecutará sin problemas [1], significando exactamente lo que se quiere hacer: preguntar si *a > b* y al mismo tiempo preguntar si *b > c*. La misma instrucción podría haber sido escrita así, equivalentemente:

```
if a > b and b > c:
```

Además, es oportuno indicar que tanto el conector *and* como el *or* en Python actúan en forma *cortocircuitada*, lo cual quiere decir que dependiendo del valor de la primera

proposición evaluada, la segunda (o las restantes a partir de ella) podrían no llegar a ser evaluadas (y esto se deduce y se justifica a partir de las *tablas de verdad* de ambos conectores). La idea es la siguiente [1]:

Conejor	Ejemplo	Efecto del cortocircuito
<b>and</b>	if a > b <b>and</b> a > c:	Si la primera proposición es <i>False</i> ( $a > b$ en este caso) la segunda ( $a > c$ ) <i>no se chequea</i> y la salida también es <i>False</i> .
<b>or</b>	if n < 0 <b>or</b> n > 9:	Si la primera proposición es <i>True</i> ( $n < 0$ en este caso), la segunda ( $n > 9$ ) <i>no se chequea</i> y la salida también es <i>True</i> .

De acuerdo a las *tablas de verdad*, es fácil ver que si se tiene un **and** y la primera proposición ya es *False*, no importará el valor de la segunda: la salida será *False* sea cual sea el valor de la segunda, ya que en un **and** es suficiente un *False* en una de ellas para obtener una salida *False*. En forma similar, si se tiene un **or** y la primera proposición es *True*, la salida será *True* sin importar el valor de verdad de la segunda. En Python, estos hechos están ya incorporados en la forma en que el intérprete trabaja cuando evalúa una expresión lógica. Incluso cuando la expresión esté planteada en forma algebraica lineal:

```
if a > b > c:
```

el intérprete Python chequeará la proposición  $a > b$ , y si esta fuese *False*, no evaluará el valor de  $b > c$  y retornará directamente un *False* como salida.

Con respecto a los conectores lógicos, el **not** es de mayor precedencia que el **and**, y este a su vez es de mayor precedencia que el **or**. Si aparecen dos conectores de la misma precedencia, se aplicarán de izquierda a derecha. Las precedencias pueden esquematizarse así:

$$\text{precedencia(or)} < \text{precedencia(and)} < \text{precedencia(not)}$$

Vale decir: si en una misma expresión lógica aparecen varios conectores **and**, **or** o **not** entremezclados, sin paréntesis que cambien las prioridades, los **not** se aplicarán primero, luego los **and**, y finalmente los **or**. Mnemotécnicamente, puede decirse que el conector **not** equivale a un *signo menos* cambiando el signo en una expresión matemática, el **and** equivale a un *producto*, y el **or** a una *suma*<sup>2</sup>.

Para el ejemplo que sigue, recuerde que cualquier expresión de cualquier tipo (aritmética, lógica, etc.) entrega un resultado y ese resultado puede ser asignado en una variable o usado en el contexto que se requiera (como una instrucción condicional). En este caso, mostramos una expresión lógica cuyo valor final (un *True* o un *False*) se asigna en la variable *r1* (que en consecuencia será de tipo *boolean*) (En el desarrollo del ejemplo mostraremos todas las evaluaciones de todas las proposiciones, para dejar en claro el orden en que proceden, pero recuerde que los conectores **and** y **or** son *cortocircuitados*, por lo que algunos de los chequeos de proposiciones que siguen, podrían no llegar a efectuarse realmente):

```
# Considere estas variables con estos valores:  
a, b, c, d, e = 3, 5, 7, 2, 3
```

<sup>2</sup> De hecho, el operador **and** se designa también como el *producto lógico*, y el operador **or** como la *suma lógica*.

```
# Ejemplo: valor final asignado en r1: True
r1 = a > b and c == 3*d or not e != a

# Desarrollo paso a paso: reemplazo de variables por sus valores
# r1 = 3 > 5 and 7 == 3*2 or not 3 != 3

# Desarrollo paso a paso: primero los operadores aritméticos
# r1 = 3 > 5 and 7 == 3*2 or not 3 != 3
# r1 = 3 > 5 and 7 == 6 or not 3 != 3

# Desarrollo paso a paso: segundo los operadores relacionales
# r1 = 3 > 5 and 7 == 6 or not 3 != 3
# r1 = False and False or not False

# Desarrollo paso a paso: tercero los not, and y or en ese orden...
# r1 = False and False or not False
# r1 = False and False or True
# r1 = False or True
# r1 = True
```

Veamos ahora el **mismo** ejemplo, pero cambiando las precedencias con **paréntesis**:

```
# Considere estas variables con estos valores:
a, b, c, d, e = 3, 5, 7, 2, 3

# Ejemplo: valor final asignado en r1: False
r1 = a > b and (c == 3*d or not e != a)

# Desarrollo paso a paso: reemplazo de variables por sus valores
# r1 = 3 > 5 and (7 == 3*2 or not 3 != 3)

# Desarrollo paso a paso: primero los operadores aritméticos
# r1 = 3 > 5 and (7 == 3*2 or not 3 != 3)
# r1 = 3 > 5 and (7 == 6 or not 3 != 3)

# Desarrollo paso a paso: segundo los operadores relacionales
# r1 = 3 > 5 and (7 == 6 or not 3 != 3)
# r1 = False and (False or not False)

# Desarrollo paso a paso: tercero los not, and y or en ese orden...
# ...pero resolviendo los paréntesis antes de aplicar and y or...
# r1 = False and (False or not False)
# r1 = False and (False or True)
# r1 = False and True
# r1 = False
```

Como cierre de esta sección, considere que una variable de tipo *boolean* constituye en sí misma una *expresión lógica* trivial, por lo que si el programador necesita comprobar su valor en una instrucción condicional no es necesario hacer *explícitamente la comparación mediante operadores relacionales*. La siguiente secuencia de instrucciones:

```
r1 = a > b and (c == 3*d or not e != a)
if r1 == True:
    print('Verdadero...')
else:
    print('Falso...')
```

podría haber sido escrita, equivalentemente, *replanteando la condición de esta forma implícita*:

```
r1 = a > b and (c == 3*d or not e != a)
if r1:
    print('Verdadero...')
else:
    print('Falso...')
```

Como se ve, la sola presencia de la variable es suficiente: si su valor es *True*, la instrucción condicional activará la rama verdadera, y si es *False*, entrará por la falsa. En forma similar, si lo que se desea es *invertir la lógica de la comprobación* (esto es, activar la rama verdadera si la variable *r1* vale *False*, y viceversa) está claro que puede hacerse con una *condición explícita* en la forma siguiente:

```
r1 = a > b and (c == 3*d or not e != a)
if r1 == False:
    print('Falso...')
else:
    print('Verdadero...')
```

Pero se puede hacer exactamente lo mismo con una *condición implícita*, simplemente negando el valor de la variable *r1* con el operador *not* (cuya función es justamente esa: invertir la lógica de una comprobación):

```
r1 = a > b and (c == 3*d or not e != a)
if not r1:
    print('Falso...')
else:
    print('Verdadero...')
```

Tómese un par de minutos para analizar este último script y comprender que es *estrictamente equivalente al anterior*: si la variable *r1* vale *False*, el operador *not* obtendrá un *True* y la instrucción condicional entrará por la rama verdadera. Y si vale *True*, el *not* obtendrá un *False* y el camino seguirá por la salida falsa.

#### 4.] Aplicaciones prácticas básicas.

Presentamos en esta sección algunos problemas resueltos, de naturaleza sencilla, que incluyen el uso de condiciones.

**Problema 8.)** *Cargar por teclado dos números enteros. Mostrarlos ordenados de menor a mayor.*

##### a.) Identificación de componentes:

- **Resultados:** Un listado ordenado de dos números. (*men, may: int*)
- **Datos:** Dos números enteros. (*n1, n2: int*)
- **Procesos:** Problema de *ordenamiento de un conjunto*. La idea es que dados los dos números originales *n1* y *n2*, se obtengan otras dos variables *men* y *may* que contengan respectivamente al menor y al mayor de los valores *n1* y *n2*, y luego se muestren los valores finales de *men* y *may*, siempre en ese orden para lograr la visualización ordenada de los datos originales [4].

Esto puede lograrse en forma simple con una instrucción condicional como la que sugerimos a continuación:

```

    si n1 > n2:
        may = n1
        men = n2
    sino:
        may = n2
        men = n1

```

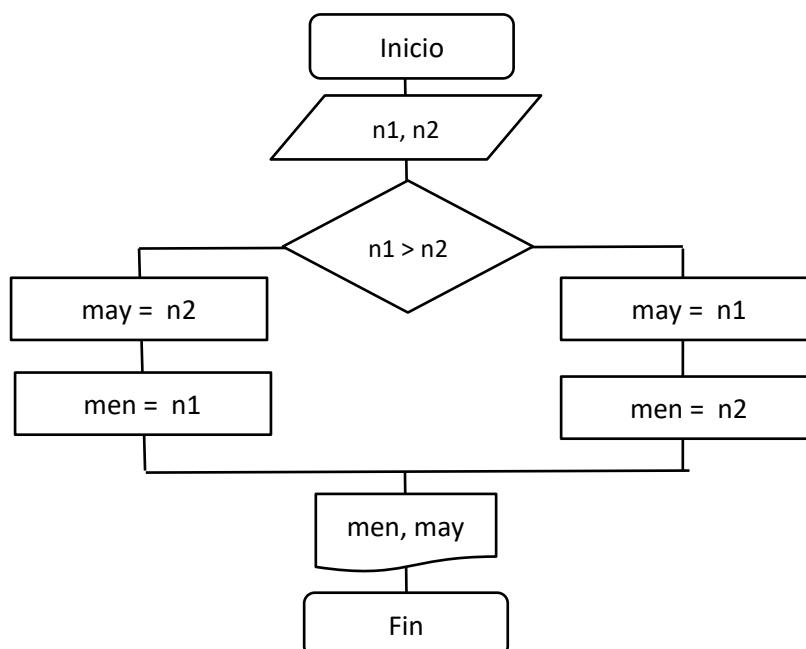
Si la expresión lógica  $n1 > n2$  es cierta, el orden entre  $n1$  y  $n2$  queda totalmente definido (sin necesidad de hacer otra pregunta):  $n1$  es el mayor (y ese asigna en *may*) y  $n2$  es el menor (y se asigna en *men*). Y si la expresión  $n1 > n2$  fuese falsa, entonces el orden *también queda definido* pero a la inversa: se asigna  $n2$  en *may*, y luego  $n1$  en *men*.

**b.) Planteo del algoritmo:** Mostramos tanto el *pseudocódigo* como el *diagrama de flujo* a continuación:

**Figura 8: Pseudocódigo y diagrama de flujo del problema del ordenamiento de dos números.**

Algoritmo:

- 1.) Cargar *n1* y *n2*: dos números enteros.
- 2.) Si *n1 > n2*:
  - 2.1.) *may = n1*
  - 2.2.) *men = n2*
- 3.) sino:
  - 3.1.) *may = n2*
  - 3.2.) *men = n1*
- 4.) Mostrar *men* y *may*: el menor y el mayor, *siempre* en ese orden.



c.) **Desarrollo del programa:** En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```
__author__ = 'Cátedra de AED'

# Título general y carga de datos...
print('Problema del ordenamiento de dos números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))

# Procesos...
if n1 > n2:
    may = n1
    men = n2
else:
    may = n2
    men = n1

# Visualización de resultados..
print('Números ordenados:', men, ' ', may)
```

La instrucción condicional que aparece en este script sigue fielmente la estructura del pseudocódigo que mostramos más arriba. Insistimos en hacer notar la importancia de la **correcta indentación de las instrucciones**, sobre todo las que aparecen en las ramas verdadera y falsa de la condición.

**Problema 9.)** *Cargar por teclado tres números enteros. Determinar si el primero que se cargó es el mayor de los tres (informe en pantalla con un mensaje tal como: Es el mayor o No es el mayor).*

a.) Identificación de componentes:

- **Resultados:** Un mensaje. (*mensaje: cadena de caracteres*)
- **Datos:** Tres números enteros. (*n1, n2, n3: int*)
- **Procesos:** Problema de *determinación del mayor de un conjunto*. Si el primero de los números se carga en la variable *n1*, entonces lo que se debe hacer es comprobar si el valor cargado en *n1* resultó mayor al que se hubiese cargado en *n2* y *n3*.

Esto puede lograrse en con una instrucción condicional como la que sugerimos a continuación, empleando un conector tipo *and* (o sea, un **y lógico**):

```
si n1 > n2 y n1 > n3:
    mensaje = 'El primero es el mayor'
sino:
    mensaje = 'El primero no es el mayor'
```

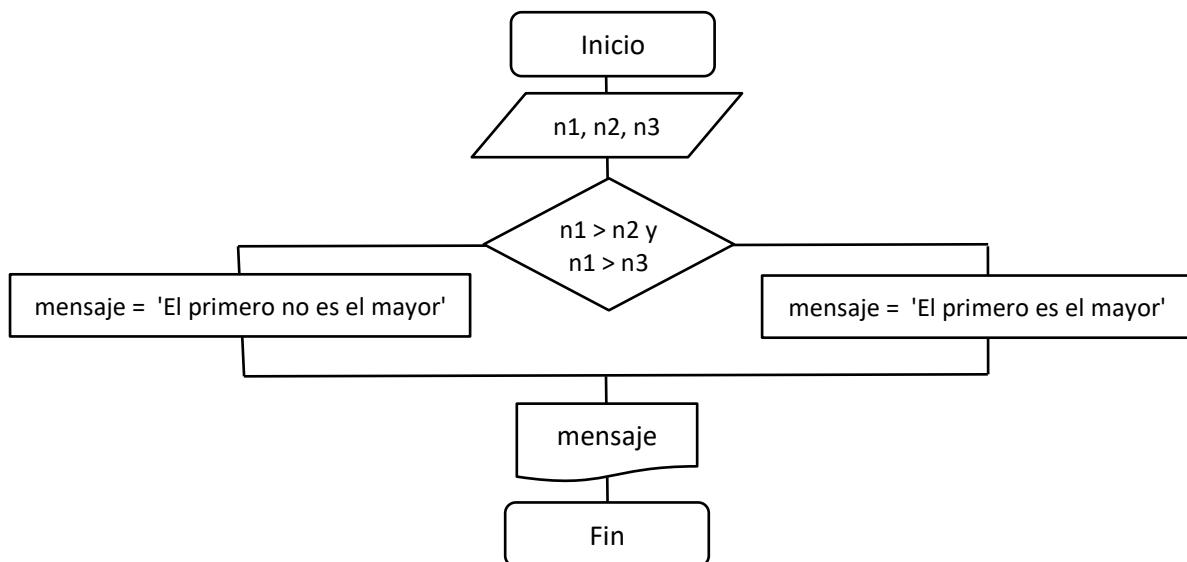
Si la expresión lógica *n1 > n2 y n1 > n3* es cierta, entonces el valor almacenado en *n1* es el mayor de los tres (recuerde que un **y lógico** sólo es verdadero si todas las proposiciones evaluadas son ciertas). En ese caso se asigna en la variable *mensaje* la cadena de caracteres '*El primero es el mayor*'. Y si la expresión *n1 > n2 y n1 > n3* fuese falsa, entonces *n1* no contiene al mayor de los tres: al menos uno de los otros dos números (*n2, n3 o ambos*) es mayor que *n1* (pues el **y lógico** salió por falso). En este caso entonces, la variable *mensaje* es asignada con la cadena '*El primero no es el mayor*'.

**b.) Planteo del algoritmo:** Mostramos tanto el *pseudocódigo* como el *diagrama de flujo* a continuación:

Figura 9: Pseudocódigo y diagrama de flujo del problema del ordenamiento de dos números.

Algoritmo:

- 1.) Cargar **n1**, **n2** y **n3**: tres números enteros.
- 2.) Si **n1 > n2 y n1 > n3**:
  - 2.1.) **mensaje** = 'El primero es el mayor'
- 3.) sino:
  - 3.1.) **mensaje** = 'El primero no es el mayor'
- 4.) Mostrar **mensaje**.



**c.) Desarrollo del programa:** En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```

__author__ = 'Cátedra de AED'

# Título general y carga de datos...
print('Problema de determinar si el primero es el mayor')
n1 = int(input('N1: '))
n2 = int(input('N2: '))
n3 = int(input('N3: '))

# Procesos...
if n1 > n2 and n1 > n3:
    mensaje = 'El primero es el mayor'
else:
    mensaje = 'El primero no es el mayor'

# Visualización de resultados..
print('Conclusión:', mensaje)
  
```

Note que la instrucción `print()` que muestra en pantalla el mensaje pedido, está ubicada *luego de la instrucción condicional*, cuando la misma ha terminado. Sea cual sea el valor de la expresión lógica (*True* o *False*) la instrucción `print()` será ejecutada y el valor de la variable *mensaje* se mostrará correctamente.

**Problema 10.)** *Cargar por teclado tres números enteros que se supone representan las edades de tres personas. Determinar si alguno de los valores cargados era negativo, en cuyo caso informe en pantalla con un mensaje tal como: **Alguna es incorrecta: negativa**. Si todos los valores eran positivos o cero, informe que todas eran correctas.*

a.) Identificación de componentes:

- **Resultados:** Un mensaje. (*mensaje: cadena de caracteres*)
- **Datos:** Tres números enteros. (*n1, n2, n3: int*)
- **Procesos:** Problema de *validación de valores de un conjunto*. Una vez cargados los tres valores se puede usar una condición que chequee cada una de las variables controlando si su valor es negativo, y usar un conector **o lógico** para armar la expresión completa.

Esto puede lograrse en con una instrucción condicional como la que sugerimos a continuación:

```
si n1 < 0 o n2 < 0 o n3 < 0:
    mensaje = 'Alguna es incorrecta: negativa'
sino:
    mensaje = 'Todas son correctas'
```

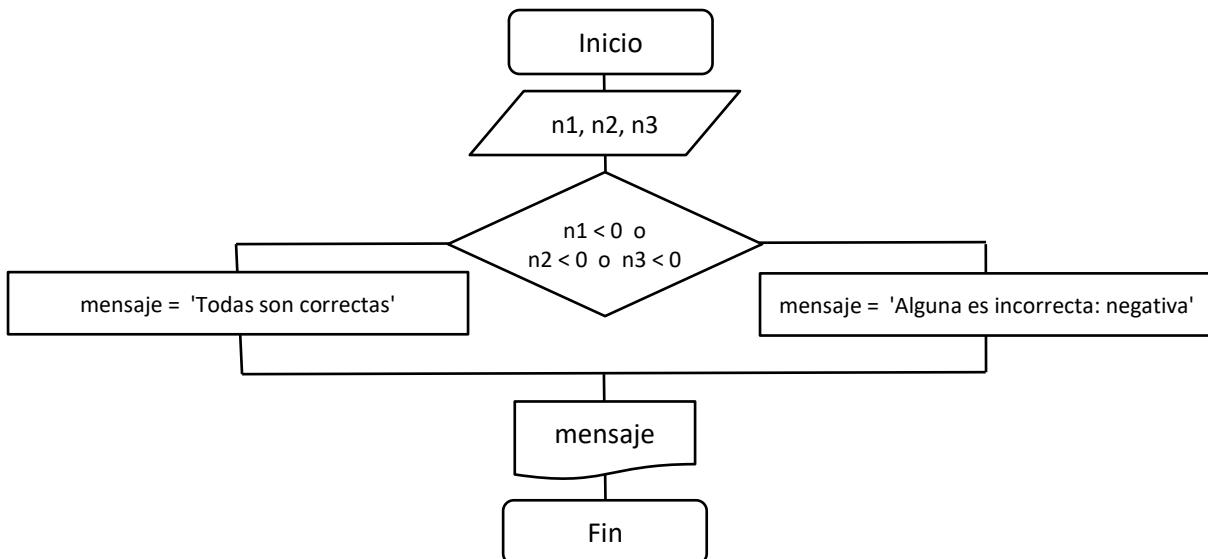
Si la expresión lógica  $n1 < 0 \text{ o } n2 < 0 \text{ o } n3 < 0$  es cierta, entonces alguno de los tres valores es negativo (al menos uno lo es, aunque podrían serlo dos de ellos o incluso los tres). Recuerde que un **o lógico** es verdadero si al menos una de las proposiciones evaluadas es cierta. En ese caso se asigna en la variable *mensaje* la cadena de caracteres '**Alguna es incorrecta: negativa**'. Y si la expresión fuese falsa, entonces *ninguno de los valores era negativo*. En este caso la variable *mensaje* es asignada con la cadena '**todas son correctas**'.

b.) Planteo del algoritmo: Mostramos tanto el *pseudocódigo* como el *diagrama de flujo* a continuación:

Figura 10: Pseudocódigo y diagrama de flujo del problema del ordenamiento de dos números.

Algoritmo:

- 1.) Cargar **n1, n2** y **n3**: tres números enteros (representan edades).
- 2.) Si **n1 < 0 o n2 < 0 o n3 < 0**:
  - 2.1.) **mensaje** = 'Alguna es incorrecta: negativa'
- 3.) sino:
  - 3.1.) **mensaje** = 'Todas son correctas'
- 4.) Mostrar **mensaje**.



c.) **Desarrollo del programa:** En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```

__author__ = 'Cátedra de AED'

# Título general y carga de datos...
print('Problema de determinar si alguna edad es negativa')
n1 = int(input('Edad 1: '))
n2 = int(input('Edad 2: '))
n3 = int(input('Edad 3: '))

# Procesos...
if n1 < 0 or n2 < 0 or n3 < 0:
    mensaje = 'Alguna es incorrecta: negativa'
else:
    mensaje = 'Todas son correctas'

# Visualización de resultados..
print('Resultado del control:', mensaje)
  
```

Otra vez, la instrucción `print()` que muestra en pantalla el mensaje pedido está ubicada *luego de la instrucción condicional*, y sea cual sea el valor de la expresión lógica (`True` o `False`) la instrucción `print()` será ejecutada y el valor de la variable `mensaje` se mostrará correctamente.

## 5.] Generación de valores aleatorios<sup>3</sup>.

Para cerrar los temas generales de esta Ficha, introduciremos brevemente la forma práctica de hacer que un programa en Python genere números al azar (o números aleatorios), lo cual es de mucha utilidad en aplicaciones que se basan en desarrollos probabilísticos (como por ejemplo, el desarrollo de juegos de computadora).

En muchos problemas, se requiere poder pedirle al programa que almacene en una o más variables algún número seleccionado al azar (o al menos, en una forma que no sea

<sup>3</sup> Parte del contenido de esta sección fue aportado por la ing. Romina Teicher en ocasión del planteo del enunciado general del Trabajo Práctico 1.

predecible por el usuario ni por el mismo programador). Esto se conoce como *generación de números aleatorios* (o *números random*) y en general, todo lenguaje de programación provee un cierto número de funciones predefinidas que permiten hacer esto en forma aceptable.

Conceptualmente hablando, estas funciones no generan *realmente* números en forma completamente aleatoria, sino que se basan en algún algoritmo que partiendo de un valor inicial (llamado el *valor semilla* o *seed* del generador) es capaz de generar una secuencia de aspecto aleatorio para cualquiera que no conozca el valor inicial (lo cual suele ser suficiente para aplicaciones generales que requieran algún nivel de aleatoriedad, como los video juegos). Esto se conoce como *generación de números pseudo-aleatorios* y en general los lenguajes de programación suelen tomar ese valor semilla desde el reloj interno del sistema.

Dejando de lado estos elementos (que por ahora tomaremos como tecnicismos), en Python existe lo que se conoce como un *módulo* (o una *librería*) llamado *random*, que contiene las definiciones y funciones necesarias para poder acceder a la gestión de números *pseudo-aleatorios* [1]. Veremos en una ficha posterior la forma de gestionar módulos en Python, pero por ahora baste con saber que para poder usar el contenido del módulo *random* en un programa, debe usarse la instrucción *import random* al inicio del script en donde se requiera. Luego de esto, el programa podrá acceder a las funciones contenidas en ese módulo invocando a esas funciones pero precediendo la invocación con el prefijo "*random.*" A modo de ejemplo, el siguiente script sencillo obtiene un valor aleatorio entre 0 y 1 con la función *random.random()*, lo almacena en una variable *x*, y finalmente lo muestra:

```
__author__ = 'Cátedra de AED'

import random

x = random.random()
print(x)
```

Concretamente, al escribir *random.random()* se está indicando al intérprete que debe buscar la función *random()* dentro del módulo *random* que fue habilitado con la instrucción *import* anterior. La función *random.random()* calcula y retorna lo que se conoce como un *valor de probabilidad*: un número en coma flotante pseudo-aleatorio dentro del intervalo [0, 1) (incluye al cero como posible salida, pero no al 1).

Si lo que necesita el programador es obtener un valor aleatorio de tipo *int* (y no un *float*) en un rango o intervalo entero específico, puede usar la función *random.randrange(a, b)* que obtiene y retorna un número entero pseudo-aleatorio pero comprendido en el intervalo entero  $[a, b-1]$  (incluye al valor *a* como posible salida, pero no al *b*). En el ejemplo siguiente, la variable *y* quedará asignada con un número al azar tomado del intervalo [2, 9] (recuerde: en general en Python, las funciones que trabajan sobre un rango o intervalo de valores, *incluyen al límite izquierdo* de ese intervalo, pero *excluyen al límite derecho*) [1]:

```
__author__ = 'Cátedra de AED'

import random

# número aleatorio entero y tal que 2 <= y < 10
y = random.randrange(2, 10)
print(y)
```

Una alternativa es la función `random.randint(a, b)` que permite hacer lo mismo que `random.randrange(a, b)`, pero de forma tal que el número retornado estará en el intervalo  $[a, b]$  (incluirá a  $b$  como posible salida):

```
__author__ = 'Cátedra de AED'

import random

# número aleatorio entero y tal que 2 <= y <= 10
y = random.randint(2, 10)
print(y)
```

El módulo `random` provee muchas otras funciones para manejo de números pseudo-aleatorios, que en este momento escapan al alcance de lo visto hasta ahora en el curso. Sin embargo, queda al menos una que puede ser de interés inmediato: La función `random.choice(sec)` acepta como parámetro una *secuencia* (que aquí llamamos *sec*) (o sea, una tupla, una cadena, una lista, etc.) y retorna un elemento cualquiera de esa secuencia, elegido al azar. El siguiente script usa dos veces esta función: la primera vez para obtener al azar un número cualquiera de la tupla `sec1 = 2, 10, 7, 9, 3, 4` y la segunda vez para obtener al azar una letra cualquiera de la cadena `sec2 = 'ABCDEFGHI'`:

```
__author__ = 'Cátedra de AED'

import random

sec1 = 2, 10, 7, 9, 3, 4
r1 = random.choice(sec1)
print(r1)

sec2 = 'ABCDEFGHI'
r2 = random.choice(sec2)
print(r2)
```

Para finalizar esta sección, mostramos nuevamente el programa que calcula ordena dos números (problema 8 en esta misma Ficha), pero modificado levemente para que los dos números de entrada **sean generados en forma aleatoria** (tomando números entre 1 y 10), en lugar de ser cargados por teclado:

```
__author__ = 'Cátedra de AED'

import random

# Título general y generación de datos...
print('Problema del ordenamiento de dos números')
print('(los numeros de entrada son generados aleatoriamente...)')

n1 = random.randint(1,10)
n2 = random.randint(1,10)

# Procesos...
if n1 > n2:
    may = n1
    men = n2
else:
    may = n2
    men = n1
```

```
# Visualización de resultados..
print('Números ordenados:', men, ' ', may)
```

## 6.] Expresiones lógicas: forma general de evaluación en Python.

Hemos visto a lo largo de las secciones de esta Ficha la forma de plantear una instrucción condicional en Python y hemos indicado que en general, la forma típica de una instrucción condicional incluye una *expresión lógica* cuyo valor *True* o *False* determina el camino que seguirá la ejecución del programa. Hemos indicado además, que una *expresión lógica* es una fórmula cuyo resultado es un *valor de verdad*, y este hecho surge de la utilización de *operadores relacionales* (o de *comparación*) combinados eventualmente con *conectores lógicos*.

Esto es: si se aplican operadores relacionales y conectores lógicos, la expresión entregará un resultado *True* o *False* y será por ello una expresión lógica. Sin embargo, debemos hacer notar que en Python pueden combinarse estos operadores en formas que podrían resultar extrañas para los programadores poco experimentados. Esta sección está dedicada a estudiar esas aplicaciones.

Específicamente, en Python, la aplicación de *conectores lógicos* (u *operadores booleanos*) como *and*, *or* o *not* hace que *todo operando (constantes o variables) sea considerado como booleano*, incluso si su valor inicial no es de tipo *boolean*. En ese sentido, en una expresión que contenga operadores booleanos, Python asumirá que los siguientes valores equivalen a un *False* [1]:

- *False*
- *None*
- El valor numérico *0(cero)* de todos los tipos
- Las *cadenas de caracteres vacías*, y en general, *todos los objetos contenedores (o estructuras de datos) vacíos* (incluyendo cadenas, tuplas, listas, diccionarios y conjuntos de distintos tipos)

Y en correspondencia con esto, en una expresión con operadores booleanos Python interpretará como un *True* a cualquier otro valor: números diferentes de cero, cadenas o contenedores no vacíos, y el propio valor *True*.

Lo anterior también es aplicable *en cualquier contexto en el cual Python espere encontrar una expresión lógica* (por ejemplo, en una *instrucción condicional* o en una *instrucción repetitiva*), haya o no *operadores booleanos and, or y not* en la expresión analizada.

En lo inmediato, esto implica que si se quiere simplemente chequear si el valor de una variable (posiblemente numérica) es *diferente de cero*, no necesita el programador hacer la *comparación explícita* mediante operadores relacionales, sino que puede simplemente escribir la variable y *comprobarla en forma implícita como si fuese booleana*: el valor *cero* será interpretado como *False* y cualquier otro valor será tomado como un *True*. La siguiente *comprobación explícita*:

```
n = int(input('Ingrese un número entero: '))

if n != 0:
    print('Es diferente de cero')
else:
    print('Es igual a cero')
```

puede hacerse también de *manera implícita*, aun cuando la variable *n* contiene un número, en la forma siguiente:

```
n = int(input('Ingrese un número entero: '))

if n:
    print('Es diferente de cero')
else:
    print('Es igual a cero')
```

Insistimos: en este último ejemplo la variable *n* está siendo usada directamente en el lugar donde Python *esperaría una expresión lógica*, y por lo tanto, el valor de *n* es interpretado *según ese contexto en forma booleana*: si su valor fuese cero, Python lo tomará como un *False* y activará la rama falsa de la instrucción condicional; pero si *n* valiese cualquier otro número, Python lo interpretará como un *True*, activando la rama verdadera.

En forma similar, si la variable analizada fuese una cadena de caracteres, una tupla, una lista u otro tipo de contenedor o variable estructurada se puede chequear en forma implícita si esa variable está vacía o no. En el script siguiente, la variable *cad* representa una *cadena de caracteres* (que asignamos en forma fija para hacer más claro el ejemplo) y *se chequea en forma explícita* si *cad* realmente contiene una cadena o está vacía (la cadena vacía se representa en Python con la constante " (dos comillas simples sin ningún carácter entre ellas) o bien con la constante "" (dos comillas dobles sin ningún carácter entre ellas)):

```
cad = 'Hola'

if cad != '':
    print('La cadena es:', cad)
else:
    print('La cadena está vacía')
```

Pero como queda dicho, la cadena vacía será interpretada por Python como un *False* en un contexto lógico, por lo cual el script podría replantearse así, mediante una *condición implícita*:

```
cad = 'Hola'

if cad:
    print('La cadena es:', cad)
else:
    print('La cadena está vacía')
```

Como dijimos, el uso de conectores lógicos *and*, *or* y *not* convierte la expresión analizada en una *expresión lógica*, sean cuales sean los tipos de las variables sobre las que se aplican estos conectores, y sin importar el contexto. Un caso trivial pero muy ilustrativo ayuda a comenzar a entender la idea:

```
a = 10
b = not a
print('b: ', b)
```

En el pequeño script anterior la variable *a* contiene un valor numérico (el 10) y por lo tanto, *no es booleana*. Sin embargo, en la segunda línea se aplica sobre ella el operador booleano *not*, y se asigna en la variable *b* el *resultado de la negación lógica* de *a*. Podría parecer que

esto no tiene sentido, pero en Python lo tiene: la aparición del *operador booleano not* hace que la expresión *not a* sea considerada como lógica, y por lo tanto el valor de la variable *a* será interpretado en forma *booleana* y *no* en forma *numérica*. Dado que el valor de *a* es 10, en un contexto booleano como es este, Python lo interpretará como un *True* (ya que es diferente de cero) y al negar ese valor con *not*, *el resultado asignado en b será un False*. La ejecución de este script producirá la siguiente salida en consola estándar:

```
b: False
```

Por cierto, note que *no es lo mismo* hacer *b = not a* que hacer *b = -a*. La primera asignación, como vimos, toma el valor de *a* en forma booleana y asigna en *b* un *False* si *a* valía 10 o cualquier otro número diferente de cero. Pero la segunda asignación toma el valor de *a* y cambia su signo, por lo que si *a* valía 10, el valor asignado en *b* será un -10.

Y a partir de todo lo expuesto, prácticamente cualquier expresión que incluya operadores booleanos será válida en una instrucción condicional o en cualquier otra que espere una expresión lógica: los operadores booleanos *and*, *or* y *not* convertirán esa expresión en una expresión lógica aplicando los criterios de transformación de valores no booleanos en booleanos según vimos al inicio de esta sección. A modo de ejemplo de cierre, analicemos este modelo en el cual se asigna en la variable *r* el resultado (*False*) de una expresión lógica basada en las ideas que hemos expuesto.

```
a, b, c, d, e = 10, 4, 5, 5, 3

# Asignación en r del valor False...
r = a and b > d or c != e and not (d or b)
# Desarrollo paso a paso: reemplazo de las variables por sus valores...
r = 10 and 4 > 5 or 5 != 3 and not (5 or 4)

# Desarrollo paso a paso: se aplican los operadores relacionales...
r = 10 and 4 > 5 or 5 != 3 and not (5 or 4)
r = 10 and False or True and not (5 or 4)

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...primero los que vengan entre paréntesis
r = 10 and False or True and not (5 or 4)
r = 10 and False or True and not True

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...luego los not...
r = 10 and False or True and not True
r = 10 and False or True and False

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...luego los and...
r = 10 and False or True and False
r = False or False

# Desarrollo paso a paso: se aplican los conectores lógicos...
# ...y finalmente los or...
r = False or False
r = False
```

Puede verse que el manejo de expresiones en contextos booleanos requiere que el programador preste atención a lo que hace y comprenda el efecto de las trasformaciones de valores que efectúa Python para la conversión lógica. Esta forma de trabajo del lenguaje

puede parecer oscura y hasta contraria al espíritu de Python de favorecer la claridad y la legibilidad del código fuente, pero el hecho es que las expresiones booleanas han sido implementadas así, siguiendo en ese sentido el concepto de manejo de valores lógicos del lenguaje C y del C++. Aquellos programadores que migran a Python desde C o C++ asumen estos manejos como naturales, mientras que aquellos que llegan desde otros lenguajes más estructurados en la gestión de tipos (como Java o Pascal) deberán hacer un esfuerzo para incorporar estos elementos. Y por cierto, aquellos que son programadores absolutamente novatos, deberán hacer el esfuerzo en cualquier caso y de todos modos...

### 7.] Breve discusión: el Álgebra de Boole y las Leyes de De Morgan.

El estudio y sistematización de las operaciones lógicas como *and*, *or* y *not*, forma parte de la rama de la Matemática denominada *Álgebra de Boole*, la que toma su nombre del matemático inglés *George Boole* que fue quien sentó sus bases en 1847. El planteo inicial fue simplemente el de intentar generalizar las técnicas del álgebra para el tratamiento de la lógica proposicional, pero a lo largo del tiempo evolucionó y hoy se utiliza ampliamente en el campo de la electrónica, específicamente en el diseño de circuitos.

Como hemos visto en toda esta Ficha, los elementos fundamentales del *Álgebra de Boole* se utilizan también en el área de la programación de computadoras, ya que está contenida en el planteo de las expresiones lógicas que se usan para el control de instrucciones tales como las condicionales y las repetitivas, entre otras. El planteo y estudio de las ya citadas *tablas de verdad*, y la demostración de teoremas y propiedades relativas a la aplicación de los operadores booleanos, forman parte de esta disciplina.

Hemos visto que el planteo de expresiones lógicas que combinen operadores booleanos, operadores relacionales, operadores aritméticos y variables y constantes de distintos tipos, puede dar lugar a fórmulas de aspecto realmente intimidante que podrían poner a cualquier programador (experto o inexperto) en serios problemas para interpretarlas correctamente. Es común encontrarse con casos de programas en fase de desarrollo que muestran fallas más o menos graves debidas al planteo incorrecto de una o más expresiones lógicas. Por lo tanto, uno de los tantos trabajos que un buen programador debe saber llevar a cabo, es el de simplificar expresiones lógicas, de forma que la expresión simplificada sea más clara, más breve, menos extensa y más simple de entender (lo cual lleva a que finalmente sea más simple de modificar o ajustar si fuese el caso).

En esa dirección apuntan algunas reglas y teoremas del Álgebra de Boole. Dos de esas reglas son muy conocidas y aplicadas en la simplificación de expresiones lógicas y en el diseño de circuitos, y se conocen como *Leyes de De Morgan* [3], en honor al matemático británico (nacido en la India) *Augustus De Morgan* que era contemporáneo de *George Boole*, al punto de que los trabajos de ambos se complementaron entre sí.

Las dos Leyes de *De Morgan* están orientadas a la forma de simplificar la *negación de una conjunción* (o sea, la negación de un *and*) y a la forma de simplificar la *negación de una disyunción* (o sea, la negación de un *or*). Si suponemos que *p* y *q* son dos proposiciones lógicas, usando sintaxis de Python el planteo de ambas reglas es el que sigue:

- 1.] Negación de un *and*:  $\text{not}(p \text{ and } q) \Leftrightarrow \text{not } p \text{ or } \text{not } q$
- 2.] Negación de un *or*:  $\text{not}(p \text{ or } q) \Leftrightarrow \text{not } p \text{ and } \text{not } q$

En lenguaje informal, la primera regla se expresa como: *la negación de un and es igual al or de las negaciones de ambas proposiciones*. Y la segunda se expresa como: *la negación de un or es igual al and de las negaciones de ambas proposiciones*.

Ambas reglas pueden demostrarse por diversas vías. Una forma de hacerlo, es simplemente exponer las tablas de verdad y comprobar que para cada regla, las tablas de las expresiones a la izquierda y a la derecha son equivalentes, lo cual hicimos en la figura que sigue:

**Figura 11: Tablas de verdad de las Leyes de De Morgan.**

Tablas de verdad de las <i>Leyes de De Morgan</i>					
p	q	not(p and q)	not p or not q	not(p or q)	not p and not q
True	True	False	False	False	False
True	False	True	True	False	False
False	True	True	True	False	False
False	False	True	True	True	True

1.] Negación de un **and**  
not(p and q) ⇔ not p or not q
2.] Negación de un **or**  
not(p or q) ⇔ not p and not q

Como se ve, las columnas 3 y 4 son equivalentes entre sí y justamente esas columnas corresponden a las expresiones de la regla 1 (*negación de un and*). Y a su vez también las columnas 5 y 6 son equivalentes entre sí, correspondiendo ambas a las expresiones de la regla 2 (*negación de un or*).

La utilidad práctica de estas dos reglas en programación se hace evidente por medio de un ejemplo: Sin importar los valores que asuman las variables, supongamos una expresión lógica como la que sigue:

```
r = not(a < b and c != b and d >= e and c <= 0)
```

En principio, no hay mayor problema en dejarla como está, pero el planteo en base a un *not* inicial suele producir confusión, ya que el programador debe pensar la expresión completa sin el negador, y finalmente negar la salida para terminar de verla. Dado que esta expresión es la *negación de un encadenamiento de proposiciones con el conector and*, se puede aplicar la regla 1 de *De Morgan*, e intentar eliminar el negador inicial. En el consabido proceso "paso a paso" que sigue, mostramos como se va reduciendo la expresión, hasta llegar a la versión final, más simple (y siempre recuerde que tanto el *and* como el *or* son cortocircuitados, por lo que en realidad, alguna de las proposiciones siguientes podría no llegar a chequearse cuando sea ejecutada):

```
# Expresión inicial...
r = not(a < b and c != b and d >= e and c <= 0)

# Aplicar De Morgan - regla 1... (no parece que ganemos nada...)
r = not a < b or not c != b or not d >= e or not c <= 0

# Reducir los not individuales... y entonces sí... ☺
# Expresión final, equivalente a la inicial, pero más simple...
r = a >= b or c == b or d < e or c > 0
```

La expresión finalmente asignada en  $r$  está compuesta sólo por proposiciones conectadas por el operador *or*: todas las proposiciones individualmente negadas fueron reemplazadas por sus equivalentes en lógica directa (sin los negadores) y los *and* se reemplazaron por los *or*. El truco para lograr la simplificación final, consiste en el reemplazo de cada proposición individual negada por su equivalente directa, de forma de eliminar el negador. Por ejemplo, la expresión *not a < b* es equivalente a la expresión  $a \geq b$  (tienen la misma tabla de verdad), lo cual es simple de deducir: preguntar si *no es cierto que a es menor que b*, es lo mismo que preguntar si *a es mayor o igual que b*. Asegúrese de entender por su propia deducción el resto de los reemplazos que hemos indicado en el ejemplo anterior.

---

## Bibliografía

- [1] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: <https://docs.python.org/3/>.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [3] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.
- [4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

# Ficha 5

## Estructuras Condicionales: Variantes

### 1.] Variantes de la instrucción condicional.

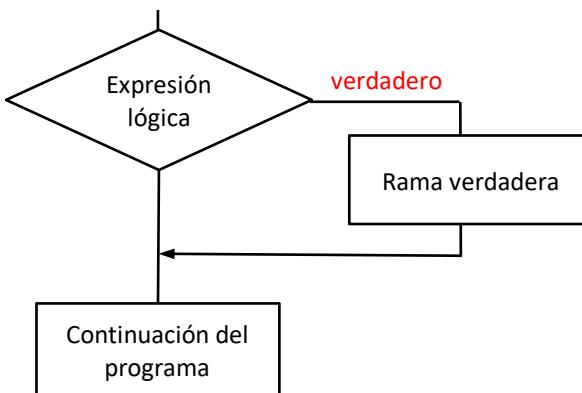
Hemos visto que la forma general típica de una instrucción condicional, consistente en una expresión lógica y dos ramas o salidas: la rama verdadera y la rama falsa. Planteada de esta forma, la instrucción condicional suele ser designada como una *condición doble*. En muchos casos esta forma típica es adecuada y suficiente para el planteo del programa que está desarrollando.

Sin embargo, puede ocurrir (y de hecho es muy común) que para una condición sólo se especifique la realización de una acción si la respuesta es verdadera y no se requiera hacer nada en caso de responder por falso. Para estos casos, en Python y otros lenguajes es perfectamente válido escribir una instrucción condicional que *sólo tenga la rama verdadera, omitiendo por completo la falsa*. Una instrucción condicional de ese tipo se suele designar como *condición simple*, y en ella no se especifica la rama *else*: la instrucción condicional termina cuando termina la rama verdadera. La forma general típica de una *instrucción condicional simple* en Python es la siguiente:

```
if expresión lógica:  
    instrucciones de la rama verdadera  
  
    continuación del programa
```

Las instrucciones de la rama verdadera se encolumnan en un bloque hacia la derecha, del mismo modo que en una condición doble, pero al terminar este bloque se escriben directamente las instrucciones para continuar con el programa, en la misma columna del *if* inicial, sin escribir la rama *else*. El diagrama de flujo de la *Figura 1* aclara la forma de funcionamiento.

**Figura 1: Diagrama general de una instrucción condicional *simple* típica.**



Como se ve en la figura, si la expresión lógica es verdadera se ejecutará la rama verdadera y luego continuará el programa, igual que en las condiciones dobles. Pero si la expresión es

falsa, no se ejecutará ninguna rama especial y también se continuará con el programa. Vale decir: el bloque etiquetado como "*Continuación del programa*" en la figura, **no es** la rama o salida falsa de la condición, ya que ese mismo bloque se ejecutará también al responder por verdadero a la expresión lógica. De hecho, ese bloque está *frente* de la instrucción condicional.

A modo de ejemplo, en el siguiente script el valor de la variable *n* empieza siendo cero. Si el valor de otra variable *x* que se carga por teclado es mayor a cero, se cambia el valor de *n* asignándole el cociente entre *a* y *x*, pero de otro modo no se hace nada y el valor final de la variable *n* queda en 0:

```

n = 0
a = 100
x = int(input('x: '))

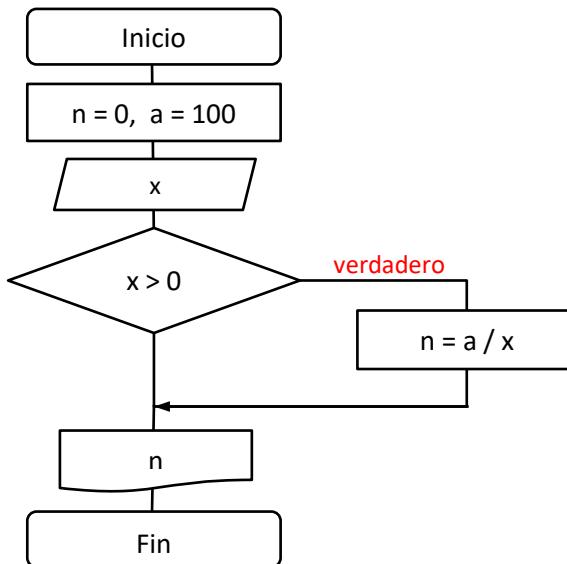
if x > 0:
    n = a / x

print('Valor final:', n)

```

El diagrama de flujo del script anterior, se ve en la figura siguiente:

**Figura 2: Diagrama de flujo del script del cálculo condicional del cociente.**



Por otra parte, y continuando con el estudio de variantes para la instrucción condicional, es posible que en la rama falsa y/o en la rama verdadera de una instrucción condicional se requiera plantear otra instrucción condicional. De hecho, es posible que a su vez cada nueva instrucción condicional incluya otras y así sucesivamente según lo vaya necesitando el programador, sin límites teóricos. Cuando esto ocurre, se tiene lo que se conoce como un *anidamiento de condiciones*. A modo de ejemplo, analicemos el siguiente problema:

**Problema 11.)** *Cargar por teclado tres números enteros y determinar y mostrar el mayor de ellos. No utilice para el proceso la función max() de la librería estándar de Python: diseñe el algoritmo suponiendo que tal función no existe en el lenguaje que usará para el desarrollo del programa.*

a.) Identificación de componentes:

- **Resultados:** El mayor entre tres números. (*may*: int)
- **Datos:** Tres números enteros. (*n1, n2, n3*: int)
- **Procesos:** Problema de *búsqueda del mayor de un conjunto*. El desafío básico es tratar de plantear un esquema de condiciones que lleve al mayor del conjunto, pero de forma que ese esquema incluya *la menor cantidad posible de condiciones*. De esta forma, el programa será más breve, más simple de comprender, y posiblemente más eficiente en cuanto a tiempo de ejecución.

Existen muchas estrategias que podrían aplicarse, pero una muy intuitiva es la siguiente: comenzar con una de las tres variables (sea *n1*) y tratar de determinar si esa variable contiene al mayor, sin preocuparse por las otras dos. Evidentemente, el siguiente esquema de pseudocódigo condicional cumple con esa consigna, y deja el valor del mayor en la variable *may* **si ese mayor estaba en *n1***:

```
si n1 > n2 y n1 > n3:
    may = n1
sino:
    # el mayor no es n1!!!
```

La condición planteada es simple y directa: se pregunta si *n1* es mayor que *n2* y si al mismo tiempo *n1* es mayor que *n3*, usando un conector *y* (conjunción lógica) (o sea, un *and* en Python). Si la respuesta es cierta, evidentemente el mayor está en *n1* y en la rama verdadera de esa condición se asigna *n1* en *may*.

El problema surge si la condición fuese falsa, en cuyo caso no podemos afirmar en forma inmediata cuál es el mayor. Sin embargo, aun cuando en este caso el problema no está resuelto, tenemos una pequeña ganancia: si la condición fue falsa, no sabemos cuál es el mayor, pero sabemos de manera categórica que ese mayor **no es *n1***: si la construcción *and* fue falsa, significa que o bien *n1* es menor que *n2*, o bien es menor que *n3*, o bien es menor que ambas... y en los tres casos implica que *n1* no es el mayor.

Por lo tanto, si la condición entró en la rama falsa sólo nos queda comparar *n2* con *n3* para saber cuál de las dos contiene al mayor, lo cual puede hacerse con otra condición, *anidada en esa rama falsa*. La estructura de pseudocódigo podría verse así:

```
si n1 > n2 y n1 > n3:
    may = n1
sino:
    si n2 > n3:
        may = n2
    sino:
        may = n3
```

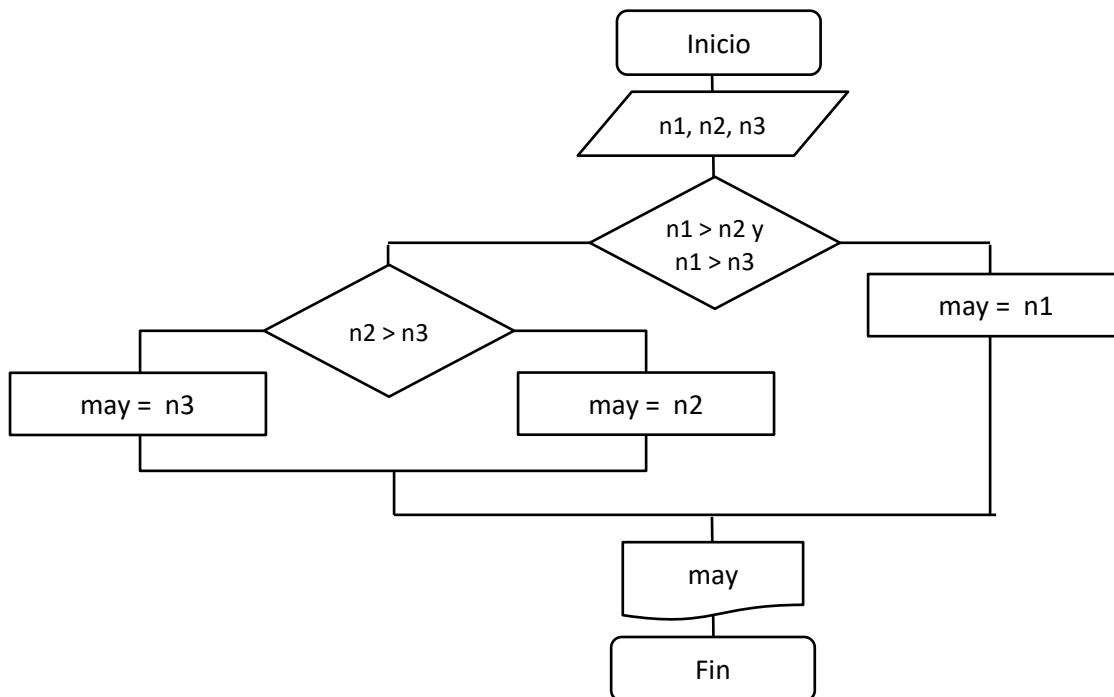
Note la forma en que se escribió el pseudocódigo del *anidamiento de condiciones*: la rama falsa de la primera condición incluye a la segunda condición, escrita (o indentada) en una nueva columna hacia la derecha. Toda la segunda condición está escrita a partir de esa columna, indicando que toda ella pertenece al bloque de la rama falsa de la primera condición. Y a su vez, las ramas verdadera y falsa de la segunda condición, se indentaron en una nueva columna hacia la derecha.

- b.) Planteo del algoritmo:** Mostramos tanto el *pseudocódigo* como el *diagrama de flujo* a continuación:

Figura 3: Pseudocódigo y diagrama de flujo del problema de cálculo del mayor.

Algoritmo:

- 1.) Cargar **n1**, **n2** y **n3**: tres números enteros.
- 2.) Si **n1 > n2** y **n1 > n3**:
  - 2.1.) **may = n1**
- 3.) sino:
  - 3.1.) si **n2 > n3**:
    - 3.1.1.) **may = n2**
  - 3.2.) sino:
    - 3.2.1.) **may = n3**
- 4.) Mostrar **may**: el mayor de los tres números.



c.) Desarrollo del programa: En base al diagrama y/o el pseudocódigo, el script o programa se deduce en forma simple:

```

__author__ = 'Cátedra de AED'

# Título general y carga de datos...
print('Problema del cálculo del mayor entre tres números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))
n3 = int(input('N3: '))

# Procesos...
if n1 > n2 and n1 > n3:
    may = n1
else:
  
```

```

if n2 > n3:
    may = n2
else:
    may = n3

# Visualización de resultados..
print('El mayor es:', may)

```

La instrucción condicional anidada que aparece en este script sigue fielmente la estructura del pseudocódigo que mostramos más arriba. La *segunda instrucción condicional está incluida en el bloque de la rama falsa de la primera*, y esto se indica en Python por medio de la correcta indentación de ese bloque.

Señalemos finalmente que en Python se puede plantear una variante para la instrucción condicional, *que permite evitar el anidamiento excesivo de condiciones*. Se trata de la variante *if – elif*, que puede usarse en forma combinada con el *if – else* normal [1] [2]. La parte *elif* de una condición, cuando está presente, equivale a un *else* que contenga a su vez a otra condición, sin tener que trabajar tanto en la indentación de la estructura anidada y simplificando el código fuente. A modo de ejemplo, una estructura anidada tal como:

```

if opcion == 1:
    print('Se eligió la opción 1')
else:
    if opcion == 2:
        print('Se eligió la opción 2')
    else:
        if opcion == 3:
            print('Se eligió la opción 3')
        else:
            print('Opción no válida')

```

podría reescribirse de la siguiente forma, más compacta, usando *elif*:

```

if opcion == 1:
    print('Se eligió la opción 1')
elif opcion == 2:
    print('Se eligió la opción ')
elif opcion == 3:
    print('Se eligió la opción 3')
else:
    print('Opción no válida')

```

Cada línea que contiene un *elif* continúa en forma directa con la *expresión lógica* que se quería evaluar en la salida falsa de la condición anterior, sin tener que volver a escribir la palabra *if* en esa línea. Observe que en este esquema, la última condición llevó un *else* y no un *elif*, ya que esa rama no incluye (en el esquema original anidado) otra condición, sino directamente un *print()*. Note también que al usar *elif* se simplificó de manera notable el esquema de indentación del script.

Un detalle interesante, es que Python *no dispone de instrucciones condicionales múltiples* especiales como las instrucciones *switch* o *case* de otros lenguajes, pero como se vio en el ejemplo anterior, el uso de *elif* permite plantear una estructura de condiciones anidadas que equivale por completo a una condición múltiple.

Por otra parte, observe que tanto el diagrama de flujo como el pseudocódigo no tienen por qué modificarse si el programador piensa hacer uso de *if – elif* en lugar de *if – else* anidados. La lógica es la misma, y en este caso Python provee una alternativa sintáctica más compacta para escribir el mismo algoritmo. El script para el cálculo del mayor que antes mostramos con condiciones *if – else* anidadas, puede replantearse así, mediante instrucciones *if – elif*, sin tocar en absoluto el diagrama de flujo ni el pseudocódigo de la *Figura 3*:

```
__author__ = 'Cátedra de AED'

# Título general y carga de datos...
print('Problema del cálculo del mayor entre tres números')
n1 = int(input('N1: '))
n2 = int(input('N2: '))
n3 = int(input('N3: '))

# Procesos...
if n1 > n2 and n1 > n3:
    may = n1
elif n2 > n3:
    may = n2
else:
    may = n3

# Visualización de resultados..
print('El mayor es:', may)
```

Para cerrar esta sección, mostramos ahora un nuevo problema de aplicación de los temas vistos hasta aquí:

**Problema 12.)** *Una compañía de alquiler de automóviles necesita un programa que calcule lo que se debe cobrar a cada cliente, teniendo en cuenta el kilometraje recorrido por el cliente al devolver el automóvil:*

- i. *Si el cliente no superó los 300 km recorridos se deberá cobrar \$500.*
- ii. *Para recorridos desde más de 300 km y hasta no más de 1000 km se le cobrará \$500 más el kilometraje excedente a los 300, a razón de \$3 por kilómetro.*
- iii. *Para recorridos mayores a 1000 km se le cobrará \$500 más el kilometraje excedente a los 300, a razón de \$1.5 por kilómetro.*

a.) Identificación de componentes:

- **Resultados:** El monto a cobrar. *(monto: float)*
- **Datos:** El kilometraje recorrido. *(kilometraje: int)*
- **Procesos:** Problema de aritmética básica aplicado a un contexto comercial. La base del algoritmo es determinar en qué rango está el valor de la variable *kilometraje*, lo cual puede hacerse con condiciones anidadas y prestando atención a los límites de esos rangos. Por otra parte, sabemos que si *kilometraje* es mayor que 300, debemos saber cuántos kilómetros de excedente hubo, para calcular con ello el monto final a pagar. Todo eso puede resumirse en el siguiente esquema de pseudocódigo general:

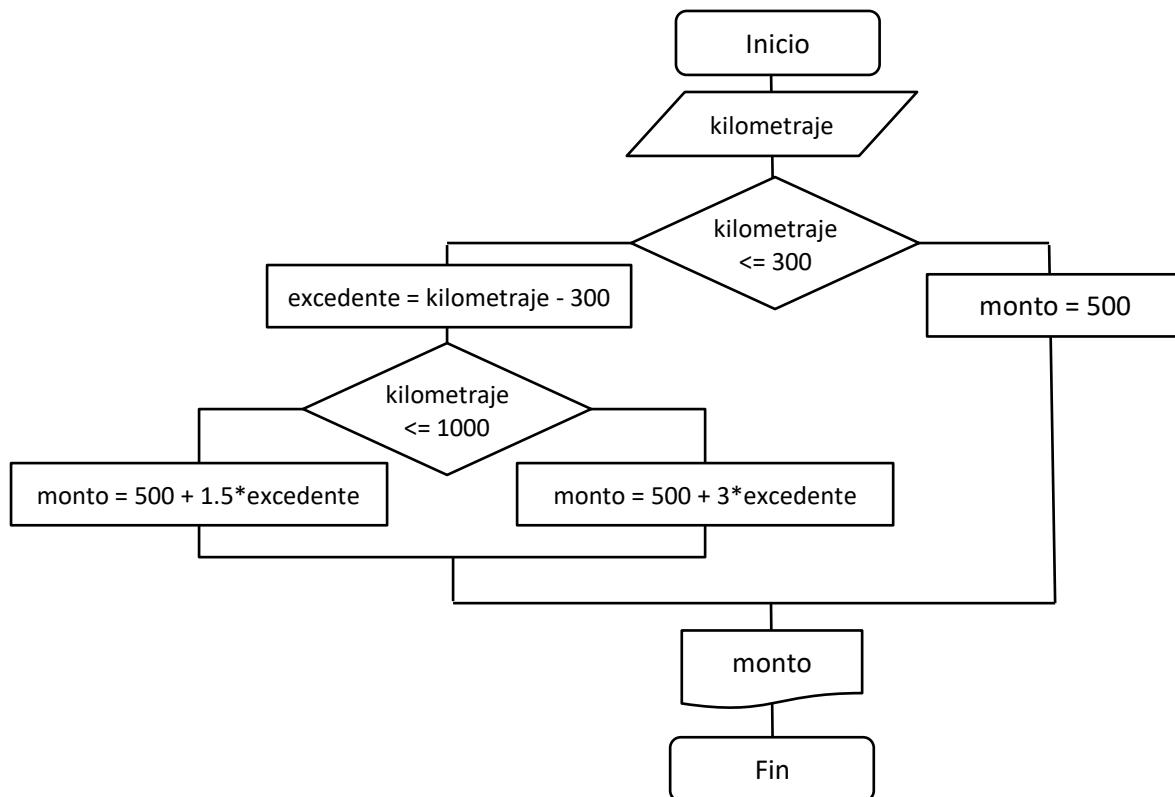
```

    si kilometraje <= 300:
        monto = 500
    sino:
        excedente = kilometraje - 300
        si kilometraje <= 1000:
            monto = 500 + 3*excedente
        sino:
            monto = 500 + 1.5*excedente

```

La idea es simple: si *kilometraje* es mayor a 300, se calcula por única vez el valor del *excedente*, y luego se determina si el kilometraje superó o no los 1000 kilómetros. Si lo hizo, se calcula el monto final sumando  $1.5 * \text{excedente}$  al valor base de 500. Y si no superó los 1000 kilómetros, se suma  $3 * \text{excedente}$  al monto base de 500.

**b.) Planteo del algoritmo:** Mostramos ahora el *diagrama de flujo* de la solución:



**c.) Desarrollo del programa:** Como siempre, en base al diagrama el script se deduce en forma inmediata:

```

__author__ = 'Cátedra de AED'

# Título general y carga de datos...
kilometraje = int(input("Ingrese la cantidad de kilómetros: "))

# Proceso: cálculo del valor a cobrar
if kilometraje <= 300:
    monto = 500
else:
    excedente = kilometraje - 300
    if kilometraje <= 1000:

```

```

        monto = 500 + 3*excedente
else:
    monto = 500 + 1.5*excedente

# Visualización de resultados
print("El importe a pagar es:", monto)

```

## 2.] Expresiones de conteo y acumulación.

A medida que se avanza en el estudio y planteo de algoritmos y programas para problemas cada vez más complejos, se verá que en la mayoría de esos programas será necesario eventualmente llevar a cabo procesos de *conteo* (por ejemplo, determinar cuántas veces apareció un número negativo), o de *sumarización* (por ejemplo, determinar cuánto vale la suma de todos los valores que tomó la variable *x* a lo largo de la ejecución de programa, suponiendo que *x* cambia de valor durante esa ejecución). El primer caso se resuelve incorporando una *variable de conteo* (o simplemente un *contador*), y el segundo, incorporando una *variable de acumulación* (o simplemente un *acumulador*) [3].

En ambas situaciones se trata de variables que en una expresión de asignación aparecen en *ambos miembros*: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de ese cálculo. Los siguientes son dos ejemplos de *expresiones de conteo* o *acumulación* (en el primero, la variable *a* se usa como un *contador*, y en el segundo la variable *b* se usa como un *acumulador*):

```

a = a + 1
b = b + x

```

En general, un *contador* es una variable que sirve para *contar* ciertos eventos que ocurren durante la ejecución de un programa. Intuitivamente, se trata de una variable a la cual se le suma el valor *1 (uno)* cada vez que se ejecuta la expresión. Esto es así porque *contar* normalmente significa *sumar 1*, pero en la práctica puede sumarse cualquier valor, o incluso no hacer una suma sino cualquier otra operación que involucre a cualquier *constante*.

Entonces, técnicamente, un *contador* es una variable que actualiza su valor en términos de *su propio valor anterior* y de *una constante*. A modo de ejemplo: la variable *a* del siguiente esquema actúa a modo de *contador* para determinar cuántos números son negativos entre tres que se cargan por teclado:

**Figura 4: Uso básico de un contador.**

```

__author__ = 'Catedra de AED'

a = 0
num = int(input('Ingrese un número: '))
if num < 0:
    a = a + 1

num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1

num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1

print('Cantidad de negativos cargados:', a)

```

Cada vez que alguno de los números cargados en la variable *num* sea negativo, se ejecuta la instrucción indicada en color azul: *a = a + 1* la cual funciona así:

- ✓ En primer lugar se ejecuta la parte *derecha* de la asignación, con el valor actual de *a*. Si *a* comenzó valiendo cero, entonces la primera vez que se ejecute la expresión *a + 1* se obtiene un uno.
- ✓ En segundo lugar se asigna el valor así obtenido en la misma variable *a*, con lo cual se cambia el valor original. En este caso, la primera vez que se cargue un negativo en *num* la variable *a* quedará valiendo 1. Si se carga un segundo negativo en *num*, *a* quedará valiendo 2, y así cada vez que se ingrese un negativo, asignando a la variable *a* su valor anterior sumado en uno.

En nuestro ejemplo, el valor final de la variable *a* indicará cuántos números negativos se ingresaron, mostrándolo con un mensaje. Observar que la variable que se usa como contador *debe* ser inicializada con un valor adecuado antes de comenzar a usarla, para eliminar cualquier valor residual y garantizar que el conteo comience desde el valor correcto (normalmente el *cero*, como en el ejemplo, aunque esto depende de lo pedido en el problema).

Del mismo modo que en la expresión *a = a + 1* la variable *a* funciona como *contador*, en forma similar la variable *c* en la expresión *c = c - 1* funciona como *decrementador*: va restando de a uno a partir del valor original de la variable *c*. Pero también notemos que cualquier *constante* y cualquier *operador* sirven para formar la expresión general. En los siguientes ejemplos mostramos *contadores* de formas diversas (asegúrese de entender lo que cada instrucción hace cada vez que se ejecuta):

```
a = a + 1
b = b - 1
c = c + 2
d = d * 4
e = e / 3
```

Por otra parte, un *acumulador* o *variable de acumulación* es básicamente una variable que permite *sumar* los valores que va asumiendo *otra variable* o bien *otra expresión* en un proceso cualquiera. Técnicamente, y en general, un *acumulador* es una variable que actualiza su valor en términos de su propio valor anterior y el valor de *otra variable* u *otra expresión*. Por ejemplo, el siguiente esquema permite ir *sumando* los valores que asume la variable *x*, usando una *variable de acumulación* *s*:

**Figura 5: Uso básico de un acumulador.**

```
__author__ = 'Catedra de AED'

s = 0
x = int(input('Ingrese un número: '))
s = s + x
x = int(input('Ingrese otro: '))
s = s + x
x = int(input('Ingrese otro: '))
s = s + x

print('La suma de los valores cargados es:', s)
```

En este esquema, se cargan por teclado distintos valores en la variable *x* a lo largo de tres instrucciones de carga. La instrucción *s = s + x* funciona en forma similar a como trabajaba un contador, pero ahora no se va sumando de a uno sino que se va *sumando el valor que en cada carga tenga la variable x*. Si dicha variable asume sucesivamente los valores 3, 4 y 7 entonces la variable *s* quedará valiendo 14. Observar que al usar un *acumulador* también es obligación del programador asegurar un valor inicial a la variable de acumulación antes de empezar a usarla, para que efectivamente quede definida o para eliminar cualquier valor

residual si ya estaba definida y se usó en otro lugar del mismo programa. En el ejemplo anterior, la variable `s` se inicializó en cero, y luego comenzó el proceso de carga de los valores de la variable `x`.

Según la definición dada de un acumulador, las siguientes expresiones también son expresiones de acumulación:

```
s = s + x
b = b * z
a = a - y
p = p / t
c = c + 2*x
```

Es interesante notar que en Python (como en otros lenguajes) cualquier *expresión de conteo o de acumulación* responde a la *forma general* siguiente [1]:

```
variable = variable operador expresión
```

donde `variable` es la variable cuyo valor se actualiza (y aparece en ambos miembros de la expresión de asignación) y `expresión` es una constante, una variable o una expresión propiamente dicha (formada a su vez por constantes, variables y operadores). Y el hecho es que en el lenguaje Python cualquier expresión que venga escrita en la *forma general* anterior, se puede escribir también en la *forma resumida* siguiente:

```
variable operador= expresión
```

A modo de ejemplo, veamos las siguientes equivalencias:

Forma general	Forma resumida
<code>a = a + 1</code>	<code>a += 1</code>
<code>b = b - 1</code>	<code>b -= 1</code>
<code>c = c + 2</code>	<code>c += 2</code>
<code>d = d * 3</code>	<code>d *= 3</code>
<code>e = e / 4</code>	<code>e /= 4</code>
<code>s = s + x</code>	<code>s += x</code>
<code>b = b * z</code>	<code>b *= z</code>
<code>a = a - x</code>	<code>a -= x</code>
<code>p = p / t</code>	<code>p /= t</code>
<code>c = c + 2*x</code>	<code>c += 2*x</code>

### 3.] Variables centinela (o banderas).

En muchas ocasiones los programadores necesitan *marcar* de alguna forma *un suceso que hubiese ocurrido* a lo largo de la ejecución de un programa, para luego poder chequear en distintos puntos del programa y en forma simple si ese suceso realmente tuvo lugar o no, para ejecutar en consecuencia ciertas acciones.

Las situaciones prácticas que darían lugar a esta clase de necesidad son muy diversas y dependen del contexto de cada problema: por ejemplo, se podría querer saber si algunos de los números que se cargaron en un programa eran negativos para que al finalizar la carga se muestre un único mensaje de advertencia, o si todos los datos cargados en otro programa correspondían a mujeres para lanzar al final un único aviso de posibles beneficios por maternidad, o si en algún momento el valor de una variable cambió para que al terminar el proceso se avise al administrador del sistema de un posible error.

En estos casos los programadores pueden recurrir a diferentes técnicas para marcar el suceso. Una técnica muy común se designa como uso de *variables centinela*, también conocida como uso de *variables bandera*, o uso de *señales* o simplemente uso de *banderas* (o uso de *flags* por su traducción al inglés) [3].

Una *bandera* es una variable (típicamente *booleana*, aunque podría ser de cualquier otro tipo) cuyo valor se controla en forma metódica a lo largo de la ejecución de un programa, de forma que cada valor posible se asocia a la ocurrencia o no de un evento (por ejemplo, si el evento ocurrió la *bandera* se asigna en *True*, y mientras el evento no ocurra el *flag* se mantiene en *False*). En cualquier momento el programador puede chequear el estado de la *bandera* y saber si el evento marcado ocurrió o no.

La mejor forma de terminar de incorporar la idea del uso de una *bandera*, es analizando su aplicación efectiva en un programa, para lo cual proponemos el siguiente problema (que además servirá como ejemplo de aplicación del uso de *variables de conteo*):

**Problema 13.)** *Se cargan por teclado las notas obtenidas por un estudiante en tres parciales realizados durante el cursado de una materia universitaria. Además, se carga la nota final que ese estudiante obtuvo en el desarrollo de los trabajos prácticos en esa misma materia. Se sabe que al terminar el cursado de la materia, todo alumno puede quedar en uno de los siguientes estados académicos:*

- a. *Libre: si no llegó a cumplir con las condiciones para ser Regular.*
- b. *Regular: si aprobó al menos dos de los tres parciales con nota de 4 o más y además obtuvo nota de 4 o más en la nota final de trabajos prácticos.*
- c. *Promocionado: si aprobó los tres parciales con nota de 7 o más pero con promedio entre ellos de 8(ocho) o más, y además obtuvo nota de 8 o más en la nota final del práctico.*
- d. *Aprobado: si aprobó los tres parciales con nota de 7 o más pero con promedio entre ellos de 9(nueve) o más, y además obtuvo nota de 8 o más en la nota final del práctico.*

*El programa debe determinar y mostrar por pantalla el estado en que finalmente quedó el estudiante.*

**Discusión y solución:** Asumiremos que las notas de los tres parciales serán cargadas por teclado en tres variables *n1*, *n2* y *n3* y que la nota final del práctico se cargará a su vez en la variable *np*. El programa determinará la condición final del estudiante, y almacenará en la variable *condicion* una cadena de caracteres que podrá ser '*Libre*', '*Regular*', '*Promocionado*' o '*Aprobado*'.

En primera instancia, notemos que la diferencia entre el estado *Promocionado* y el estado *Aprobado* es el valor del promedio de las notas de los tres parciales (que llamaremos *pp*): en el primer caso se pide un promedio *pp* de 8 o más y en el segundo un promedio *pp* de 9 o más. Pero en ambos estados el resto de las exigencias es el mismo: aprobar los tres parciales con nota de 7 o más, y obtener nota de 8 o más en la nota final del práctico *np*.

Lo anterior implica que tanto para verificar si el estudiante está *Promocionado* como para saber si está *Aprobado* el programa deberá chequear si los tres parciales están aprobados y con notas de 7 o más en cada caso y también si la nota final de práctico es 8 o más. La condición para hacer eso tendría la forma que sigue:

```
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8:
```

lo cual incluye cuatro proposiciones encadenadas con operadores tipo *and*. Si esta misma condición debe plantearse en dos o más lugares diferentes del programa, y además hay que agregar alguna otra proposición (por ejemplo: el chequeo del promedio entre los tres

parciales) entonces cada condición sería bastante extensa de escribir. Por caso, estas serían las dos condiciones a plantear para saber si el estudiante está *Aprobado* o *Promocionado*:

```
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8 and pp >= 9:
    condicion = 'Aprobado'
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8 and pp >= 8:
    condicion = 'Promocionado'
```

El esquema anterior ilustra una situación simple pero válida en la cual podría usarse una *bandera* para simplificar el control. Claramente, el programador necesita *dejar marcado* el evento consistente en que los tres parciales estén aprobados con nota de 7 o más, al mismo tiempo que la nota final del práctico sea de 8 o más.

Procederemos para eso de la siguiente manera: sea la variable booleana *notas\_ok* que comenzaremos asignando con el valor *False* antes de hacer el chequeo de las notas. La idea es que el valor de esa variable indique lo que hasta ese momento se sabe respecto de los parciales y la nota del práctico: usamos el valor *False* para registrar que hasta ese momento, no hay nada que nos indique que esas notas cumplen con los requisitos pedidos. Si luego del chequeo condicional se descubre que efectivamente esas notas son correctas, en ese momento el valor de *notas\_ok* se cambia a *True* y se deja marcado con eso que las notas han sido verificadas con éxito.

En cualquier lugar del programa en que el programador necesite saber si las notas de los parciales y la nota del práctico cumplían con los requisitos exigidos, se podrá preguntar por el valor final de *notas\_ok*: el valor *True* indicará que todo estaba en orden (*notas\_ok = True*) o no (*notas\_ok = False*). El esquema podría quedar modificado así:

```
notas_ok = False
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8:
    notas_ok = True
if notas_ok and pp >= 9:
    condicion = 'Aprobado'
if notas_ok and pp >= 8:
    condicion = 'Promocionado'
```

Recordemos que si la variable *notas\_ok* es booleana, entonces no es necesario usar explícitamente el operador == para verificar su valor. Las dos últimas condiciones del esquema anterior son completamente equivalentes a las que se muestran a continuación:

```
if notas_ok == True and pp >= 9:
    condicion = 'Aprobado'
if notas_ok == True and pp >= 8:
    condicion = 'Promocionado'
```

Queda claro que el programador podría haber planteado el esquema de alguna forma alternativa para evitar tener que chequear dos veces las notas de los parciales y el práctico (invitamos a los estudiantes a que piensen por sí mismos en qué forma se podría haber hecho...) pero la idea aquí era simplemente mostrar la manera de aplicar en forma efectiva una *variable bandera* para marcar un evento. El programa completo que resuelve el problema se muestra a continuación:

```
__author__ = 'Catedra de Algoritmos y Estructuras de Datos'
# Carga de datos...
n1 = int(input('Nota en el parcial 1 (cargue 0 si no hizo el parcial): '))
n2 = int(input('Nota en el parcial 2 (cargue 0 si no hizo el parcial): '))
```

```

n3 = int(input('Nota en el parcial 3 (cargue 0 si no hizo el parcial): '))
np = int(input('Nota final del practico: '))

# Procesos...
# ...cuantos parciales aprobó?
cpa = 0
if n1 >= 4:
    cpa += 1
if n2 >= 4:
    cpa += 1
if n3 >= 4:
    cpa += 1

# ...aprobó los tres con nota >= 7 y el practico con nota >= 8?
notas_ok = False
if n1 >= 7 and n2 >= 7 and n3 >= 7 and np >= 8:
    notas_ok = True

# ...promedio entre los tres parciales? (con coma y sin redondear)...
pp = (n1 + n2 + n3) / 3

# ...empezamos asumiendo que el estudiante es Regular...
condicion = 'Regular'

# ...pero si no lo es, cambiamos su estado...
if cpa < 2 or np < 4:
    condicion = 'Libre'
elif notas_ok and pp >= 9:
    condicion = 'Aprobado'
elif notas_ok and pp >= 8:
    condicion = 'Promocionado'

# Visualización de resultados...
print('Condición final del estudiante:', condicion)

```

Adicionalmente, puede verse que el programa usa una *variable de conteo* llamada *cpa* para llevar la cuenta de la cantidad de parciales que el estudiante aprobó, lo cual es requerido para determinar si ese estudiante alcanzó o no la condición de Regular. El valor inicial de ese contador se ajusta en 0, y luego simplemente se incrementa en 1 por cada parcial con nota mayor o igual a 4 que se detecte.

#### 4.] Caso de análisis: un algoritmo para ordenar tres números contenidos en tres variables.

Nos proponemos analizar con cierta profundidad un problema que se presentará en reiteradas ocasiones a todo lo largo de este curso: el ordenamiento de un conjunto de valores. En estas primeras etapas de la asignatura el problema aparecerá simplificado: dado un conjunto de unos pocos números, mostrar esos números en forma ordenada de menor a mayor.

En este caso, nuestro objetivo será *ordenar un conjunto de sólo tres números, contenidos en tres variables*. Podemos hacer esto con las herramientas que hemos visto hasta ahora, en forma muy elemental. Pero a medida que el curso avance será cada vez mayor el volumen de datos a ordenar, y para esos momentos deberán estudiarse algoritmos algo más sofisticados y emplear otras herramientas disponibles en un lenguaje de programación.

Para formalizar el trabajo, enunciamos el problema a modo de ejercicio:

**Problema 14.)** *Se cargan por teclado tres números. Se pide mostrarlos en pantalla, ordenados de menor a mayor.*

**Discusión y solución:** Comenzaremos suponiendo que tenemos que ordenar un conjunto de sólo dos números, que están contenidos en sendas variables *a* y *b* que se cargarán por teclado. Este caso es trivial, y puede resolverse con sólo una condición (como ya vimos en la *Ficha 4, problema 8*). Se puede plantear un esquema que proceda a ordenar dos números almacenados originalmente en las variables *a* y *b* guardándolos ordenados en otras dos variables que llamaremos *may* y *men*. La idea final es que si queremos ver esos dos números ordenados, apliquemos este proceso y luego mostremos los valores de *men* y *may* (en ese orden). En pseudocódigo el proceso podría quedar así (sin numeración de líneas para abreviar):

```
ordenar_dos_numeros:
    si a > b:
        may = a
        men = b
    sino:
        may = b
        men = a
```

Si ahora queremos pasar a ordenar los *tres* números que nos dieron, almacenados en tres variables *a*, *b* y *c*, podemos mantener la idea que ya hemos usado: intentaremos reasignar esos tres números en otras tres variables llamadas *men*, *med* y *may*, de forma que la primera (*men*) siempre termine conteniendo al menor de los tres originales, *med* al valor mediano, y *may* al mayor.

Una primera idea que siempre suele surgir, consiste en usar tantas condiciones como combinaciones posibles existan entre los números originales. En nuestro caso, siendo tres las variables de entrada, las combinaciones posibles son 6 y eso nos llevaría a tener que plantear 6 condiciones (si usamos condiciones simples), como se ve en el esquema siguiente (en el que para simplificar, asumimos que las tres variables *a*, *b* y *c* tienen **valores diferentes**):

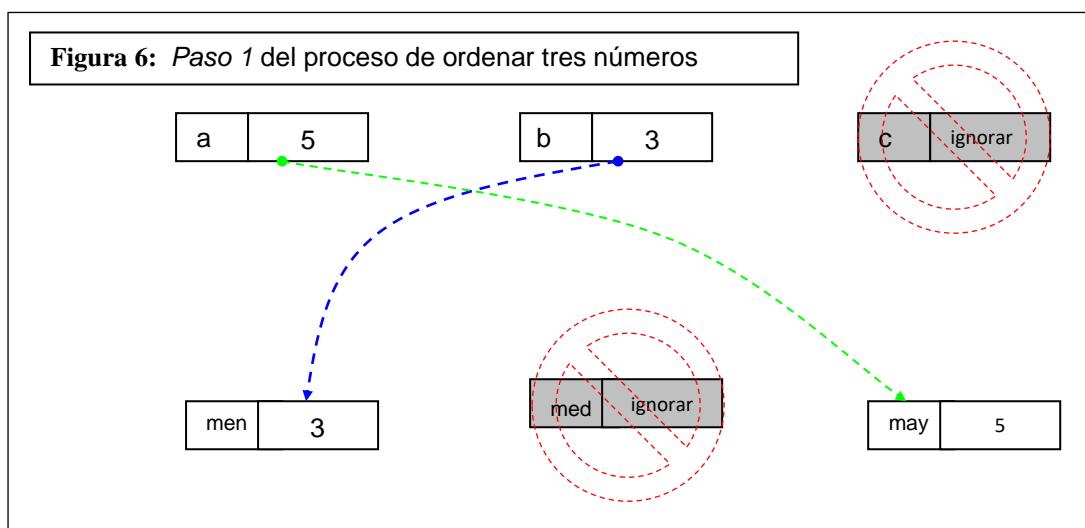
```
# Pseudocódigo - ordenamiento de tres números: versión 1...
ordenar_tres_numeros:
    si a > b > c:
        may, med, men = a, b, c
    si a > c > b:
        may, med, men = a, c, b
    si b > a > c:
        may, med, men = b, a, c
    si b > c > a:
        may, med, men = b, c, a
    si c > a > b:
        may, med, men = c, a, b
    si c > b > a:
        may, med, men = c, b, a
```

Aun cuando la solución obtenida parece buena, un programador debería acostumbrarse a desconfiar de su "primer impulso" cuando busca un algoritmo, pues ese primer impulso suele terminar en una solución intuitivamente obvia, pero ineficiente o poco práctica en términos de demora en el tiempo de ejecución o en la cantidad de líneas de código que deberá escribir para implementarla. Estas soluciones en las que el programador explora todas las posibles variantes y combinaciones posibles de resultados, se suelen designar como estrategias de "*fuerza bruta*".

Veamos el caso: hemos encontrado una solución simple, pero nos ha llevado a escribir *seis* condiciones para plantearla. Podemos intuir que si seguimos este camino, nos irá cada vez peor a medida que el número de datos aumente. Por caso, si queremos ordenar 4 números dispuestos en 4 variables de entrada, y seguimos la misma idea, requeriremos 24 condiciones (que es igual al número de combinaciones posibles entre 4 valores...) y eso ya parece demasiado trabajo...

Nuestro objetivo es encontrar un algoritmo que haga el ordenamiento de tres números, pero con la *menor cantidad posible de instrucciones condicionales*. Además, queremos que este algoritmo nos abra la mente, para repetir la misma la idea básica cuando el número de datos suba a 4 (por ejemplo) y también queramos la mínima cantidad de condiciones.

Una forma de hacerlo consiste en aplicar una variante simplificada de un algoritmo más amplio llamado *ordenamiento por inserción*. La idea es comenzar con una *reducción del volumen de datos del problema*: en vez de considerar los tres números *a*, *b* y *c* originales, suponemos que nos han dado sólo dos (*a* y *b*) e intentamos ordenarlos llevándolos a las variables *men* y *may* (por ahora, nos olvidamos de *c* y de la variable *med*):



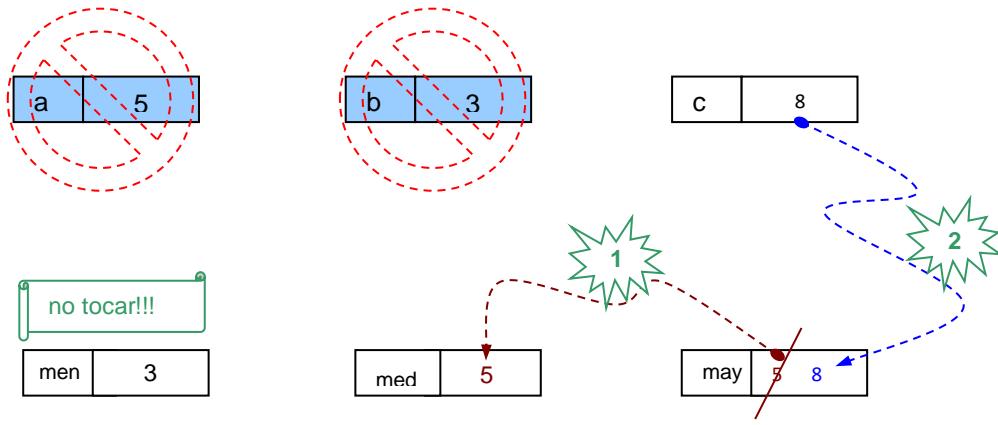
Está claro que este primer ordenamiento de dos números puede hacerse con nuestro ya conocido proceso *ordenar\_dos\_numeros* (que ordena los dos valores *a* y *b* y usa un sola condición...) No importa si el menor estaba en *a* o en *b*, finalmente ese proceso lo asignará en *men*. Y algo similar ocurrirá con el mayor y la variable *may*. Una vez ordenados esos dos primeros números, de aquí en más podemos olvidarnos de las variables *a* y *b*, y seguir trabajando con *men* y *may* (que contienen los mismos valores que *a* y *b*, pero ahora tenemos más información: sabemos cuál de las dos contiene al menor y cuál al mayor.)

El siguiente paso (ahora sí) es abrir el juego y considerar la variable *c* y la variable *med*. Es obvio que tenemos aquí tres posibilidades:

- ✓ El valor de *c* podría ser *mayor que el que tenemos guardado en may*. Sería el caso de la *Figura 6* si *c* valiese 8. Esto significaría que el valor que teníamos como el mayor, es en realidad el valor mediano. Reasignamos primero el valor de *may* en la variable *med*, y luego guardamos *c* en *may*. No tocamos el valor que teníamos en *men* (ya que definitivamente era el menor) (ver *Figura 7*).

Figura 7: Paso 2 del proceso, si  $c > may$ 

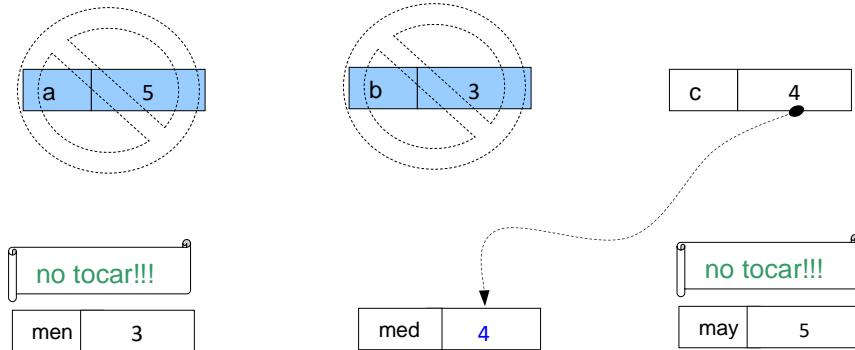
```
if c > may:
    med = may # 1
    may = c # 2
    # no tocar men!!!
```



- ✓ El valor de  $c$  podría ser menor que el que tenemos guardado en  $may$ , pero mayor al que tenemos en  $men$ . Sería el caso del gráfico de la *Figura 6* si  $c$  valiese 4. Y esto significaría que el valor que tenemos en  $c$  es directamente el mediano: sólo tendríamos que asignar  $c$  en  $med$ , y dejar  $men$  y  $may$  como estaban (ver *Figura 8*).

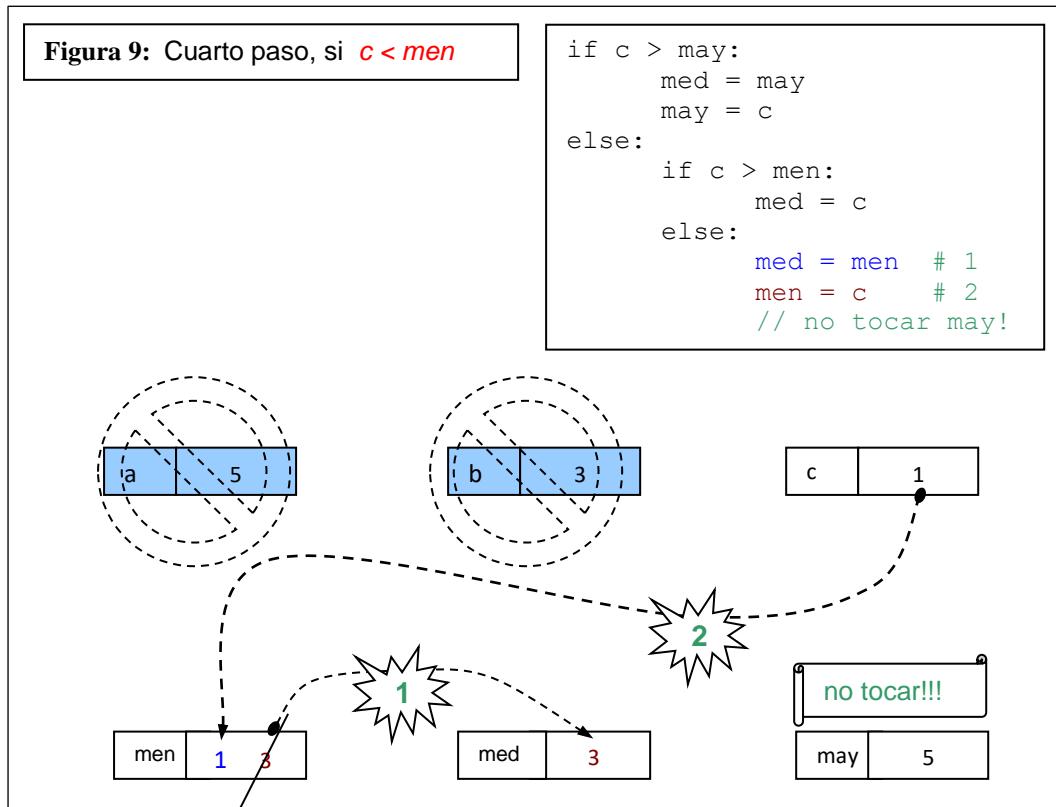
Figura 8: Tercer paso, si  $men < c < may$ 

```
if c > may:
    med = may
    may = c
else:
    if c > men:
        med = c
```



- ✓ Finalmente, el valor de  $c$  podría ser menor que el que tenemos guardado en  $men$ . Sería el caso del gráfico de la *Figura 6* si  $c$  valiese 1. Esto significaría que el valor que teníamos como el menor, es en realidad el valor mediano. Reasignamos primero el valor de  $men$  en la

variable *med*, y luego guardamos *c* en *men*. No tocamos el valor que teníamos en *may* (ya que definitivamente era el mayor) (ver Figura 9 ).



El algoritmo descripto es una simplificación para tres variables de un algoritmo más general que se conoce como *ordenamiento por inserción*, porque una vez ordenados los dos primeros números en *men* y *may*, se toma el valor *c* y se lo *inserta* en el lugar correcto (delante de *may*, entre *may* y *men* o detrás de *men*, según sea el caso) siguiendo los mecanismos sugeridos.

Después de todo este análisis podemos plantear un programa completo que cargue por teclado los tres números *a*, *b*, *c* y los ordene por inserción. Primero se procede a aplicar el proceso *ordenar\_dos\_numeros* para el primer paso de ordenar *a* y *b* entre *men* y *may*; y luego se aplican los pasos que hemos visto más arriba para terminar el proceso insertando *c* entre *men* y *may*. Es fácil ver que la cantidad total de condiciones empleadas en todo el proceso es de sólo tres: una en el proceso *ordenar\_dos\_numeros* y otras dos en el proceso que ordenar los tres, con lo que logramos una sustancial mejora respecto de las seis condiciones de nuestro primer intento...<sup>1</sup>

<sup>1</sup> Y si se trata de resolver problemas ahorrando recursos tanto como sea posible, no podemos dejar de recomendar la película *El Marciano* (*The Martian*) estrenada en 2015 y candidata al Oscar. Fue dirigida por Ridley Scott (el mismo que dirigió *Blade Runner*) y protagonizada por Matt Damon. Está basada en el reciente best seller del escritor Andy Weir. Es la historia de un integrante de una hipotética misión a Marte, que por accidente es dado por muerto y abandonado en el Planeta Rojo. El pobre astronauta deberá resolver a partir de allí una enorme cantidad de problemas para sobrevivir hasta su eventual rescate, con un buen humor infalible y atacando "de a un problema por vez". Imperdible la película. Pero mucho mejor el libro.

```
__author__ = 'Cátedra de AED'

# Título general y carga de datos...
print('Problema del ordenamiento de tres numeros')
a = int(input('a: '))
b = int(input('b: '))
c = int(input('c: '))

# Primero: ordenar los dos primeros numeros...
if a > b:
    men, may = b, a
else:
    men, may = a, b

# Segundo: ordenar los tres numeros...
if c > may:
    med, may = may, c
else:
    if c > men:
        med = c
    else:
        med, men = men, c

# Visualización de los resultados ordenados...
print('Menor:', men)
print('Medio:', med)
print('Mayor:', may)
```

---

## Bibliografía

- [1] Python Software Foundation, "Python Documentation", 2020. [Online]. Available: <https://docs.python.org/3/>.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [3] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

# Ficha 6

## Estructuras Repetitivas: El Ciclo *while*

### 1.] Introducción.

Todos los problemas y ejercicios analizados hasta ahora tenían, en última instancia, un elemento en común: en todos los casos se trataba de problemas cuya solución básica y más elemental nunca implicó *repetir la ejecución de un bloque de instrucciones*. Los algoritmos diseñados para resolver esos problemas eran de *naturaleza no repetitiva*.

Sin embargo, basta observar con cuidado a nuestro alrededor para notar que la mayor parte de las tareas que realizamos a diario incluyen algún tipo de *repetición de acciones* en cierto momento: Quien hable por teléfono debe *repetir* la operación de marcar un dígito en el teclado telefónico (aunque no siempre necesariamente el mismo dígito), tantas veces como dígitos tenga en total el número con el que se quiere comunicar. Quien tenga la mala fortuna de pinchar un neumático mientras va manejando su automóvil, debe cambiar la goma pinchada, pero eso implica *repetir* la operación de desajustar una tuerca y retirarla, tantas veces como tuercas tenga esa goma. El sólo hecho de leer un libro implica *repetir* la acción de dar vuelta una página, tantas veces como páginas se alcance a leer. Un empleado administrativo que reciba la orden de llenar formularios mientras dure su turno de trabajo, está realizando una tediosa *tarea repetitiva*: toma un formulario de la pila de formularios en blanco, lo llena siguiendo ciertas pautas, y lo deja a un lado en otra pila de formularios ya completados. Luego repite la secuencia y así prosigue hasta que su turno termina.

Los ejemplos citados en el párrafo anterior son muestras de *situaciones repetitivas cotidianas*. Pero el hecho es que cuando se desarrollan programas para una computadora estas situaciones algorítmicas repetitivas son también muy comunes. De hecho, *lo más común* es enfrentar problemas cuya solución requiera alguna clase de repetición en la ejecución de ciertas instrucciones para llegar al resultado deseado, o al menos para llegar con mayor eficiencia a dicho resultado<sup>1</sup>.

En ese sentido, en programación un *ciclo* es una *instrucción compuesta* que permite la repetición controlada de la ejecución de cierto conjunto de instrucciones en un programa, determinando si la repetición debe detenerse o continuar de acuerdo al valor de cierta condición que se incluye en el ciclo [1]. Los ciclos también se designan como *estructuras repetitivas*, o bien como *instrucciones repetitivas*. Básicamente, en todo lenguaje de programación un ciclo consta de dos partes:

<sup>1</sup> Si de repetir acciones se trata, no podemos menos que referir la fantástica película *Groundhog Day* (en español conocida como *Hechizo del Tiempo*) del año 1993, dirigida por *Harold Ramis* y protagonizada por *Bill Murray* y *Andie MacDowell*. Un locutor de televisión engreído y cascarrabias fue condenado a revivir una y otra vez el mismo día que más odiaba, sin que lo sepa nadie aparte de él mismo, y sólo pudo escapar a ese castigo cuando finalmente cambió y se convirtió en una buena persona... después de miles y miles de repeticiones de la misma jornada y los mismos eventos. Impecable. Extraordinaria. Quizás lo mejor de *Bill Murray*.

- El *bloque de acciones o cuerpo del ciclo*, que es el conjunto de instrucciones cuya ejecución se pide repetir.
- La *cabecera del ciclo*, que incluye una *condición de control* y/o elementos adicionales en base a los que se determina si el ciclo continúa o se detiene.

Las instrucciones repetitivas o ciclos son implementados por los diversos lenguajes en formas que varían ligeramente de un lenguaje a otro. El lenguaje Python implementa dos tipos de ciclos mediante instrucciones específicas y muy flexibles, designándolos respectivamente como ciclo *for* y como ciclo *while* [2]. En esta Ficha 6 analizaremos la forma de uso y aplicación del ciclo *while*, y en la Ficha 7 veremos la forma de trabajo y aplicación del ciclo *for*.

## 2.] El ciclo **while** en Python.

Hemos sugerido que en muchas situaciones necesitaremos desarrollar programas que ejecuten en forma repetida un conjunto de instrucciones. Tomemos como ejemplo el mismo programa básico que hemos mostrado en la Ficha 5 para introducir el tema de las variables de conteo, en el que se cargaban tres números y se pedía determinar cuántos números negativos se ingresaron. El programa básico que se mostró en la Ficha 5 era el siguiente:

**Figura 1: Uso básico de un contador (tomado de la Ficha 5).**

```
__author__ = 'Catedra de AED'

a = 0
num = int(input('Ingrese un número: '))
if num < 0:
    a = a + 1

num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1

num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1

print('Cantidad de negativos cargados:', a)
```

Si bien el programa citado funciona perfectamente bien, encierra dos grandes limitaciones:

1. ¿Qué pasaría si en lugar de sólo *tres números* tuviéramos que cargar *mil números*? Resultaría muy poco práctico desarrollar un programa con mil instrucciones de lectura y mil instrucciones condicionales. ¿Y si en lugar de *mil números* fueran *diez mil*, o *cien mil*, o *un millón*?
2. ¿Qué pasaría además, *si no supiésemos la cantidad exacta de números que se deben cargar* al momento de desarrollar el programa? Por ejemplo, es común que el enunciado o requerimiento no exprese cuántos números vendrán y en cambio indique que *se carguen números hasta que en algún momento se ingrese el cero*. Intente resolver este problema **sólo** en base a estructuras secuenciales y condicionales y descubrirá que no es posible.

Las *instrucciones repetitivas* o *ciclos* están previstas para que el programador pueda resolver situaciones como estas con sencillez. Como dijimos, Python provee dos tipos de ciclos: el *ciclo while* y el *ciclo for*.

En muchas ocasiones necesitamos plantear un ciclo que ejecute en forma repetida un bloque de acciones pero sin conocer previamente la cantidad de vueltas a realizar. Para estos casos la mayoría de los lenguajes de programación, y en particular Python, proveen un ciclo designado como *ciclo while*, aunque como veremos, puede ser aplicado sin ningún problema también en situaciones en las que se conoce la cantidad de repeticiones a realizar.

Como todo ciclo, un *ciclo while* está formado por una *cabecera* y un *bloque o cuerpo de acciones*, y trabaja en forma general de una manera muy simple. La *cabecera* del ciclo contiene una *expresión lógica* que es evaluada en la misma forma en que lo hace una instrucción condicional *if*, pero con la diferencia que el *ciclo while* ejecuta su bloque de acciones en *forma repetida* siempre que la expresión lógica arroje un valor verdadero. Así como un *if* hace una *única* evaluación de la expresión lógica para saber si es verdadera o falsa, un *ciclo while* realiza *múltiples* evaluaciones: cada vez que termina de ejecutar el bloque de acciones vuelve a evaluar la expresión lógica y si nuevamente obtiene un valor verdadero repite la ejecución del bloque y así continúa hasta que se obtenga un falso [3].

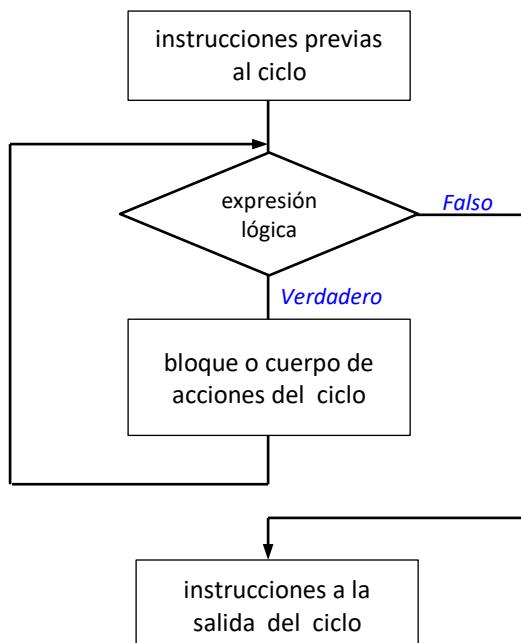
En Python el *ciclo while* se escribe básicamente en la forma siguiente:

```
while expresión_lógica:  
    bloque de acciones
```

El programador no debe escribir ninguna instrucción especial dentro del bloque de acciones para indicar que el ciclo debe volver a la cabecera y evaluar nuevamente la expresión lógica: ese retorno es automático e implícito.

La lógica esencial de funcionamiento de un *ciclo while* se refleja muy bien en la *forma clásica* de *graficarlo* en un diagrama de flujo (ver Figura 2). La expresión lógica conforma la cabecera del ciclo, y su valor determina si el ciclo continúa ejecutando el bloque de acciones, o se detiene. En la primera oportunidad que esa expresión lógica sea falsa, el ciclo se detendrá y el programa continuará ejecutando las instrucciones que figuren a la salida del ciclo. Pero si la expresión lógica es verdadera, ejecutará el bloque y luego automáticamente volverá a chequear la expresión lógica de la cabecera, repitiendo el esquema.

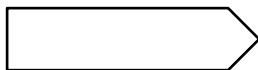
**Figura 2: Diagrama de flujo de un ciclo while (diagramación clásica)**



Note especialmente que si al evaluar la expresión lógica *por primera vez* esta fuese falsa, entonces el bloque de acciones no sería ejecutado y el programa continuaría con las instrucciones a la salida del ciclo. Debido a este comportamiento, se dice que el **ciclo while** es un ciclo de tipo  $[0, N]$ : el bloque de acciones puede llegar a ejecutarse entre 0 y  $N$  veces, siendo  $N$  un número entero positivo normalmente desconocido (pero que indica un número finito de repeticiones) [1].

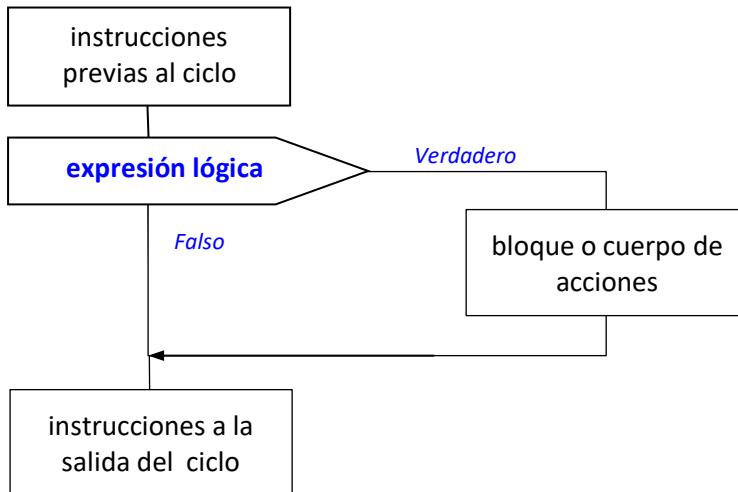
Si bien la forma clásica de graficar un **ciclo while** (en rigor, un ciclo  $[0, N]$ ) es muy clara en cuanto a su lógica, se puede plantear alternativamente un gráfico basado en otro símbolo no estándar pero específico para el planteo de ciclos, como es el siguiente:

**Figura 3: Símbolo alternativo (no estándar) para graficar un ciclo en un diagrama de flujo.**



El diagrama completo alternativo [1] sería el que se muestra en la *Figura 4*. La *forma alternativa* (o *indentada*) de diagramar el ciclo no es tan clara ni tan directa en cuanto a la lógica, pero mantiene coherencia con el tipo de símbolo usado para graficar un ciclo (distinguiéndolo sin ambigüedades del rombo usado para graficar una instrucción condicional) y mantiene visualmente el esquema de indentación que luego será usado al escribir el programa. *Pero queda claro que tanto los estudiantes como los profesores pueden optar por la forma que deseen*: recuerde que el diagrama de flujo es una *técnica auxiliar* de representación de algoritmos. Si en el contexto en que se esté trabajando se necesitase mantener consistencia en cuanto al tipo de diagrama a usar, entonces los miembros de ese equipo simplemente deberán acordar a qué reglas atenerse y qué tipo de esquema gráfico aplicar.

**Figura 4: Diagrama de flujo de un ciclo while (diagramación alternativa o indentada)**



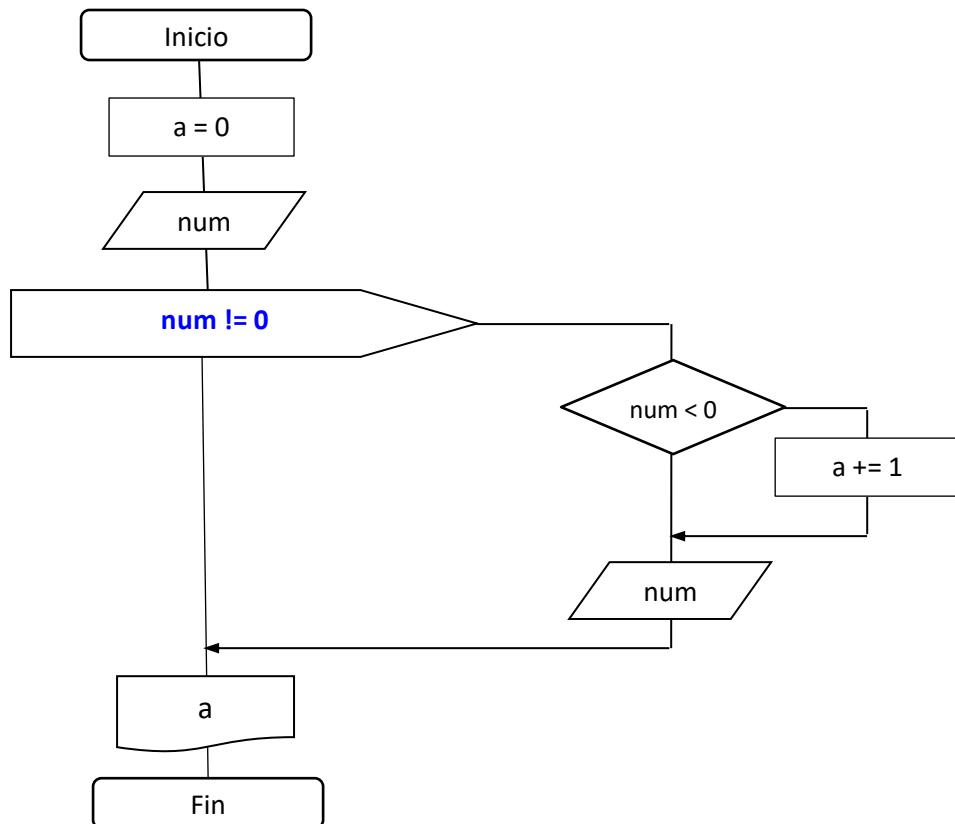
Volviendo al problema de cargar un conjunto de números y determinar cuántos eran negativos, supongamos ahora que desconocemos cuántos serán los números a cargar, pero que en su lugar sabemos que el usuario *ingresará un cero cuando ya no le queden números por ingresar*. Este planteo (como veremos) sería imposible de resolver (al menos en forma

simple y directa) mediante un *ciclo for* en Python<sup>2</sup> porque se desconoce la cantidad de repeticiones que debería hacer ese ciclo.

En un caso como este, el *ciclo while* se aplica con naturalidad. Si en el enunciado del problema no se indica la cantidad de datos a procesar o la cantidad de repeticiones a realizar, pero en cambio se indica alguna condición que especifique cómo debe cortar el ciclo, entonces esa condición se designa en forma general como *condición de corte*, y en base a ella debe duducirse y escribirse la cabecera del *ciclo while* para controlar su avance.

El programador sólo debe recordar que un *ciclo while* en Python, repite la ejecución del bloque si la expresión lógica de su cabecera es verdadera, y corta si es falsa. Por lo tanto, para plantear la cabecera del ciclo debe pensar en la siguiente pregunta general: *¿qué es lo que debe ser cierto para que el ciclo continúe?* Y la expresión lógica que obtenga es la que debe usar en la cabecera del ciclo. Por lo tanto, si se indica que la carga de datos terminará cuando se ingrese un cero, entonces *lo que debe ser cierto para que el ciclo continúe* es que el número ingresado **no** sea cero. La Figura 5 muestra el diagrama de flujo propuesto.

Figura 5: Diagrama de flujo (forma indentada) del problema de contar negativos usando doble lectura.



En base a este diagrama mostramos ahora el programa completo para el conteo de los números negativos, replanteado para usar un ciclo *while*, sin conocer cuántos números vendrán pero sabiendo que al ingresar un 0 la carga debe terminar:

<sup>2</sup> Y algo similar ocurriría en otros lenguajes en los que el *ciclo for* está diseñado para desplegar una cantidad exacta de repeticiones, como en el lenguaje *Pascal*. Sin embargo, note que en otros lenguajes (como *C*, *C++* o *Java*) la instrucción *for* se implementa en forma tan amplia, que no hay problema en aplicarla en este tipo de situaciones y en cualquier otra que requiera simplemente un ciclo.

```

__author__ = 'Catedra de AED'

a = 0
num = int(input('Ingrese un numero (o cero para terminar): '))
while num != 0:
    if num < 0:
        a += 1
    num = int(input('Ingrese otro numero (o cero para terminar): '))

print('Cantidad de negativos cargados:', a)

```

El programa utiliza un *ciclo while* para la carga y procesamiento de los números. Por otra parte, puede notarse que se carga un valor de *num* antes de comenzar el ciclo, y otro valor de *num* inmediatamente antes de finalizar el bloque de acciones. La primera lectura se realiza porque de otro modo la variable *num* podría no existir o podría tener un valor residual cuando se verifique la expresión lógica de control del *ciclo while* por primera vez, y podría esa verificación dar resultados no deseados. Cuando en el ciclo se pregunta si *num* es diferente de cero, *num debe existir y tener siempre un valor controlado por el programador*. La segunda lectura, efectuada dentro del bloque de acciones al final del mismo, se efectúa porque en caso de no hacerla la variable *num* seguiría teniendo el *mismo valor* que se cargó en la primera lectura, y eso provocaría que la condición de control se vuelva a evaluar en verdadero, comenzando una serie infinita de repeticiones para el mismo valor.

En un ciclo bien planteado, el valor de la variable o variables que se usan en la expresión lógica de control debe cambiar adecuadamente dentro del bloque de acciones, pues de otro modo, si el valor nunca cambia, el ciclo repetirá una y otra vez, entrando en una situación de *ciclo infinito*. El esquema anterior, que incluye dos lecturas para evitar los problemas descriptos, es de uso muy común cuando se procesan conjuntos de valores usando un *ciclo while*, sin conocer cuántos valores vendrán, y se designa como *proceso de carga por doble lectura* [1].

Debe el programador tener especial cuidado cuando usa ciclos, para evitar caer en *ciclos infinitos*. Si en el ejemplo anterior se elimina la segunda lectura, se cae irremisiblemente en un ciclo infinito. Por otra parte, el ciclo infinito podría provocarse en situaciones donde no haya necesariamente lectura por teclado de la variable de control. Por ejemplo, el siguiente ciclo *está mal planteado*, debido a que la variable *c* empieza valiendo uno (por asignación), pero nunca cambia luego de valor:

```

c = 1
while c != 0:
    x = int(input('Ingrese un valor: '))

```

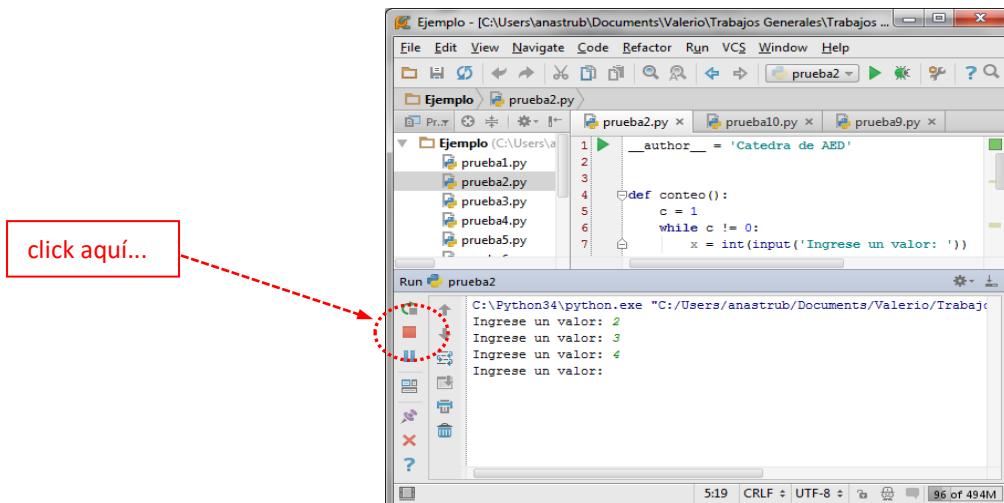
En este caso se produce un error muy común: se usa una variable en la condición de control, pero dentro del bloque de acciones del ciclo se cambia el valor de *otra (u otras)* variable/s. Este segmento de programa provocará que aparezca en pantalla un mensaje solicitando cargar un valor, se cargará ese valor, y luego se pedirá otro, y luego otro más, y así sucesivamente, sin terminar nunca el programa.

¿Qué hacer si un programa entra en un ciclo infinito? La manera de interrumpir un programa (ya sea porque por alguna razón no pueda finalizar normalmente o solo porque el programador desea interrumpirlo), suele consistir en una combinación de teclas que fuerza al programa a finalizar. En nuestro caso, si el programa se ejecuta desde *PyCharm*, la

combinación de teclas a usar es: **<Ctrl> + <F2>**. Es decir, se presiona la tecla *Control* (ubicada normalmente a los lados de la barra espaciadora del teclado), y *sin soltarla* se presiona *también* la tecla *F2* (la tecla de *Función 2*, por lo general ubicada en la primera fila de arriba del teclado). Esta operación suele funcionar sin problemas en *PyCharm*, cancelando el programa. Por cierto, es siempre una buena idea *grabar* los programas *antes* de intentar ejecutarlos...

En algunas ocasiones si el programa entra en ciclo infinito o pareciera estar *clavado* (aparentemente funcionando pero sin respuesta alguna al usuario) y nada funciona para detenerlo, no queda otro remedio que intentar interrumpirlo desde el sistema operativo (usando el *administrador de tareas* o similar) o incluso apagar el equipo y volverlo a encender, pero esta operación implica que si el programa no fue grabado, se perderá todo lo que el programador haya escrito. En *PyCharm* también se puede intentar interrumpir el programa usando el mouse: a la izquierda del panel de salida de *PyCharm*, apunte con el mouse al pequeño ícono en forma de *cuadrado de color rojo*, y haga *click* sobre él. Esto equivale a pulsar la combinación **<Ctrl> + <F2>** y el programa en curso se interrumpirá:

**Figura 6: Interrupción de un programa en PyCharm.**



Por supuesto, podemos utilizar un **ciclo while** para resolver problemas donde **sí** se conoce de antemano la cantidad de iteraciones a realizar, tal como lo hubiéramos hecho con un **ciclo for**, aunque al costo de un poco más de trabajo ya que, como veremos, el **ciclo for** se plantea de forma más automática mientras que en el **ciclo while** tendremos que hacer en forma manual el conteo de vueltas.

Por ejemplo, el siguiente script Python usa un **ciclo while** para mostrar los primeros diez múltiplos de un número *x* que se carga por teclado:

```
x = int(input("Ingrese un número: "))
i = 1
while i <= 10:
    print(x, " * ", i, " = ", x*i)
    i += 1
```

En este caso, la variable *i* es un contador que sirve para contar la cantidad de vueltas del ciclo y a la vez para realizar el cálculo de cada multiplicación. Antes de que comience el ciclo se le da el valor 1 con lo que la expresión lógica de control del ciclo entrega un valor verdadero y se ejecuta por primera vez el bloque de acciones del ciclo. En la primera

instrucción de ese bloque se calcula la multiplicación y se muestra el resultado y en la segunda la variable *i* se incrementa en una unidad justo antes de que la expresión lógica de control del ciclo vuelva a evaluarse y esto es lo que permite contar las repeticiones del ciclo. Cuando el valor del contador *i* llegue a valer 11, la expresión lógica de control del ciclo será falsa y el ciclo se detendrá.

Como ya se mencionó antes, el *ciclo while* es lo suficientemente versátil como para poder aplicarse en cualquier situación que requiera repetir la ejecución de un bloque de acciones, aunque el *ciclo for* que veremos se adapte mejor a situaciones en las que se conoce la cantidad de repeticiones a realizar. El programador decidirá cuál de ambas herramientas es la que mejor le resulte según el problema que esté analizando.

A modo de ejemplo de aplicación, consideremos el siguiente problema sencillo (que además servirá para volver a aplicar el concepto de *variable centinela* o *variable bandera*):

**Problema 15.)** *Cargar por teclado una lista de números enteros que irán llegando uno a uno, y que representan cantidades mensuales de automóviles nuevos vendidos en el país durante cierto período. Para indicar que la carga de datos debe finalizar, se ingresará el valor -1 (menos uno) (note que el valor 0 (cero) puede ser un dato valido: es perfectamente posible que no haya habido ventas en un mes determinado). Se pide:*

- a. *Determinar cuántas de esas cantidades fueron mayores o iguales que 0 pero menores que 10000 unidades, cuántas fueron mayores o iguales a 10000 pero menores que 15000, y cuántas fueron mayores o iguales que 15000.*
- b. *Determinar la cantidad total de automóviles nuevos que se vendieron en el país.*
- c. *Determinar si se ingresó al menos una cantidad igual a 0 o no. Informe con un mensaje simple en pantalla.*

**Discusión y solución:** Este es un problema típico de procesamiento de una sucesión de números enteros con fines estadísticos. Como no sabemos cuántos números entrarán como dato, pero sabemos que al cargar un -1 se debe terminar el proceso, entonces podemos usar un *ciclo while* y un esquema de *carga por doble lectura*.

El punto a) solicita contar cuántas cantidades hay en cada uno de tres posibles intervalos de valores, y para eso necesitaremos tres contadores (que llamaremos *c1*, *c2* y *c3*).

El punto b) en última instancia pide *acumular* todos los números cargados y por eso en cada vuelta usaremos un acumulador (que llamaremos *t*) para ir sumando todos los números que se ingresen.

Y para cumplir con el punto c), usaremos una bandera (que llamaremos *ok*) que nos permitirá dejar marcado si se cargó al menos una cantidad igual a 0.

El programa completo se muestra a continuación, y luego del mismo haremos algunas observaciones:

```
__author__ = 'Catedra de Algoritmos y Estructuras de Datos'

# inicialización de contadores, acumuladores y banderas...
c1, c2, c3, t = 0, 0, 0, 0
ok = False

# proceso de carga por doble lectura...
# ... hacer la primera lectura...
cant = int(input('Ingrese cantidad vendida (con -1 termina el proceso): '))
```

```

while cant != -1:

    # punto a): chequear en qué intervalo está cada cantidad y contar...
    if 0 <= cant < 10000:
        c1 += 1
    elif 10000 <= cant < 15000:
        c2 += 1
    elif cant >= 15000:
        c3 += 1

    # punto b): acumular cada cantidad...
    t += cant

    # punto c): chequear si cant es 0, y marcar con un flag...
    if cant == 0:
        ok = True

    # hacer la segunda lectura...
    cant = int(input('Ingrese otra cantidad (con -1 termina): '))

# visualización de resultados... punto a)
print()
print('Cantidad de valores >= 0 pero < 10000:', c1)
print('Cantidad de valores >= 10000 pero < 15000:', c2)
print('Cantidad de valores >= 15000:', c3)

# visualización de resultados... punto b)
print('Cantidad total de vehículos vendidos:', t)

# visualización de resultados... punto c)

# recuerde que lo que sigue es equivalente a if ok == True:
if ok:
    print('Se registró al menos una cantidad de ventas igual cero')
else:
    print('No se registró ninguna cantidad de ventas igual cero')

```

La lógica general es bastante directa: a través del ciclo *while* se procede a cargar por teclado los valores de las cantidades vendidas usando para ello la variable *cant* y un mecanismo de *doble lectura*. El ciclo controla que el número ingresado en *cant* sea diferente de -1 (que es el valor de corte). Dentro del ciclo, cada valor de *cant* se controla **con un esquema de condiciones anidadas** para saber en qué intervalo se encuentra ese valor y así poder contarla en alguno de los tres contadores *c1*, *c2* o *c3*. Inmediatamente luego el valor de *cant* simplemente se suma en el acumulador *t* para ir obteniendo así el total general de vehículos vendidos:

```

# inicialización de contadores, acumuladores y banderas...
c1, c2, c3, t = 0, 0, 0, 0
ok = False

# proceso de carga por doble lectura...
# ... hacer la primera lectura...
cant = int(input('Ingrese cantidad vendida (con -1 termina el proceso): '))
while cant != -1:

    # punto a): chequear en qué intervalo está cada cantidad y contar...
    if 0 <= cant < 10000:
        c1 += 1
    elif 10000 <= cant < 15000:

```

```

c2 += 1
elif cant >= 15000:
    c3 += 1

# punto b): acumular cada cantidad...
t += cant

```

Para determinar si alguno de los valores cargado en *cant* en alguna de las vueltas del ciclo fue igual a cero, se usa una bandera que hemos llamado *ok*. Antes del ciclo, esa bandera se inicializa con el valor *False*, para indicar que hasta ese momento no se ha detectado la carga de un valor igual a 0 en *cant*. Si en alguna iteración del ciclo se detecta la carga de un 0 en *cant*, el valor del flag *ok* se vuelve a *True* y se lo deja en ese estado hasta el final:

```

ok = False

# proceso de carga por doble lectura...
# ... hacer la primera lectura...
cant = int(input('Ingrese cantidad vendida (con -1 termina el proceso): '))
while cant != -1:

    # ...

    # punto c): chequear si cant es 0, y marcar con un flag...
    if cant == 0:
        ok = True

    # hacer la segunda lectura...
    cant = int(input('Ingrese otra cantidad (con -1 termina): '))

```

Al finalizar el ciclo (por la carga del valor -1) se muestran los valores de los tres contadores y el del acumulador (note la llamada a *print()* sin pasarle parámetros, que se hace para simplemente dejar una línea en blanco en la salida):

```

# visualización de resultados... punto a)
print()
print('Cantidad de valores >= 0 pero < 10000:', c1)
print('Cantidad de valores >= 10000 pero < 15000:', c2)
print('Cantidad de valores >= 15000:', c3)

# visualización de resultados... punto b)
print('Cantidad total de vehículos vendidos:', t)

```

Finalmente, para se controla el valor final del flag *ok*: si ese valor es *True*, significa que al menos una vez se cargó el valor 0 en la variable *cant* durante el ciclo. Pero si el valor de *ok* es *False* significa que *ok* mantuvo el valor inicial que se le asignó al comenzar el programa y por lo tanto, nunca se cargó un cero:

```

# visualización de resultados... punto c)
if ok:
    print('Se registró al menos una cantidad de ventas igual cero')
else:
    print('No se registró ninguna cantidad de ventas igual cero')

```

Veamos ahora otro ejemplo de aplicación en el que también se procesa una secuencia números, pero algo más complejo:

**Problema 16.)** *Cargar por teclado un conjunto de números enteros, uno a uno. La carga sólo debe terminar cuando se ingrese el cero. Determine si los números que se ingresaron eran todos positivos o todos negativos (sin importar en qué orden hayan entrado). Por ejemplo, la secuencia {8, 4, 3, 7} cumple con la consigna indicada (son todos positivos). La secuencia {-2, -1, -5} también cumple (son todos negativos), pero esta otra secuencia {10, -8, 2, 12} no cumple (hay números de distinto signo entremezclados). Si todos los números eran efectivamente del mismo signo, muestre también la suma de esos números (no mostrar la suma si había números de signos diferentes entremezclados).*

**Discusión y solución:** Este es un problema típico de procesamiento de una sucesión de números enteros. Como no sabemos cuántos números entrarán como dato, pero sabemos que al cargar un 0 se debe terminar el proceso, entonces podemos usar un *ciclo while* y un esquema de carga por doble lectura. Y como además finalmente se pide acumular los números cargados, en cada vuelta usaremos un acumulador para ir sumando los números que se ingresen (por ahora no nos preocuparemos del hecho de que esos números podrían ser de signos opuestos).

Si llamamos *actual* a la variable de carga y *suma* al acumulador, el ciclo podría comenzar planteándose así:

```
# acumulador para los números del mismo signo...
suma = 0

# cargar el primero...
actual = int(input('Cargue un número entero (con 0 corta): '))
while actual != 0:

    # acumular el número ingresado
    suma += actual

    # cargar el siguiente...
    actual = int(input('Cargue otro (con 0 corta): '))
```

Para controlar si los números son o no del mismo signo, podemos aplicar una técnica muy común entre los programadores para dejar algún tipo de señal o aviso lógico, que le indique al programador si ha ocurrido o no algún suceso en particular.

Procederemos para eso de la siguiente manera: sea la variable booleana *ok* que comenzaremos asignando con el valor *True* antes de comenzar el ciclo de carga. La idea es que el valor de esa variable indique lo que hasta ese momento se sabe respecto de los signos de los números cargados: usamos el valor *True* para registrar que hasta ese momento, todos los números que se hayan visto eran efectivamente del mismo signo (lo cual es efectivamente así si sólo se cargó uno). Si a lo largo del ciclo, se descubre que los números cargados eran de signos diferentes, en ese momento el valor de *ok* se cambia a *False* y se avisa con eso que se ha descubierto una anomalía. Cuando el ciclo termina (al cargarse el 0), el valor final de *ok* nos dice si los números cargados eran del mismo signo (*ok = True*) o no (*ok = False*). El esquema (casi en pseudocódigo...) podría quedar modificado así:

```
# si ok es True, todos son del mismo signo hasta aquí...
ok = True

# acumulador para los números cargados...
suma = 0

# carga del primer número...
actual = int(input('Cargue un número entero (con 0 corta): '))
```

```

while actual != 0:

    # la pregunta que sigue está en pseudocódigo por ahora...
    si hubo signos diferentes:
        ok = False

    # acumular los números, sin importar si son del mismo signo...
    suma += actual

    # cargar el siguiente...
    actual = int(input('Cargue otro (con 0 corta): '))

    # si al cortar el ciclo, ok es True
    # entonces la secuencia estaba bien...
    if ok:
        print('Todos los números eran del mismo signo')
        print('La suma de esos números es:', suma)
    else:
        print('Los números no eran todos del mismo signo')

```

El acumulador *suma* sólo se muestra si al finalizar el ciclo se comprueba que todos los números eran del mismo signo. De otro modo, su valor simplemente se ignora.

Note que una vez que *ok* cambia a *False*, no vuelve ya a cambiar a *True*: si cambió a *False* al menos una vez, significa que al menos una vez se detectaron números con signos cambiados y esa anomalía es definitiva: no importa como siga la carga de la sucesión, el valor de *ok* seguirá en *False*.

Para determinar si en algún momento hay un cambio de signo, la idea sería poder contar en cada vuelta con dos números en lugar de sólo uno. Por caso, sabemos que la variable *actual* contiene el número que se acaba de cargar en esa iteración, pero podríamos usar una segunda variable llamada *anterior* para guardar en ella el valor del número que se cargó en la vuelta anterior. Eso es fácil de hacer: al cargar el primer número, se asigna ese mismo en *anterior* (en la primera vuelta y sólo en ella las variables *actual* y *anterior* contendrán el mismo número) y a partir de allí se almacena en *anterior* el valor de *actual* inmediatamente antes de cargar el siguiente valor en *actual*:

```

# si ok es True, todos son del mismo signo hasta aquí...
ok = True

# acumulador para los números cargados...
suma = 0

# carga del primer número...
actual = int(input('Cargue un número entero (con 0 corta): '))

# inicializar anterior con el mismo primer número...
anterior = actual

while actual != 0:

    # la pregunta que sigue está en pseudocódigo por ahora...
    si hubo signos diferentes:
        ok = False

    # acumular los números, sin importar si son del mismo signo...
    suma += actual

    # guardar el actual para ser el anterior en la vuelta siguiente...

```

```

anterior = actual

# cargar el siguiente...
actual = int(input('Cargue otro (con 0 corta): '))

# si al cortar el ciclo, ok es True
# entonces la secuencia estaba bien...
if ok:
    print('Todos los números eran del mismo signo')
    print('La suma de esos números es:', suma)
else:
    print('Los números no eran todos del mismo signo')

```

Y como ahora tenemos el valor *actual* y el de la vuelta *anterior*, sólo debemos controlar si ambos son del mismo signo o no. Si *no* fuesen del mismo signo, tendríamos detectada una anomalía y *ok* cambiaría a *False*. La forma más sencilla se determinar si dos números son del mismo signo o no (sabiendo que ninguno de ellos es cero) es multiplicarlos entre sí: si el resultado es positivo, los números eran del mismo signo (los dos positivos o los dos negativos); pero si el resultado es negativo, entonces ambos eran de signos opuestos. El esquema puede entonces replantearse así, en forma de programa definitivo:

```

__author__ = 'Catedra de AED'

print('Procesamiento de números del mismo signo...')

# si ok es True, todos son del mismo signo hasta aquí...
ok = True

# acumulador para los números cargados...
suma = 0

# carga del primer número...
actual = int(input('Cargue un número entero (con 0 corta): '))

# variable para tener el anterior en cada vuelta...
anterior = actual
while actual != 0:

    # controlar los signos del numero actual y el anterior
    if actual * anterior < 0:
        # si el producto es < 0, eran de distinto signo...
        # cambiar el valor de ok...
        ok = False

    # acumular los números, sin importar si son del mismo signo...
    suma += actual

    # poner el actual como el anterior de la vuelta siguiente...
    anterior = actual

    # cargar el siguiente...
    actual = int(input('Cargue otro (con 0 corta): '))

# si al cortar el ciclo, ok es True
# entonces la secuencia estaba bien...
if ok:
    print('Todos los números eran del mismo signo')
    print('La suma de esos números es:', suma)

```

```

else:
    print('Los números no eran todos del mismo signo')

```

### 3.] Tratamiento de números complejos en Python.

En muchos problemas se requerirá trabajar con cálculos de raíces de distintos números, y sabemos que una raíz de orden par (raíz cuadrada, raíz cuarta, etc.) de un número negativo no está definida en el campo de los números reales (ya que no existe un *número real* tal que elevado al cuadrado dé como resultado un número negativo).

En álgebra y análisis matemático el campo de los *números complejos* (que surgió como una extensión del campo de los *números reales*), permite trabajar con raíces de orden par de números negativos introduciendo un elemento llamado la *unidad imaginaria*, cuyo valor se asume como igual a la *raíz cuadrada de (-1)* y se denota como *i* en matemática o también como *j* en el área de la ingeniería. Obviamente, el valor efectivo de la raíz cuadrada de (-1) no se calcula, sino que se deja expresado como  $j = \sqrt{-1}$  y luego se arrastra ese valor como parte de las operaciones algebraicas a realizar y así se puede continuar adelante con ciertos cálculos algebraicos y reducir algunas expresiones *llevándolas al campo complejo*. Por ejemplo, si se pide calcular la raíz cuadrada de (-4) (que no tiene solución en el campo real) se puede proceder así en el campo complejo:

$$\begin{aligned}
\sqrt{(-4)} &= \sqrt{(4 * (-1))} \\
&= \sqrt{4} * \sqrt{(-1)} \\
&= 2 * j
\end{aligned}$$

En general, un número complejo  $c$  es de la forma  $(a + bj)$  donde  $a$  es un número en coma flotante (designado como la *parte real* del complejo) y  $b$  es otro número en coma flotante (designado como la *parte imaginaria* del complejo). El factor  $j$  representa la unidad imaginaria  $\sqrt{-1}$  ya citada. Tanto la parte real como la imaginaria pueden valer 0, con lo cual se tendrían números complejos tales como  $(0 + 3j)$  o como  $(4 + 0j)$  y está claro que en estos casos, la parte igual a cero puede obviarse. En nuestro caso, el valor de  $\sqrt{(-4)}$  es el complejo  $(0 + 2j)$  que fue expresado como  $2j$ .

El hecho es que en muchos lenguajes de programación se producirá un error en tiempo de ejecución (el programa se interrumpirá) si se intenta calcular la raíz cuadrada (o cualquier raíz de orden par) de un número negativo. En otros lenguajes, el programa no se interrumpirá pero la variable donde se almacene el resultado quedará asignada con una constante que representa un valor indefinido (el cual no podrá usarse para cálculos posteriores).

Pero en Python *el resultado será efectivamente un número complejo*: Python provee un tipo especial de dato numérico llamado *complex* (que en realidad es lo que se conoce como una *clase*) que permite representar números complejos de la forma  $a + bj$  tal como se indicó más arriba [3]. El factor  $j$  representa la unidad imaginaria y es automáticamente gestionado por Python. Por lo tanto, cualquier expresión que pudiese dar como resultado un número complejo, en Python efectivamente funcionará y entregará el complejo como resultado, sin que el programador deba preocuparse por detalles de validación. En la siguiente secuencia de instrucciones, se está intentando calcular la raíz cuadrada del valor -4 para asignar el resultado en  $x$  (recuerde que calcular la raíz cuadrada es lo mismo que elevar a la potencia 0.5):

```

x = (-4)**0.5
print('x:', x)

```

El resultado de ejecutar este script Python, es la siguiente salida en consola estándar (remarcamos la **parte real en azul** y la **parte imaginaria en rojo**, para mayor claridad):

```
x: (1.2246467991473532e-16+2j)
```

Como se ve, Python realiza el cálculo obteniendo un valor de tipo *complex*, y lo mismo ocurrirá cada vez que se quiera calcular cualquier raíz de valor par (raíz cuarta, raíz sexta, etc.) de un número negativo. Note que el número en coma flotante que representa la *parte real* del complejo está expresado en este caso en *notación científica*: el valor flotante

```
1.2246467991473532e-16
```

equivale a:

```
1.2246467991473532 * 10-16
```

lo que finalmente equivale al número:

```
0.00000000000000012246467991473532
```

que es prácticamente igual a cero.

Sabemos que  $\sqrt{-4} = (0 + 2j)$  y no  $(1.2246467991473532e-16+2j)$  (que fue el resultado entregado por nuestro script). En la práctica, y como vimos, la *parte real* del complejo calculado por Python es igual a cero hasta una precisión de 15 dígitos de mantisa (o de decimales a la derecha del punto) y el resto de esos dígitos pueden ser despreciados por redondeo: a los efectos prácticos, el complejo entregado por Python equivale a  $(0 + 2j)$  y esto no contradice lo que hemos calculado en forma manual<sup>3</sup>.

Por otra parte, si un programador necesita asignar directamente un número complejo en una variable, en Python tiene un par de mecanismos para hacerlo recurriendo a la ya citada clase *complex*. Una forma se basa en el siguiente ejemplo [3]:

```
c1 = complex(3.4, 4)
print('c1:', c1)
```

La función *complex()* en este caso toma dos parámetros: el primero (3.4 en el ejemplo) será la *parte real* del complejo a crear, y el segundo (4 en este caso) será la *parte imaginaria*. El factor *j* será automáticamente incluido por Python. La salida de este script será la siguiente:

```
c1: (3.4+4j)
```

Por otra parte, la misma función *complex()* puede ser usada de forma que se le envíe una cadena de caracteres representando al complejo, y no ya los dos números:

```
c2 = complex('2-5j')
print('c2:', c2)
```

El script anterior producirá la siguiente salida en consola estándar:

```
c2: (2-5j)
```

Sólo debemos hacer notar que si se crea un complejo usando una cadena de caracteres como parámetro de la función *complex()*, no deben introducirse espacios en blanco a los lados del signo + o del signo – que separa ambas partes del complejo. El mismo ejemplo pero expresado con espacios:

<sup>3</sup> La forma en que se representan internamente los números en coma flotante en un computador lleva a mínimos errores de precisión que se hacen más evidentes mientras más dígitos a la derecha de la coma se miren, pero en muchas situaciones prácticas estos errores de precisión pueden despreciarse (como en el caso del ejemplo analizado).

```
c2 = complex('2 - 5j')
print('c2:', c2)
```

producirá un error en tiempo de ejecución como éste:

```
Traceback (most recent call last):
  File "C:/ Ejemplo/prueba.py", line 10, in <module>
    c2 = complex('2 + 5j')
ValueError: complex() arg is a malformed string
```

Todo lo que hemos introducido hasta aquí respecto del tratamiento de números complejos en Python, puede ser aplicado ahora en el análisis y solución de un problema clásico del Álgebra:

**Problema 17.)** Un polinomio de segundo grado tiene la forma  $p(x) = ax^2 + bx + c$  donde  $a$ ,  $b$ , y  $c$  son valores constantes conocidos como los coeficientes del polinomio y  $x$  es la variable independiente. Se supone que el coeficiente  $a$  es diferente de cero (pues de lo contrario el término  $ax^2$  desaparece y el polinomio se convierte en un de primer grado).

Si el polinomio se iguala a cero, se obtiene una ecuación de segundo grado:  $ax^2 + bx + c = 0$  y resolver la ecuación es el proceso de hallar los valores de  $x$  que hacen que efectivamente el polinomio valga cero.

Por el Análisis Matemático se sabe que todo polinomio de grado  $n$  tiene exactamente  $n$  raíces (reales y/o complejas) para su ecuación y por lo tanto, una ecuación de segundo grado tiene dos raíces a las que tradicionalmente se designa como  $x_1$  y  $x_2$ . El problema de encontrar estos dos valores fue estudiado desde varios siglos antes de Cristo al menos por los babilonios, los indios y los árabes (de hecho, nuestro ya conocido Al-Jwarizmi contribuyó de forma significativa) y la fórmula de cálculo es bien conocida:

$$x_{1-2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

El valor de  $x_1$  se obtiene usando el signo  $+$  (más) en el numerador, y el valor de  $x_2$  se obtiene usando el signo  $-$  (menos). Desarrolle un programa que dados los valores de los coeficientes  $a$ ,  $b$  y  $c$  (y suponiendo que  $a$  será diferente de cero) calcule y muestre las dos raíces  $x_1$  y  $x_2$  de la ecuación, incluso si las mismas son complejas. El programa debe permitir resolver más de una ecuación: al finalizar con una de ellas, el mismo programa debe preguntar al usuario si desea continuar con otra, y en ese caso cargar otro juego de coeficientes y volver a resolver.

**Discusión y solución:** Está claro que los datos a ingresar son los valores de los coeficientes  $a$ ,  $b$  y  $c$  y que estos valores serán *números reales* (de coma flotante). Los resultados a obtener también son claros: los valores de las raíces  $x_1$  y  $x_2$  que podrán ser *números reales* o eventualmente *números complejos*.

El hecho que puede llevar a que las raíces sean números complejos surge de la presencia del cálculo de la raíz cuadrada que se ve en el numerador de la fórmula. La expresión  $b^2 - 4ac$  (que se conoce como el *discriminante* de la ecuación) podría dar como resultado un valor negativo, y en ese caso la raíz cuadrada no puede calcularse en el campo real. En consecuencia, si el discriminante fuese negativo las raíces de la ecuación estarían en el campo complejo e incluso en ese caso nuestro programa debe mostrar sus valores.

Si programador está trabajando con Python, entonces la solución a este problema es directa y lineal, ya que Python calculará esa raíz cuadrada y obtendrá resultados reales o complejos

según sea el caso. Hay poco que discutir aquí, y mostramos la solución como un script directo:

```
__author__ = 'Cátedra de AED'

seguir = 's'
while seguir == 's' or seguir == 'S':

    # título y carga de datos...
    print('Raíces de la ecuación de segundo grado...')
    a = float(input('a: '))
    b = float(input('b: '))
    c = float(input('c: '))

    # procesos: aplicar directamente las fórmulas...
    x1 = (-b + (b**2 - 4*a*c)**0.5) / (2*a)
    x2 = (-b - (b**2 - 4*a*c)**0.5) / (2*a)

    # visualización de resultados...
    print('x1:', x1)
    print('x2:', x2)

    seguir = input('Desea resolver otra ecuación? (s/n): ')

print('Gracias por utilizar nuestro programa...')
```

El programa incluye un ciclo **while** que permite resolver una ecuación y ofrecer al usuario la posibilidad de resolver otra. La variable *seguir* comienza con el valor 's' (abreviatura de 'sí') para forzar al ciclo a entrar en la primera vuelta y resolver la primera ecuación. Pero luego de resolverla, al final del bloque de acciones del ciclo se muestra un mensaje al usuario preguntándole si desea resolver otra o no. La respuesta del usuario se carga por teclado en la variable *seguir*, asumiendo que cargará una 's' si quería responder que 'sí', o una 'n' para responder que 'no'. Al retornar a la cabecera del ciclo, se chequea el valor de la variable *seguir*: si la misma llegó valiendo una 's' o una 'S' (habilitando que tanto la minúscula como la mayúscula sean válidas) el ciclo ejecutará otra vuelta, cargando nuevamente valores en las variables *a*, *b* y *c* y resolviendo la nueva ecuación. El ciclo se detendrá cuando el usuario *no cargue* una 's' ni una 'S': en los hechos, terminará con *cualquier letra* diferente de esas dos.

En cuanto al resto del programa, es interesante remarcar que las variables *x1* y *x2* serán asignadas con los valores correctos, sean estos de tipo *float* o de tipo *complex*, y esto es así debido al hecho (que ya hemos estudiado) de ser Python un lenguaje de *tipado dinámico* [2].

Ahora bien: ¿qué pasaría en el caso que el lenguaje de programación usado no tuviese disponible la clase *complex*, lista para usar e integrada en la forma de trabajo de sus operadores aritméticos? Incluso si el programador sabe que puede usar Python, podría presentarse este caso como un desafío práctico o como un objetivo didáctico. En resumen: ¿cómo plantear (incluso en Python) el cálculo de las raíces de la ecuación de segundo grado, sin usar la clase *complex* de Python?

Ahora el trabajo es un poco más desafiante, ya que el programador deberá evitar el cálculo de la raíz cuadrada si el discriminante es negativo, y en ese caso hacer los cálculos de los valores de la parte real y la parte imaginaria para finalmente mostrar todo en la consola de salida. Por lo pronto, entonces, sería prudente calcular el valor del discriminante y dejarlo alojado en una variable:

$$d = b^{**2} - 4*a*c$$

La instrucción anterior calcula el valor del discriminante y lo deja almacenado en una variable **d**. Recuerde que el discriminante no incluye el cálculo de la raíz, sino sólo el valor de la expresión  $b^2 - 4ac$ .

Por otra parte, si el valor del discriminante **d** es mayor o igual a cero, entonces el cálculo de las raíces no presenta problema alguno y puede hacerse exactamente en la misma forma que se aplicó en nuestro primer script:

```
if d >= 0:
    # raices reales...
    x1 = (-b + d**0.5) / (2*a)
    x2 = (-b - d**0.5) / (2*a)
```

La secuencia anterior hace los cálculos de ambas raíces en forma directa, aplicando las fórmulas, pero usando el valor del discriminante que fue almacenado antes en la variable **d**. Con esto se logra algo de ahorro de código fuente y también un poco de ahorro de tiempo al ejecutar el programa, ya que no tendrá que volver a calcular dos veces más el discriminante.

Para el cálculo de las raíces complejas tendremos algo de trabajo extra. La raíces serán complejas sólo si el discriminante **d** es negativo, y en este caso ambas raíces tendrán la misma *parte real* y la misma *parte imaginaria*, pero aplicando la suma en **x1** y la resta en **x2** (se dice esos dos números complejos son *complejos conjugados*). Para deducir cuánto vale entonces la parte real *pr* de ambos complejos, y cuánto vale la parte imaginaria *pi* de ambos, nos basamos en el siguiente desarrollo, que se aplica en general y sin importar por ahora si el discriminante es negativo o no (ver Figura 7):

**Figura 7: Desarrollo del cálculo de las raíces de la ecuación de segundo grado**

$$x_{1-2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x_{1-2} = -\frac{b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_{1-2} = -\frac{b}{2a} + \frac{\sqrt{d}}{2a}$$

Si en la última expresión del desarrollo anterior el valor de **d** es negativo, entonces el factor  $\sqrt{d}$  no puede calcularse en el campo real. Ambas raíces serán complejas y el cálculo debe seguir en el campo complejo. Para eso, el discriminante **d** debe cambiar de signo, y hacer aparecer la unidad imaginaria *j*. El cambio de signo del valor contenido en una variable **d**, se puede hacer simplemente así:

$$d = -d$$

Observe: si el valor inicial era **d** = 4, entonces **-d** es -4. Pero si el valor era **d** = -4, entonces el valor **-d** es: **-d** = -(-4) o sea **-d** = 4. El valor cambiado de signo (**-d**) se asigna en la propia variable **d** y eso provoca el cambio del signo del discriminante a los efectos del cálculo. Y

entonces, según la última parte del desarrollo que hicimos en la *Figura 7*, nos queda que la parte real  $pr$  de ambos complejos  $x1, x2$  es:

$$pr = -\frac{b}{2a}$$

Y la parte imaginaria  $pi$  de ambos complejos  $x1, x2$ , es:

$$pi = \frac{\sqrt{-d}}{2a} * j$$

De este modo, el **cálculo de las raíces complejas** puede verse como sigue, en la rama falsa de la condición:

```
if d >= 0:
    # raices reales...
    x1 = (-b + d**0.5) / (2*a)
    x2 = (-b - d**0.5) / (2*a)

else:
    # raices complejas...
    # calcular la parte real...
    pr = -b / (2*a)

    # calcular la parte imaginaria...
    # ...cambiando el signo del discriminante...
    pi = (-d)**0.5 / (2*a)

    # armar cadenas uniendo las partes a mostrar...
    x1 = '(' + str(pr) + '+' + str(pi) + 'j)'
    x2 = '(' + str(pr) + '-' + str(pi) + 'j)'
```

El resultado final en este caso está formado por cadenas de caracteres: en  $x1$  se asigna una cadena formada por la conversión a string de los números  $pr$  y  $pi$ , unida con un par de paréntesis, el signo más ('+') y la letra 'j' que representa a la unidad imaginaria. Algo similar se hace con  $x2$ , pero cambiando el signo más ('+') por el menos ('-').

Note que para armar cada cadena final, las distintas partes se unen o concatenan usando el **operador suma** (que justamente actúa como un concatenador cuando suma cadenas de caracteres) [3]:

```
x1 = '(' + str(pr) + '+' + str(pi) + 'j)'
x2 = '(' + str(pr) + '-' + str(pi) + 'j)'
```

Sin embargo, recuerde que  $pr$  y  $pi$  son variables que al momento de ejecutar estas dos instrucciones contienen números (y no cadenas). Si el intérprete encuentra que el operador suma debe "sumar" una cadena (como 'j') con un número (como  $pr$ ), el programa se interrumpirá con un mensaje de error: no pueden unirse o sumarse una cadena y un número. Para evitar este problema, se usa la función `str()` de la librería estándar de Python (ver Ficha 3, sección 3) que permite convertir un objeto cualquiera en una cadena de caracteres. Por ejemplo, si  $pr$  vale 2.45, entonces `str(pr)` retornará la cadena '2.45' que es el número original, pero en formato de string.

El programa completo para el cálculo de las raíces puede verse entonces así:

```
__author__ = 'Cátedra de AED'

seguir = 's'
while seguir == 's' or seguir == 'S':
```

```

# título general y carga de datos...
print('Raíces de la ecuación de segundo grado...')
a = float(input('a: '))
b = float(input('b: '))
c = float(input('c: '))

# procesos: calcular y controlar el discriminante...
d = b**2 - 4*a*c

# ... y calcular las raíces según corresponda...
if d >= 0:
    # raíces reales...
    x1 = (-b + d**0.5) / (2*a)
    x2 = (-b - d**0.5) / (2*a)

else:
    # raíces complejas...
    # calcular la parte real...
    pr = -b / (2*a)

    # calcular la parte imaginaria...
    # ...cambiando el signo del discriminante...
    pi = (-d)**0.5 / (2*a)

    # armar cadenas uniendo las partes a mostrar...
    x1 = '(' + str(pr) + '+' + str(pi) + 'j)'
    x2 = '(' + str(pr) + '-' + str(pi) + 'j)'

# visualización de resultados...
print('x1:', x1)
print('x2:', x2)

seguir = input('Desea resolver otra ecuación? (s/n): ')

print('Gracias por utilizar nuestro programa...')


```

Este script une todas las piezas de la división en subproblemas que hemos propuesto (raíces reales calculadas como un proceso separado de las raíces complejas). Luego de cargar los valores de las variables  $a$ ,  $b$ ,  $c$  calcula el discriminante y deja ese valor en la variable  $d$ . Se usa una condición para verificar el signo de  $d$ , y se calculan las raíces reales o las raíces complejas según sea el caso. Al final, se muestran los valores finales de  $x_1$  y  $x_2$ , que serán dos números reales si las raíces eran reales, o dos cadenas de caracteres representando a los dos complejos si las raíces eran complejas. Igual que antes, el programa incluye un ciclo *while* que permite al usuario del programa cargar una nueva ecuación si así lo desea.

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [3] Python Software Foundation, "Python Documentation", 2021. [Online]. Available: <https://docs.python.org/3/>.

# Ficha 7

## Estructuras Repetitivas: El Ciclo for

### 1.] El ciclo for en Python.

El *ciclo for* suele ser el más práctico cuando se conoce previamente la cantidad de repeticiones a realizar. El ejemplo que vimos en la *Ficha 6* para cargar tres números y contar cuántos eran negativos, es un caso ideal para resolver mediante un *ciclo for* porque conocemos la cantidad de repeticiones (tres, o las que sea que se indiquen).

Sin embargo, el *ciclo for* en Python está formulado para hacer mucho más que simplemente ejecutar en forma repetida un bloque de acciones durante un número determinado de veces. En realidad es un ciclo especialmente diseñado para *recorrer secuencias* como *tuplas*, *cadenas de caracteres*, *rangos*, *listas*, etc. (presentadas en la *Ficha 3*) recuperando de a un elemento por vez (es decir, un elemento en cada vuelta) [1].

Por ejemplo, si definimos una *tupla* con tres nombres de esta forma:

```
nombres = ('Juan', 'Pedro', 'Maria')
```

es posible plantear un *ciclo for* de manera que use una variable para recuperar de a un nombre por vez en cada vuelta. El siguiente script de código Python define la tupla y muestra de a un nombre por vez en la consola de salida:

```
nombres = ('Juan', 'Pedro', 'Maria')
for nom in nombres:
    print(nom)
```

Analicemos el script línea por línea. Primero se define una tupla llamada *nombres* con los valores de las cadenas '*Juan*', '*Pedro*' y '*Maria*'. Luego se lanza un *ciclo for* que define la variable *nom* y está planteado para ejecutar en forma repetida el bloque de acciones asociado a él. En este caso, el *ciclo for* repetirá la ejecución de la única instrucción que tiene en su bloque: *print(nom)*.

En cada repetición el propio ciclo se encarga de colocar en la variable *nom* un elemento de la tupla en el orden en que han sido alojados en ella y finaliza cuando ya no tiene elementos que tomar de la tupla. La cantidad de *vueltas* (o *iteraciones* o *repeticiones*) que hace este ciclo en el ejemplo es tres, porque la tupla que está recorriendo tiene exactamente tres elementos. La salida en consola del script anterior será algo como:

```
Juan
Pedro
Maria
```

La primera línea de la instrucción *for* (en el ejemplo: *for nom in nombres*) se conoce como la *cabecera del ciclo*. Típicamente la cabecera incluye la definición de una variable (*nom* en este caso) que será usada para ir almacenando uno a uno los valores que se recuperen automáticamente desde la estructura de datos que se pretende recorrer (la tupla *nombres* en este caso).

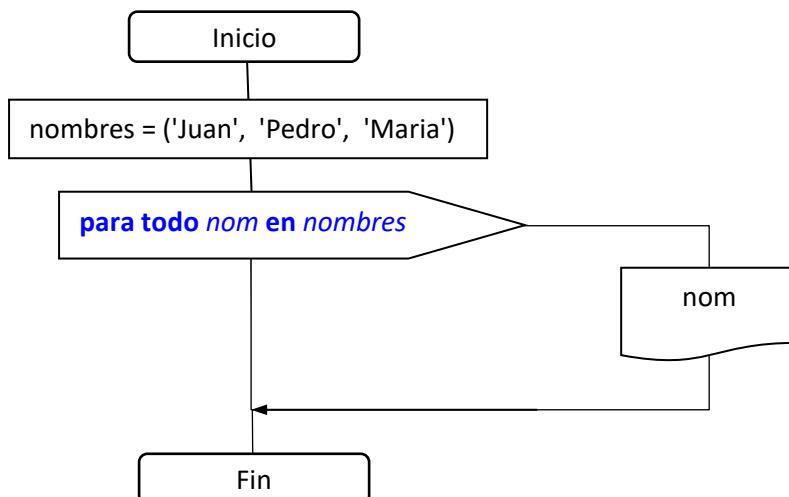
La secuencia de instrucciones que el ciclo debe ejecutar en forma repetida, se conoce como el *bloque de acciones* del ciclo, y obviamente debe escribirse indentado hacia la derecha de la cabecera. En el ejemplo el bloque sólo contiene una instrucción de visualización (`print(nom)`), pero podría contener tantas instrucciones como el programador disponga.

En cuanto a su representación en un diagrama de flujo, el *ciclo for* típicamente se ha representado siempre con el símbolo no estándar que hemos mostrado para el *ciclo while*, con diversas variantes, escribiendo en su interior los elementos que definen la *cabecera del ciclo*, aunque sin necesidad de usar sintaxis estricta de Python o de otro lenguaje: puede escribir los elementos de la cabecera en castellano, en forma resumida, o de la forma que prefiera, siempre que deje en claro lo que necesita que haga el ciclo.

La *Figura 1* muestra el diagrama de flujo que representa al sencillo script anterior. Como se ve en el diagrama, el símbolo que representa al ciclo contiene en su interior la definición informal de la cabecera del mismo, y el bloque de acciones se dibujó *indentada hacia la derecha* (en forma similar a lo que se hace con la rama verdadera de una condición en un diagrama, y respetando el hecho que luego ese bloque será efectivamente indentado hacia la derecha en el código fuente del programa).

El gráfico busca expresar que las instrucciones del bloque se ejecutarán en forma repetida una y otra vez hasta que se hayan procesado todos los elementos de la tupla *nombres*. Para indicar el final del ciclo se dibuja una línea recta que sale desde abajo del símbolo del ciclo y continúa con la línea descendente del diagrama original (de nuevo, en forma similar a lo que se hace al graficar una *instrucción condicional simple*, sin rama *else*).

**Figura 1: Diagrama de flujo del script para mostrar tres nombres con un ciclo for.**



Cuando se tiene una estructura de datos cualquiera (por ejemplo, una *secuencia* de cualquier tipo) la operación de recorrer esa estructura y procesar de a uno sus elementos se designa en forma general como *iterar* la estructura. En forma similar, la variable que se use para ir tomando uno a uno los valores de la estructura a medida que se itera sobre ella, se conoce como *variable iteradora* (en el ejemplo, el *for* está *iterando* sobre la tupla *nombres*, y está usando a la variable *nom* como *variable iteradora*).

Por lo tanto, se puede decir que el ciclo *for* en Python está esencialmente previsto para la *iteración de estructuras de datos*. El siguiente ejemplo muestra un *ciclo for* iterando sobre

todos los caracteres de una cadena contenida en la variable *ciudad*. El *for* usa la *variable iteradora c* para ir almacenando una copia de los caracteres de la cadena, a razón de uno por vuelta del ciclo y en el orden en que aparecen, y también va mostrando cada carácter por separado en la consola de salida:

```
ciudad = 'Cordoba'
for c in ciudad:
    print(c)
```

Note que la variable *ciudad* es de tipo *cadena de caracteres*, pero la variable iteradora *c* va tomando de a un carácter por vez. El ciclo finaliza su ejecución cuando se terminan de procesar todos los caracteres de la cadena.

Nos interesa ahora definir cómo se puede definir un *ciclo for* que nos permita realizar una *determinada cantidad de repeticiones* cuando nuestra necesidad no sea procesar uno a uno los elementos de una secuencia. En otras palabras, queremos plantear un *ciclo for* para resolver problemas como el de *cargar tres números y contar cuántos de esos números eran negativos* (ver *Ficha 06*).

La forma básica de hacerlo consiste en crear un tipo de secuencia numérica designada como *rango* (o *range*), de forma que el ciclo itere sobre ella. Una secuencia de tipo *range* es una sucesión *inmutable* (sus componentes no pueden ser cambiados una vez creada la secuencia) de valores numéricos en un intervalo definido, que puede ser creada fácilmente con la función *range()* [1]. Por ejemplo, el siguiente script usa un *ciclo for* para mostrar una secuencia numérica que contiene todos los números enteros entre el 1 y el 5:

```
for i in range(1, 6):
    print(i)
```

El ciclo del ejemplo anterior realiza cinco repeticiones y en cada una ejecuta la instrucción *print(i)* para mostrar el valor que tenga en esa vuelta la variable iteradora *i*. La variable *i* comienza con el valor 1 en la primera vuelta, y luego en cada una de las iteraciones del ciclo cambia automáticamente su valor para tomar el siguiente del *range*. Cuando la variable *i* llega a tomar el valor 5 (o sea, el último contenido en el *range*), el ciclo procesa ese valor y se detiene. Observe que la variable *i* nunca llega a tomar el valor 6 con el cual se invocó a la función *range()* para crear la secuencia.

Note entonces que la secuencia generada con *range(a, b)* incluye al valor *a* pero no incluye al valor *b*. Por supuesto, los valores inicial y final de la secuencia pueden venir almacenados en variables, y los valores inicial y final pueden ser cualesquiera. Si se invoca a *range()* con un solo parámetro, se asume que ese es el valor final (sin incluirlo) de la secuencia a generar, y se toma el valor inicial como 0(cero):

```
# muestra los números del 0 al 5...
for i in range(6):
    print(i)
```

La función *range()* acepta si es necesario un tercer parámetro que indica el *incremento* a usar para pasar de un valor a otro de la secuencia generada (este incremento puede ser positivo o negativo). Por defecto se asume que ese incremento es 1(uno). El mismo script que usamos para mostrar los números del 1 al 5, podría haber sido planteado así:

```
# muestra los números del 1 al 5...
for i in range(1, 6, 1):
    print(i)
```

En general, un range  $r$  creado con la instrucción  $r = \text{range}(start, stop, step)$  representa una secuencia numérica inmutable que contiene sólo a los números dados por las siguientes dos expresiones [1]:

- Si  $\text{step} > 0$  entonces el contenido de cada elemento  $r[i]$  se determina como:

$r[i] = start + step * i$  donde  $i \geq 0$  y  $r[i] < stop$ .

- Si  $\text{step} < 0$  entonces el contenido de cada elemento  $r[i]$  se determina como:

$r[i] = start + step * i$  donde  $i \geq 0$  y  $r[i] > stop$ .

Por lo tanto, si  $r = \text{range}(1, 10, 2)$  entonces  $r$  representará una secuencia numérica de la forma  $(1, 3, 5, 7, 9)$  ya que según las fórmulas anteriores:

```
step = 2 es mayor a 0 por lo tanto:
r[0] = start + step*0 = 1 + 2*0 = 1 (y como 1 < stop (=10), se acepta)
r[1] = start + step*1 = 1 + 2*1 = 3 (y como 3 < stop (=10), se acepta)
r[2] = start + step*2 = 1 + 2*2 = 5 (y como 5 < stop (=10), se acepta)
r[3] = start + step*3 = 1 + 2*3 = 7 (y como 7 < stop (=10), se acepta)
r[4] = start + step*4 = 1 + 2*4 = 9 (y como 9 < stop (=10), se acepta)
```

Este *range* no contiene a todos los números entre 1 y 10: sólo los que cumplen con la primera de las fórmulas de cálculo, y su tamaño es 5 (y no 10).

En el siguiente ejemplo, entonces, el *ciclo for* muestra los primeros  $n$  números pares, comenzando desde el cero:

```
n = int(input('Cuantos pares quiere mostrar?: '))
for par in range(0, 2*n, 2):
    print(par)
```

Si el valor cargado en  $n$  fuese, por ejemplo, el 4, entonces el *ciclo for* del script anterior generará el *range* de 4 números  $(0, 2, 4, 6)$  y comenzará haciendo que la variable *par* valga 0, mostrando el 0 en la primera vuelta. Si el tercer parámetro fuese 1 en lugar de 2, el ciclo mostraría ocho números, ya que en ese caso, el *range* efectivamente contendría a los ocho números del 0 al 7. Recuerde que si efectivamente desea un paso de avance igual a 1, entonces no es necesario incluir el tercer parámetro: Python asumirá que su valor es 1.

Como el incremento o paso de avance puede ser negativo, entonces el siguiente script simplemente muestra los números del 5 al 1 en forma descendente:

```
for i in range(5, 0, -1):
    print(i)
```

Aquí la invocación  $\text{range}(5, 0, -1)$  crea la secuencia de números  $(5, 4, 3, 2, 1)$  (en base a la fórmula 2 de cálculo de contenidos de un *range*) y luego esa secuencia se itera usando la variable *i* del *ciclo for*, que comienza valiendo 5. Cuando *i* haya tomado todos los valores del *range*, el ciclo se detendrá.

Es interesante notar que el mismo script anterior, pero sin el incremento de -1 indicado en la función *range()* en su tercer parámetro, no mostrará nada en la consola de salida ya que el ciclo no hará ninguna repetición:

```
for i in range(5, 0):
    print(i)
```

Esto se debe a que en este caso el incremento o paso de avance es igual a 1, que es positivo. Como en este caso se aplica la fórmula 1, los valores generados serían de la forma  $(5 + 1*0, 5 + 1*1, 5 + 1*2, \dots) = (5, 6, 7, \dots)$  y como ninguno de ellos es menor a 0 (que es el valor *stop*) y

la fórmula 1 exige que cada valor  $r[i]$  calculado sea menor a *stop*, entonces el *range* queda vacío y el ciclo no hace repetición alguna.

El caso es que la función *range()* nos permite definir muy simplemente una secuencia de números enteros sucesivos lo que, a su vez, es muy útil a la hora de plantear un *ciclo for* que permita una determinada cantidad de iteraciones. Si volvemos al problema de *cargar tres números y contar los negativos*, es fácil ver que ahora podemos usar un *ciclo for* ajustado para hacer tres repeticiones y ahorrar muchas líneas de código. El siguiente programa aplica la idea (y es un replanteo del programa de la Ficha 06 que se basaba en un *while*):

```
__author__ = 'Catedra de AED'

a = 0
for i in range(1, 4):
    num = int(input('Ingrese un número: '))
    if num < 0:
        a += 1

print('Cantidad de negativos cargados:', a)
```

Al ejecutarlo vemos que en tres ocasiones nos pide ingresar un número y al finalizar nos indica cuántos de esos números eran negativos. La cabecera del *ciclo for* crea un *range* con los números (1, 2, 3) (recuerde que en este caso el 4 no queda incluido) y define la *variable iteradora* *i* para recorrer ese *range*. El *bloque de acciones* del ciclo consta de una instrucción de carga por teclado para la variable *num*, el chequeo del valor *num* para saber si es negativo, y el incremento de un contador en caso de serlo. Hay dos detalles interesantes a rescatar:

1. Las *instrucciones del bloque de acciones* del ciclo de este programa, estaban escritas tres veces en el programa original sin ciclos de la *Ficha 06*. Ahora, esas instrucciones se escribieron una sola vez, y el ciclo se encarga de repetir su ejecución tantas veces como se haya dispuesto.
2. La variable iteradora *i* que va tomando los valores del *range* no se utiliza dentro del bloque de acciones del ciclo en este caso. Tenemos aquí un buen ejemplo de cómo definir un *ciclo for* que lance una cantidad predefinida de repeticiones, aunque en realidad no nos resulte relevante trabajar con los valores del *range* usado para controlarlo.

Note que *si la variable iteradora no va a usarse explícitamente dentro del bloque de acciones*, sino que será usada sólo para controlar la cantidad de repeticiones, entonces puede usarse *cualquier range en ese ciclo*, siempre que tenga *la misma cantidad de elementos que el original*. El ciclo mostrado en el programa anterior ejecuta tres iteraciones, pero podría ser reemplazado por el siguiente (*que también ejecuta tres iteraciones*), sin cambio alguno en el resultado del programa:

```
for i in range(11, 14):
    num = int(input('Ingrese un número: '))
    if num < 0:
        a += 1
```

Por otra parte, nada impide que el programa que se acaba de mostrar sea replanteado para que en lugar de sólo tres números, nos pida tantos como el usuario necesite. El único cambio a realizar, es incluir la carga por teclado de una variable *n* antes del ciclo, en la cuál se almacene la cantidad de números que se desee ingresar, y luego cambiar la llamada a *range()* para que genere una secuencia entre 0 y *n*-1 (lo cual sería algo como: *range(n)*):

```

__author__ = 'Catedra de AED'

a = 0
n = int(input('Cuantos numeros quiere procesar?: '))
for i in range(n):
    num = int(input('Ingrese un numero: '))
    if num < 0:
        a += 1

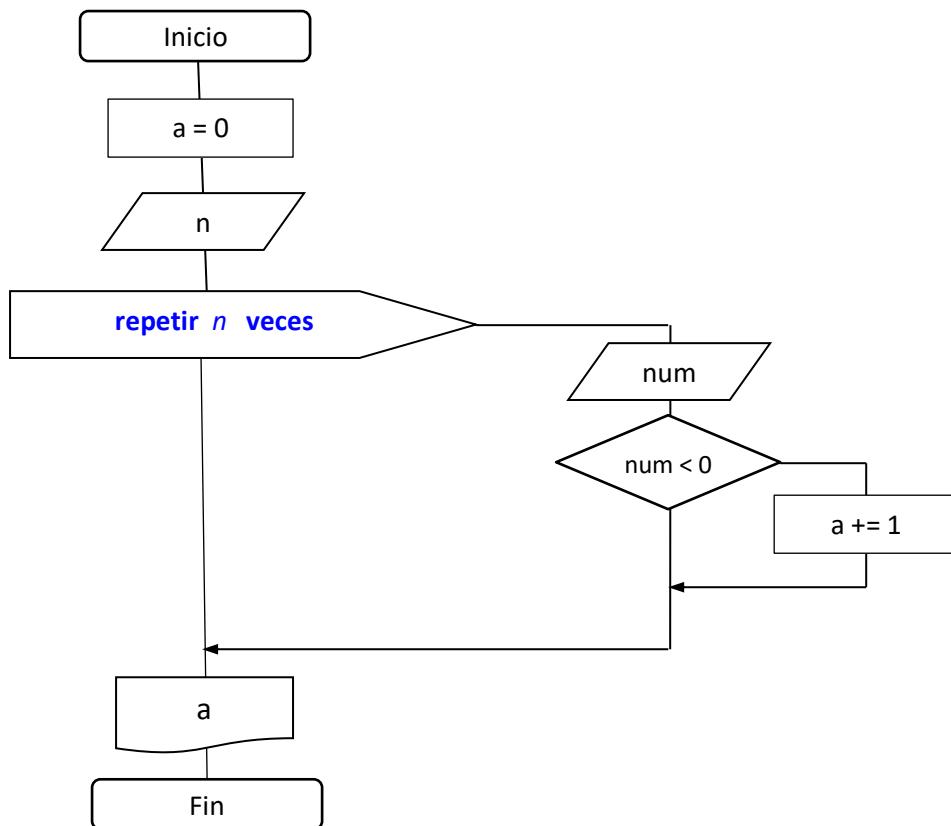
print('Cantidad de negativos cargados:', a)

```

Si se quisiera procesar cinco números, el valor cargado en *n* sería 5 y la invocación *range(n)* sería equivalente a *range(5)*. Esto crea un *range* de la forma (0, 1, 2, 3, 4) que comienza en 0 y termina en 4, pero contiene exactamente 5 valores... que es la cantidad de iteraciones que necesitábamos para el *ciclo for*. No es obligatorio que el *range* comience en el valor 1 y termine en *n* (=5): sólo necesitamos que haya *n* (=5) elementos en el *range* a iterar.

El siguiente diagrama de flujo representa la solución final para el problema, usando un *ciclo for* con los detalles explicados:

Figura 2: Diagrama de flujo (problema del conteo de negativos) con un ciclo **for** de *n* iteraciones



La versión final del programa deja clara la ventaja de usar un ciclo en lugar de una estructura secuencial: no importa cuántos números se necesite procesar (no importa el valor de *n*): el programa es siempre el mismo y desde el punto de vista de la escritura del código fuente da igual si *n* es 3, o 5, o 100, o 100000... El *bloque de acciones del ciclo* contiene lo que debe hacerse con *uno* de los números, y el ciclo repite la ejecución de ese bloque tantas veces como números haya.

No ocurre lo mismo con una *estructura secuencial* en la que la cantidad de instrucciones a escribir en el programa guarda una relación directa con la cantidad de datos  $n$  a procesar: si  $n$  es 5, el programador escribirá cinco veces el bloque de acciones. Si  $n$  es 100, el programador escribirá cien veces ese bloque... y si todavía no comprende la diferencia, intente escribir el programa anterior mediante una estructura secuencial y sin el ciclo, pero suponiendo que serán  $n = 10000$  los números a procesar. Habrá que armarse de paciencia... y además esperar a que luego no cambie el requerimiento y se pida procesar  $n = 15000$  números en lugar de 10000.

A modo de cierre de esta introducción al uso del *ciclo for*, analicemos el siguiente problema:

**Problema 18.)** *Sea  $n$  un número entero mayor o igual a 0. El **factorial** del número  $n$  (denotado algebraicamente como  $n!$ ) es el producto de todos los números enteros en el intervalo  $[1, n]$  si  $n > 0$ . En caso que  $n = 0$ , entonces se define  $0! = 1$ . Más formalmente:*

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1) * (n-2) * \dots * 3 * 2 * 1 & \text{si } n > 0 \end{cases}$$

*A modo de ejemplo:*

$$\begin{aligned} 5! &= 5 * 4 * 3 * 2 * 1 = 120 \\ 7! &= 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040 \end{aligned}$$

*Se pide desarrollar un programa en Python que cargue por teclado un número  $n$  y calcule y muestre su factorial.*

**Discusión y solución:** Este es un problema clásico para plantear en base a un ciclo que ejecute un número predeterminado de iteraciones. El producto de todos los números enteros entre 1 y  $n$  requiere un ciclo que sea capaz de tomar todos esos números, uno a uno, y multiplicarlos en forma acumulativa, esto es, multiplicar dos de ellos, y al resultado volver a multiplicarlo por el que sigue, y este nuevo resultado multiplicarlo a su vez por el siguiente, hasta terminar con los  $n$  números del intervalo.

Está claro que el intervalo de todos los números enteros entre 1 y  $n$  puede obtenerse con `range(1, n+1)`: así invocada, la función `range()` nos permitirá disponer de una secuencia de números entre 1 y  $n$  (recuerde que  $n+1$  no está incluido en el `range` generado). Por lo tanto, un *ciclo for* con la siguiente cabecera:

```
for i in range(1, n+1):
```

nos permitirá tomar uno por uno los números del `range(1, n+1)` asignándolos a razón de uno por vuelta en la *variable iteradora*  $i$ . Esto nos dará en la variable  $i$  todos los factores 1, 2, 3, ... ( $n-2$ ), ( $n-1$ ),  $n$  que necesitamos para el cálculo.

El producto acumulativo puede hacerse con sencillez. Si la variable  $f$  contiene un valor cualquiera, entonces la expresión de acumulación  $f = f * i$  (que como sabemos es equivalente en Python a la expresión  $f *= i$ ) hará que el valor actual de  $f$  se multiplique por el valor de  $i$ , y el resultado se vuelva a asignar en  $f$ .

En base a las ideas generales anteriores, el programa que se muestra a continuación permite calcular y mostrar el valor del factorial del número  $n$ :

```
__author__ = 'Catedra de AED'
```

```

n = int(input('Valor de n (>=0 por favor): '))

f = 1
for i in range(1, n+1):
    f *= i

print('Factorial de', n, '=', f)

```

Una vez cargado por teclado el valor de  $n$ , el proceso comienza asignando el valor 1 en la variable  $f$ . Como  $f$  será usada como *variable de acumulación multiplicativa*, su valor inicial es 1: no puede ser 0 ya que cualquier producto que tenga a  $f$  como factor, resultaría indefectiblemente en un resultado de 0. En general, si una variable será usada como *acumulador sumativo*, se esperaría que su valor inicial sea cero (el neutro de la suma); pero si la variable será usada como *acumulador multiplicativo* entonces se esperaría que su valor inicial fuese 1 (el neutro de la multiplicación).

El *ciclo for* del programa anterior crea un *range()* con los números del 1 al  $n$ , y luego itera sobre ese *range* con la variable  $i$ . El bloque de acciones del ciclo, simplemente ejecuta la expresión  $f *= i$  (que es lo mismo que  $f = f*i$ ) almacenando el resultado en la misma  $f$ . Está claro entonces que en la primera repetición del ciclo, hará  $f = f*i = 1*1 = 1$  sin cambiar el valor de  $f$ . Pero en la segunda vuelta, el producto será  $f = f*i = 1*2 = 2$ , cambiando  $f$  al valor 2. La siguiente iteración hará  $f = f*i = 2*3 = 6$ , y así sucesivamente hasta la última vuelta, en la que finalmente  $f$  quedará asignada con el valor del factorial de  $n$  (el resultado de la última multiplicación) que será mostrado en pantalla.

Un detalle: sabemos que si  $n = 0$ , entonces  $n!$  debe ser 1. La función *factorial()* está planteada de forma que ese caso ya está previsto: si  $n$  es cero, el *ciclo for* debe iterar en el  $range(1, n+1) = range(1, 0+1) = range(1, 1)$  que es un *range* vacío al no incluir el límite derecho: incluye los elementos del intervalo  $[1, 0]$ , que no puede iterarse con un paso de avance igual a 1 (como en el *for* del script). Por lo tanto, en ese caso el *ciclo for* no hará ninguna repetición, y el proceso pasará de largo a la función *print*, mostrando el valor 1 con el que estaba inicializada  $f$ . Y este sería el resultado correcto ya que supusimos que  $n$  valía 0.

## 2.] Búsqueda del mayor (o el menor) de una secuencia cargada por teclado.

Habiendo llegado a un punto en desarrollo de temas en el que se han introducido herramientas diversas del lenguaje Python como las *estructuras condicionales* y *cíclicas* y el uso elemental de ciertas *estructuras de datos como las secuencias* (*hasta aquí, tuplas, cadenas y rangos*) podemos entonces abordar con profundidad una serie de temas, problemas, algoritmos y técnicas de la programación básica que más temprano que tarde deben ser incorporados al conjunto de conocimientos y habilidades de todo buen programador.

Parte de esta Ficha está orientada a ese objetivo. En general los temas nuevos que podrían aparecer desde este punto y hasta el final de la Ficha, lo harán en el contexto del análisis de los problemas y casos de estudio que hemos seleccionado aquí. Tómese su tiempo para comprender, aplicar y eventualmente modificar las ideas que serán expuestas, o pensar en variantes que pudieran servirle para la resolución de problemas similares. Siempre recuerde: **a programar, se aprende programando**.

El primero de los problemas que veremos constituye todo un caso de análisis clásico de la introducción a la programación: en la práctica es común enfrentar situaciones en las cuales se tiene un conjunto grande de valores que serán introducidos uno a uno en el programa (posiblemente a razón de uno por cada vuelta de un ciclo) y se *desea saber cuál es el mayor (o el menor) valor del conjunto*, o alguna característica asociada a ese valor.

Existen varias formas de hacer esto, pero la estrategia básica es la misma [2]: se trata de tomar al primer valor de la sucesión y guardarlo en una variable *may*. Luego tomar el que sigue, y compararlo contra el guardado antes en *may*. Si el nuevo valor es mayor, reemplazar el valor de *may* por el nuevo, y en caso contrario, mantener el valor de *may*. Todo el proceso es controlado por un ciclo que se detiene cuando se terminan los números de entrada. Se plantean a continuación tres variantes de resolución del problema (de acuerdo a distintas situaciones que podrían presentarse) cuyo enunciado formal sería el que sigue:

**Problema 19.) Determinar el mayor valor de una sucesión de valores que se cargan por teclado.** *Asuma en primera instancia que la cantidad n de números a procesar se conoce de antemano. Y luego de resolverlo con esa suposición, replantee su solución asumiendo que la cantidad n de números se desconoce y que la carga de datos terminará cuando se ingrese un 0.*

**Discusión y solución:** Tenemos distintas variantes interesantes para el análisis:

**Primera variante:** Como aquí se supone que se conoce de antemano la cantidad *n* de valores que serán cargados, puede usarse un ciclo *for* para la carga y procesamiento de los mismos. Según se indicó al exponer la estrategia básica para resolver el problema, la idea era comenzar tomando *el primer valor* de la sucesión y guardarlo en la variable *may*.

En esta primera variante, para determinar si el valor que se está procesando en una vuelta del ciclo es o no el primero, se usa un truco simple, pero directo: se pregunta si el valor de la variable *i* (que es la que controla al ciclo) es igual a *1 (uno)*. Si lo es, entonces el ciclo está en su primera repetición, y por lo tanto el valor leído en *num* (que es la variable que se usará para la carga de los valores) debe ser el primero, por lo cual se lo asigna en *may*. Si la variable *i* no es 1, entonces el dato cargado no es el primero, y por lo tanto se procede a compararlo con *may* para determinar si es el nuevo mayor o no. Mostramos el *pseudocódigo general* para esta solución propuesta:

```
# El valor n es la cantidad de datos a procesar.
Variante 1:
1. Repetir para i que va entre 1 y n:
   1.1 Cargar num (el número a procesar en esta vuelta)
   1.2 Si i == 1 (¿es la primera vuelta del ciclo?):
      1.2.1 Sea may = num (iniciar may con ese número)
      sino:
         1.2.2 Si num > may (¿debo reemplazar may?):
            1.2.2.1 Sea may = num
2. Mostrar may
```

El programa completo en Python sigue a continuación:

```
__author__ = 'Catedra de AED'

print('Determinacion del mayor de una sucesion (variante 1)...')
n = int(input('Cantidad de numeros a procesar: '))
```

```

for i in range(1, n+1):
    num = int(input('Número: '))
    if i == 1:
        may = num
    else:
        if num > may:
            may = num

print('El mayor es:', may)

```

Segunda variante: Si se observa bien la solución planteada en la primera variante, puede concluirse que la pregunta por el valor de *i* en cada vuelta del ciclo en realidad provoca una pérdida de tiempo: sabemos exactamente cuándo esa condición será cierta y cuándo falsa, y sabemos que sólo resulta realmente útil en la primera vuelta del ciclo.

¿Cómo evitar esa condición? Otra vez, la solución es un truco simple y directo: el primer valor de la secuencia se puede cargar *antes* de comenzar el ciclo, y asignarse directamente en la variable *may*, con lo cual la misma queda ya inicializada. El ciclo se reserva para procesar al resto de los números de la secuencia. Como el primero se procesó antes del ciclo, entonces el ciclo debe procesar todavía  $n - 1$  valores (es decir, debe dar una vuelta menos que en la primera variante). Se puede ajustar el ciclo para ir desde 1 hasta  $n - 1$ , o bien para ir desde 2 hasta  $n$ , pues en ambos casos el ciclo realizará  $n - 1$  vueltas. Dado que ahora el primer dato se carga y se procesa *antes del ciclo*, entonces ya no es necesario que en el bloque del ciclo se pregunte en cada vuelta si el dato cargado es el primero: el ciclo está procesando los datos desde el **segundo** en adelante.

El pseudocódigo general del algoritmo adaptado a la segunda variante, se muestra a continuación:

```

# El valor n es la cantidad de datos a procesar.
Variante 2:
1. Cargar num (con el primer número de la secuencia)
2. Sea may = num (iniciar may con ese primer número)
3. Repetir para i que va entre 2 y n (una vuelta menos...):
    3.1 Cargar num (el siguiente número a procesar en esta vuelta)
    3.2 Si num > may (¿debo reemplazar may?):
        3.2.1 Sea may = num
4. Mostrar may

```

Mostremos ahora el programa completo. Note que sólo ha cambiado el planteo del algoritmo de búsqueda del mayor, para eliminar la pregunta por la primera vuelta dentro del ciclo y ajustarla a la segunda variante:

```

__author__ = 'Catedra de AED'

print('Determinacion del mayor de una sucesion (variante 2)...')
n = int(input('Cantidad de numeros a procesar: '))

num = int(input('Ingrese el primer numero: '))
may = num

for i in range(2, n+1):
    num = int(input('Siguiente numero: '))
    if num > may:
        may = num

print('El mayor es:', may)

```

**Tercera variante:** En las dos variantes anteriores se usó un ciclo *for* porque la primera parte del enunciado indicaba que se conocía de antemano la cantidad *n* de valores a procesar. Pero el mismo enunciado, en su segunda parte, sugiere que se plantee una solución suponiendo ahora que se desconoce la cantidad de datos *n* a cargar. En la primera variante el ciclo *for* ayudó mucho porque se usó la variable *i* de control del mismo para determinar si se estaba ejecutando o no la primera vuelta.

¿Qué pasaría si se desconociera la cantidad *n* de valores a procesar? Supongamos que el enunciado fuese el siguiente:

*Determinar el mayor de una secuencia de valores que ingresan de a uno. Se desconoce cuantos valores serán procesados, pero se indicará el final de la secuencia cargando el número cero.*

En este caso, como sabemos, la solución es usar un ciclo *while* que controle si el número cargado en cada vuelta es distinto de cero, y aplicar un esquema de carga por doble lectura (como se explicó en la *Ficha 6*). Si efectivamente el número cargado es distinto de cero, el ciclo continúa y se aplica básicamente la misma estrategia ya vista: se almacena el primer número la variable *may*, y a los restantes se los compara contra el valor de *may* para saber si se debe actualizar o no el valor de *may*.

Puede aplicarse sin mayores problemas la *segunda variante* ya vista. Sin embargo, ¿cómo aplicaríamos la *primera variante* si tuviésemos que hacerlo? ¿Cómo podríamos preguntar por el *primer valor* sin contar con la variable *i* de control del ciclo?

Si se analiza el papel que juega la variable *i* en la *primera variante*, puede deducirse que la idea central es asignar a dicha variable un *valor inicial conocido*, y luego en cada vuelta del ciclo, *preguntar por ese valor inicial*. En nuestro caso, el valor inicial de *i* es *1 (uno)*. En cada vuelta, se pregunta si *i* sigue valiendo *1 (uno)*. Si se responde que sí, se tiene el primer valor, por estar el ciclo en la primera vuelta. Pero el hecho importante es que en las siguientes vueltas del ciclo, el valor de *i* no es *1*, ya que el propio ciclo *for* cambia su valor en cada repetición. Por lo tanto, la pregunta por el valor inicial de *i* sólo será verdadera en la primera vuelta.

Para emular esta situación cuando no hay un ciclo *for* ni un contador disponible, puede usarse el concepto de *variable centinela* o *bandera* que se introdujo también en la *Ficha 6*. Los principios son los mismos: se usa una variable cualquiera *b*, a la cual se le asigna antes de comenzar el ciclo un valor inicial conocido arbitrario (por ejemplo, el valor *true*). En cada vuelta del ciclo, se pregunta por ese valor, pero de tal forma que si la respuesta es verdadera se asigna el número leído en *may* y se *cambia el valor de la bandera* para provocar un *false* en las vueltas posteriores.

Así de directo. La única diferencia con la *primera variante* es que ahora el cambio de valor en la variable centinela no es efectuado por la cabecera del ciclo, sino por una instrucción de asignación ubicada en la rama verdadera de la condición. Y como sabemos, es muy común que la variable centinela sea definida de tipo *boolean*, pues sólo se necesitan para ella dos valores. El programa Python se muestra a continuación:

```
__author__ = 'Catedra de AED'

print('Determinacion del mayor de una sucesion (variante 3)....')
```

```

may = None
b = False

num = int(input('Ingrese un número (con 0 finaliza): '))
while num != 0:
    if b == False:
        may = num
        b = True
    else:
        if num > may:
            may = num

num = int(input('Ingrese otro (con 0 finaliza): '))

print('El mayor es:', may)

```

En el proceso se comienza definiendo la variable *may* con el valor *None*. Esto se hace por una razón preventiva: si al pedir la carga del primer número (antes del ciclo) el usuario llegase a ingresar directamente un 0, el ciclo no ejecutaría su bloque de acciones y la variable *may* no llegaría a definirse, por lo que la instrucción *print(may)* lanzaría un error [1]. Al iniciar *may* en *None* antes del ciclo, la variable queda definida aunque con un valor "ficticio", y ese será el valor que se mostrará en caso que el usuario no haya cargado dato alguno.

### 3.] Procesamiento de datos divididos en categorías distintas.

El siguiente ejercicio contiene varias de las técnicas y elementos vistos hasta aquí, incluyendo la forma de determinar el mayor de una secuencia de valores cuando se desconoce la cantidad de datos a procesar y la *secuencia de valores de entrada se divide en categorías distintas* [2]. El enunciado es el que se muestra aquí:

**Problema 20.)** *Un pequeño comercio de papelería cuenta con dos vendedores. Cada vendedor está codificado con los números 1 y 2. Considere que la carga de datos se realizará desde teclado, de forma que una entrada consta de 3 variables que representan una venta realizada: por cada venta, cargar el código del vendedor (1 o 2) que hizo la venta, cantidad de artículos vendida en esa operación, e importe de la venta. El fin de datos se indicará con código de vendedor igual a 0 (cero). El dueño del comercio desea cierta información estadística y para ello solicita un programa que obtenga lo siguiente:*

- La cantidad de productos vendida por cada vendedor (dos totales).*
- El importe total vendido por cada vendedor (otros dos totales).*
- El importe de la menor venta realizada por el vendedor 2.*
- El importe promedio de ventas por vendedor (importe total acumulado / 2).*

**Discusión y solución:** Este problema muestra un típico caso en el que los datos de entrada se presentan de alguna forma divididos en *categorías diferentes*: en este caso, se tienen ciertos datos que pertenecen a las ventas realizadas por el *vendedor 1*, y ciertos otros datos que describen ventas del *vendedor 2*. Para cada vendedor se pide calcular resultados diferenciados, y eso lleva a tener que dividir el proceso a realizar en dos ramas (una por cada vendedor). El programa completo es algo extenso, pero no demasiado complejo en sus detalles.

La carga de datos, como es costumbre, se realiza con un ciclo, que en este caso será un *while* implementando un *esquema de carga por doble lectura*, ya que se desconoce la cantidad de ventas a procesar. Por cada venta, se tienen tres datos: el código del vendedor que la hizo (variable *codigo*), la cantidad de productos vendidos (variable *cantidad*), y el importe de esa venta (variable *importe*). El ciclo debe permitir que en cada vuelta se carguen esos tres valores, y sabemos que debe detenerse si aparece un 0 en el código de vendedor. El programa completo podría quedar así:

```
__author__ = 'Catedra de AED'

# pasó la primera venta del vendedor 2?
aviso = False

# si no se cargan ventas del vendedor 2, menor_importe queda en None...
menor_importe = None

# acumuladores de cantidades...
c1 = c2 = 0

# acumuladores de importes...
i1 = i2 = 0

print('Ventas de un Comercio... ingrese los datos de cada venta...')

# ingresar (y validar) el primer código...
codigo = -1
while codigo < 0 or codigo > 2:
    codigo = int(input('Código de vendedor (1 o 2) (0 para cortar): '))
    if codigo > 2 or codigo < 0:
        print('Error... se pidió 1 o 2 o 0 para cortar...')

while codigo != 0:
    cantidad = int(input('Cantidad vendida: '))
    importe = float(input('Importe: '))

    if codigo == 1:
        c1 += cantidad
        i1 += importe

    elif codigo == 2:
        c2 += cantidad
        i2 += importe

    # Aplicar mecanismo de cálculo del menor...
    if not aviso:
        aviso = True
        menor_importe = importe

    elif importe < menor_importe:
        menor_importe = importe

# ingresar el siguiente código y volver al ciclo...
codigo = -1
while codigo < 0 or codigo > 2:
    codigo = int(input('Código de vendedor (1 o 2) (0 para cortar): '))
    if codigo > 2 or codigo < 0:
        print('Error... se pidió 1 o 2 o 0 para cortar...')

# Calcular el importe promedio...
```

```

promedio = (i1 + i2) / 2

print('Cantidad de productos vendida por el vendedor 1:', c1)
print('Cantidad de productos vendida por el vendedor 2:', c2)
print('Importe total facturado por el vendedor 1:', i1)
print('Importe total facturado por el vendedor 2:', i2)
print('Importe de la menor venta del vendedor 2:', menor_importe)
print('Importe promedio entre los dos vendedores:', promedio)

```

El programa lanza un proceso de carga por doble lectura para el *código del vendedor*, que sólo se detendrá cuando ese *código de vendedor* sea cero. Dentro del ciclo y al inicio del bloque de acciones, se cargan los demás datos de esa venta (*cantidad e importe*). Note que estas dos variables se cargan dentro del ciclo, *una vez que nos hemos asegurado que el código del vendedor no es cero*: no tendría sentido cargar los tres datos juntos (antes del ciclo o al final de su bloque) si el código del vendedor a cargar en ese momento fuese cero... por no decir que en ese caso la carga inútil de la *cantidad* y el *importe* molestaría mucho al usuario...

Note que para la carga del *código de vendedor* se está aplicando un *proceso de validación* (que se explica en detalle en la sección 5 de esta misma Ficha): básicamente, se pide el código del vendedor por teclado y se acepta ese código si se cargó correctamente, pero se vuelve a pedir si el número ingresado no es un 0, un 1 o un 2: el cero es un valor válido para el *código de vendedor*, ya que ese es justamente el valor a ingresar para detener el ciclo de carga en el programa principal.

También dentro del ciclo de carga se usa una *instrucción condicional anidada* para saber cuál de los vendedores hizo la venta, y en función de ello se acumulan las *cantidades* y los *importes* en *distintas variables de acumulación* (*c1* y *c2* para las cantidades, además de *i1* e *i2* para los importes). Esos cuatro acumuladores se definen antes del ciclo, inicializados en cero.

Si el *código de vendedor* fue el 2 (segunda rama de las instrucciones condicionales anidadas) se aplica el ya conocido *proceso de determinación del mayor/menor* que hemos analizado en esta misma Ficha, para ir quedándose con el menor de los *importes* que bajen por esa rama. Para capturar al *primero* de esos importes, se usa una *variable booleana a modo de bandera* llamada *aviso*, que empieza con el valor *False* y quedará con ese valor *hasta que se detecte la primera venta que baje por la rama 2* (y en ese momento cambia a *True*).

Al terminar el ciclo, cuando los importes individuales ya se han cargado y acumulado completamente, se calcula el *importe promedio por vendedor*. Un análisis más detallado (que recomendamos...) se deja para el estudiante.

#### 4.] Procesamiento de secuencias de caracteres.

El siguiente problema está orientado al planteo de una situación clásica como es el procesamiento de una secuencia de caracteres que forman una frase, de forma tal que el programador no sólo debe ser capaz de identificar palabras dentro de esa frase, sino también la formación de ciertos patrones en cada palabra. Esta clase de problemas forma parte de un excelente campo para el desarrollo de habilidades algorítmicas que ya dejan de ser triviales para un programador que recién se inicia [2]. El enunciado que se sugiere es el siguiente:

**Problema 21.) Desarrollar un programa en Python que permita cargar por teclado un texto completo** (analizar dos opciones: una es cargar todo el texto en una variable de tipo cadena de caracteres y recorrerla con un *for* iterador; y la otra es cargar cada carácter uno por uno a través de un *while*). Siempre se supone que el usuario cargará **un punto** para indicar el final del texto, y que cada palabra de ese texto está separada de las demás por **un espacio en blanco**. El programa debe:

- a. Determinar cuántas palabras se cargaron.
- b. Determinar cuántas palabras comenzaron con la letra "p".
- c. Determinar cuántas palabras contuvieron una o más veces la expresión "ta".

**Discusión y solución:** Este tipo de problema es especialmente desafiante debido a que implica numerosos puntos de vista para resolverlo, además de muchas variantes en cuanto al planteo posible de la carga y el soporte de datos. Como veremos, es también muy valioso en cuanto a que permite aplicar y dominar técnicas básicas como la aplicación profunda de banderas o flags, carga por doble lectura, ajuste de un ciclo para forzarlo a ser  $[1..N]$ , uso de ciclos iteradores, condiciones anidadas, y cuanta idea pudiera ser útil para llegar a una solución.

Por lo pronto, podemos suponer que en nuestra *primera versión* el texto será ingresado en forma "inocente": cada una de las "letras" ingresará de a una por vez, una por cada vuelta del ciclo de control. Como sabemos que el final del texto se indica con la carga de un punto, entonces inicialmente ese ciclo puede ser un *while con doble lectura*, en base al esquema general que sigue<sup>1</sup>:

```
car = input('Letra (con "." termina): ')
while car != '.':
    # procesar el carácter ingresado en car

    # cargar el siguiente carácter
    car = input('Letra (con "." termina): ')
```

Sin embargo, este planteo puede traer luego algún tipo de incomodidad al tener que controlar posibles nuevos caracteres *que se cargan en dos lugares diferentes* del proceso. Por ese motivo, y para simplificar a futuro, podemos también convertir el *while de doble lectura* en un *while forzado a trabajar como ciclo  $[1..N]$*  (que podemos referir de ahora en más como *while-[1..N]*), y hacer una única lectura al inicio del bloque de acciones del ciclo:

```
car = None
while car != '.':
    car = input('Letra (con "." termina): ')
    # procesar el carácter ingresado en car
```

La inicialización de *car* con el valor *None* hace que *car* quede definida (y por lo tanto exista) antes de chequear si su valor es diferente de un punto. Y como el valor *None* es

<sup>1</sup> La idea de la llegada de alguna forma de mensaje (de texto o del tipo que sea) para ser procesado e interpretado ha conducido a pensar en qué pasaría si se recibiese un mensaje o comunicación desde alguna avanzada civilización extraterrestre. Existen programas de investigación muy serios (como el *Programa SETI: Search for ExtraTerrestrial Intelligence*) orientados a captar e interpretar alguno de estos eventuales mensajes. Y el cine se hizo cargo del tema al menos en una conocida película de 1997: *Contact* (conocida como *Contacto* en español), dirigida por *Robert Zemeckis* e interpretada por *Jodie Foster*. La película es la adaptación cinematográfica de la novela del mismo nombre escrita por el famosísimo científico *Carl Sagan*, y especula sobre la reacción de la civilización humana frente a la llegada de un mensaje de una civilización de otro planeta.

efectivamente diferente de un punto, la condición de control del ciclo será indefectiblemente verdadera en el primer chequeo, haciendo que la ejecución de la primera vuelta del ciclo esté garantizada.

Lo siguiente es controlar si el carácter que se acaba de ingresar en la variable *car* es concretamente una letra (en cuyo caso hay que procesarlo como parte de una palabra), o bien determinar si ese carácter es un espacio en blanco (que no debe ser procesado como una letra sino como un terminador de palabra). En principio, el control es simple:

```
car = None
while car != '.':
    car = input('Letra (con "." termina): ')
    # procesar el carácter ingresado en car
    # ...
    # ...

    # final de palabra?
    if car == ' ':
        # ha terminado una palabra...

    else:
        # car es una letra... la palabra sigue...
```

Lo anterior permite detectar si el carácter que se acaba de cargar en *car* es un espacio en blanco y actuar en consecuencia con la palabra que acaba de terminar. Sin embargo, revisando con cuidado el enunciado podemos darnos cuenta que si bien en general todas las palabras finalizan con un blanco, hay exactamente una que finaliza con otro carácter: **la última palabra del texto, que termina con un punto** (el mismo punto que también da por terminado al texto completo). Ese caso particular debe ser previsto, y es simple de hacer:

```
car = None
while car != '.':
    car = input('Letra (con "." termina): ')
    # procesar el carácter ingresado en car
    # ...
    # ...

    # final de palabra?
    if car == ' ' or car == '.':
        # ha terminado una palabra...

    else:
        # car es una letra... la palabra sigue...
```

Si el carácter cargado en *car* es un blanco o es un punto, la palabra que se estaba procesando ha finalizado, y pueden aplicarse los procesos que sean requeridos en ese caso (sobre los cuales volveremos luego).

Teniendo definido el *esquema de control de fin de palabra*, tenemos que analizar ahora la forma de procesar cada letra para cumplir con los requerimientos del enunciado. El primero (*determinar cuántas palabras se cargaron*) es simple. Necesitamos un contador (que llamaremos *ctp* por *contador total de palabras*) inicializado en cero antes del ciclo de carga, y de forma que sume uno cada vez que se detecta que una palabra ha terminado. Cuando el ciclo se detenga, sencillamente se muestra el valor de *ctp*:

```
ctp = 0

car = None
```

```

while car != '.':
    car = input('Letra (con "." termina): ')
    # procesar el carácter ingresado en car
    # ...
    # ...

    # final de palabra?
    if car == ' ' or car == '.':
        # ha terminado una palabra...

        # ...contarla...
        ctp += 1

    else:
        # car es una letra... la palabra sigue...

print('Cantidad de palabras:', ctp)

```

Lo anterior sería estrictamente suficiente para resolver el caso. Sin embargo, un buen programador haría bien es desconfiar y buscar algún punto débil en el planteo. Y en este caso existe uno: el programa está planteado para el supuesto de que el usuario cargará palabras separadas por un blanco y terminando la última con un punto, y mantendremos esa suposición para hacer simple el planteo. Pero específicamente, podría ocurrir que el usuario cargue un blanco (o un punto) *directamente en la primera lectura* (en la primera vuelta del ciclo). Y si ese fuese el caso, el programa contaría ese blanco (o ese punto) como una palabra, lo cual es claramente incorrecto (asegúrese de entender este hecho haciendo una prueba de escritorio rápida antes de continuar leyendo el resto de la explicación...)

Para eliminar ese molesto caso, se puede incorporar al programa un contador de letras (que llamaremos *cl*). La idea es que cada vez que se cargue un carácter se incremente ese contador. Al terminar una palabra, chequear el valor de *cl* y contar la palabra en *ctp* sólo **si cl es mayor a 1** (si *cl* es 1, el carácter contado fue el blanco o el punto, y si vale más de 1 es porque necesariamente se cargó algún otro carácter adicional):

```

# contador total de palabras
ctp = 0

# contador de letras en una palabra
cl = 0

car = None
while car != '.':
    # cargar y procesar el carácter ingresado en car
    car = input('Letra (con "." termina): ')
    cl += 1

    # final de palabra?
    if car == ' ' or car == '.':
        # ha terminado una palabra...

        # ...contarla solo si hubo al menos una letra...
        if cl > 1:
            ctp += 1

        # reiniciar contador de letras (por próxima palabra)...
        cl = 0

    else:

```

```
# car es una letra... la palabra sigue...
print('Cantidad de palabras:', ctp)
```

En el esquema anterior, una vez que la palabra ha sido contada en *ctp*, el contador de letras *cl* debe volver al valor 0 antes de comenzar a procesar la palabra que sigue, pues de otro modo seguirá contando letras en forma acumulativa: en lugar de contar las letras de una palabra, estaría contando todas las letras del texto.

El segundo requerimiento es *determinar cuántas palabras comenzaron con la letra "p"*, lo cual es un poco más complejo. Está claro que ahora necesitamos saber si la letra cargada es o no es una "p", pero además queremos saber si esa "p" es la primera letra de la palabra actual. En este caso la solución es directa ya que contamos con el contador de letras *cl* que hemos usado en el caso anterior: si la letra es una "p" y el contador de letras vale 1, eso significa que efectivamente la primera letra de la palabra es una "p" y podemos entonces contar esa palabra con otro contador (que llamaremos *cpp*, por *cantidad de palabras con p*):

```
# contador total de palabras
ctp = 0

# contador de letras en una palabra
cl = 0

# contador de palabras que empiezan con p
cpp = 0

car = None
while car != '.':
    # cargar y procesar el caracter ingresado en car
    car = input('Letra (con "." termina): ')
    cl += 1

    # final de palabra?
    if car == ' ' or car == '.':
        # ha terminado una palabra...

        # ...contarla solo si hubo al menos una letra...
        if cl > 1:
            ctp += 1

        # ...reiniciar contador de letras (por próxima palabra)...
        cl = 0

    else:
        # car es una letra... la palabra sigue...

        # ...contar la palabra si comienza con "p"...
        if cl == 1 and car == 'p':
            cpp += 1

print('Cantidad de palabras:', ctp)
print('Cantidad de palabras que empiezan con "p":', cpp)
```

De nuevo, al finalizar la carga (por la aparición del punto) se muestra en pantalla el valor final del contador *cpp*, y el segundo requerimiento queda así cumplido.

El tercer y último requerimiento es *determinar cuántas palabras contenían al menos una vez la expresión (o sílaba) "ta"*, para lo que habrá que trabajar un poco más. Veamos: si el carácter es una letra, nos interesa por el momento saber si es una "t" y dejar marcado de

alguna forma ese hecho para que al ingresar el siguiente carácter podamos comprobar si el mismo es una "a" y el anterior una "t". Hay muchas formas de hacer esto, pero puede resolverse con *flags* o *banderas* para marcar los estados que nos interesan.

Por lo pronto, usaremos una bandera llamada *st* (por *señal de la letra t*) para avisar si la última letra cargada fue efectivamente una "t" (con *st = True*) o no (*st = False*):

```
# contador total de palabras
ctp = 0

# contador de letras en una palabra
cl = 0

# contador de palabras que empiezan con p
cpp = 0

# flag: la ultima letra fue una "t"?
st = False

car = None
while car != '.':
    # cargar y procesar el carácter ingresado en car
    car = input('Letra (con "." termina): ')
    cl += 1

    # final de palabra?
    if car == ' ' or car == '.':
        # ha terminado una palabra...

        # ...contarla solo si hubo al menos una letra...
        if cl > 1:
            ctp += 1

        # ...reiniciar contador de letras (por próxima palabra)...
        cl = 0

    else:
        # car es una letra... la palabra sigue...

        # ...contar la palabra si comienza con "p"...
        if cl == 1 and car == 'p':
            cpp += 1

# ...detección de la sílaba "ta"...
# ...por ahora, solo avisar si la ultima fue una "t"...
        if car == 't':
            st = True
        else:
            st = False

print('Cantidad de palabras:', ctp)
print('Cantidad de palabras que empiezan con "p":', cpp)
```

El primer paso (*mostrado en el esquema anterior*) "reacciona" al paso de una letra "t" cambiando el estado del flag *st* a *True* o *False* de acuerdo a si el valor cargado en *car* en ese momento es o no una "t".

El segundo paso es ajustar ese esquema para que ahora reaccione al posible paso de una "a" *inmediatamente luego* del paso de una "t". Eso puede hacerse con otro flag que llamaremos *sta* (por *señal de la sílaba "ta"*) y usando un poco de ingenio:

```

# contador total de palabras
ctp = 0

# contador de letras en una palabra
cl = 0

# contador de palabras que empiezan con p
cpp = 0

# contador de palabras que tuvieron "ta"
cta = 0

# flag: la ultima letra fue una "t"?
st = False

# flag: se ha formado la silaba "ta" al menos una vez?
sta = False

car = None
while car != '.':
    # cargar y procesar el caracter ingresado en car
    car = input('Letra (con "." termina): ')
    cl += 1

    # final de palabra?
    if car == ' ' or car == '.':
        # ha terminado una palabra...

        # ...contarla solo si hubo al menos una letra...
        if cl > 1:
            ctp += 1

        # si hubo "ta" contar la palabra...
        if sta == True:
            cta += 1

        # ...reiniciar contador de letras (por próxima palabra)...
        cl = 0

        # reiniciar flags (por próxima palabra)...
        st = sta = False

    else:
        # car es una letra... la palabra sigue...

        # ...contar la palabra si comienza con "p"...
        if cl == 1 and car == 'p':
            cpp += 1

        # ...deteccion de la silaba "ta"...
        if car == 't':
            st = True

        else:
            # hay una "a" y la anterior fue una "t"?...
            if car == 'a' and st == True:
                sta = True

            st = False

print('Cantidad de palabras:', ctp)

```

```
print('Cantidad de palabras que empiezan con "p":', cpp)
print('Cantidad de palabras que contienen "ta":', cta)
```

El esquema anterior resuelve el problema. Toda palabra que efectivamente contenga la sílaba "ta" será detectada marcando en *True* la bandera **sta**, y contando esa palabra en el contador **cta** cuando la palabra termine.

Asegúrese de entender cómo funciona este "*detector de la sílaba ta*" haciendo pruebas de escritorio detalladas con palabras como "tano" (deja **sta** en *True*), "tea" (deja **sta** en *False*), "patata" (deja **sta** en *True* y *obviamente la palabra es contada sólo una vez*) o "pala" (deja **sta** en *False*). El programa completo puede plantearse así:

```
__author__ = 'Catedra de Algoritmos y Estructuras de Datos'

# titulo general...
print('Detección de palabras con la expresión "ta"')
print('Versión 1: cargando los caracteres uno a uno...')

# inicialización de flags y contadores...
# contador de letras en una palabra...
cl = 0

# contador total de palabras...
ctp = 0

# contador de palabras que empiezan con "p"...
cpp = 0

# contador de palabras que tienen la expresión "ta"...
cta = 0

# flag: la última letra vista fue una "t"?...
sta = False

# flag: se ha formado la sílaba "ta" al menos una vez?...
sta = False

# instrucciones en pantalla...
print('Ingrese las letras del texto, pulsando <Enter> una a una')
print('(Para finalizar la carga, ingrese un punto)')
print()

# ciclo de carga - [1..N] (primera vuelta forzada)...
car = None
while car != '.':
    # cargar próxima letra y contarla...
    car = input('Letra (con punto termina): ')
    cl += 1

    # fin de palabra?
    if car == ' ' or car == '.':
        # 1.) contar la palabra solo si hubo al menos una letra...
        if cl > 1:
            ctp += 1

        # 2.) si hubo 'ta' contar la palabra...
        if sta:
            cta += 1
```

```

# reiniciar contador de letras...
cl = 0

# reiniciar flags...
st = sta = False

# regresar al ciclo ahora mismo...
continue

# 2.) detección de palabras que empiezan con "p"...
if cl == 1 and car == 'p':
    cpp += 1

# 3.) detección de expresión "ta"...
if car == 't':
    st = True
else:
    if car == 'a' and st:
        sta = True
    st = False

# análisis de los resultados finales...
print('1. Cantidad total de palabras:', ctp)
print('2. Cantidad de palabras que empiezan con "p":', cpp)
print('3. Cantidad de palabras con la expresión "ta":', cta)

```

Sólo queda por aclarar un detalle técnico menor: dentro del ciclo de carga, la rama verdadera de la condición que detecta un blanco o un punto para saber si ha terminado una palabra, contiene una instrucción *continue* al final:

```

# fin de palabra?
if car == ' ' or car == '.':
    # 1.) contar la palabra si hubo al menos una letra...
    if cl > 1:
        ctp += 1

    # 2.) si hubo 'ta' contar la palabra...
    if sta:
        cta += 1

    # reiniciar contador de letras...
    cl = 0

    # reiniciar flags...
    st = sta = False

    # regresar al ciclo ahora mismo...
    continue

```

Esta instrucción (como se verá con detalle en la Ficha 8) hace que se regrese inmediatamente a la cabecera del ciclo dentro del cual está incluida, sin ejecutar ninguna de las instrucciones que estuviesen escritas debajo de ella. En este caso, el uso de *continue* al terminar la rama verdadera hace que ya no sea necesario el *else* para esa condición, simplificando mínimamente la estructura del código fuente (al no tener que escribir *else* ni tener que indentar toda esa rama).

Si bien el programa anterior resuelve por completo el problema, subsiste un detalle referido a la interfaz de usuario que es bastante molesto e incómodo: los caracteres deben cargarse uno por uno y presionar <Enter> con cada uno... incluidos los espacios en blanco y el punto

final. La ejecución del programa para cargar sólo la frase "la tea." **produce una sesión de carga como la siguiente:**

```
Deteccion de palabras con la expresion "ta"
Version 1: cargando los caracteres uno a uno...
Ingrese las letras del texto, pulsando <Enter> una a una
(Para finalizar la carga, ingrese un punto)
```

```
Letra (con punto termina): l
Letra (con punto termina): a
Letra (con punto termina):
Letra (con punto termina): t
Letra (con punto termina): e
Letra (con punto termina): a
Letra (con punto termina): .
```

1. Cantidad total de palabras: 2
2. Cantidad de palabras que empiezan con "p": 0
3. Cantidad de palabras con la expresion "ta": 0

Y se puede ver que esto se complica aún más si el texto a procesar fuese mucho más largo o bien si el programa debiese ser ejecutado muchas veces para hacer pruebas. Por ese motivo podemos sugerir que el texto se cargue "todo junto" en una sola lectura por teclado, almacenándolo directamente en una variable de tipo cadena de caracteres. Una vez cargada esa cadena (que también **debe** finalizar con un punto para que sea correctamente detectada la última palabra), se puede procesar la secuencia mediante un *for iterador*, respetando el espíritu del enunciado en cuanto a analizar uno a uno los caracteres. La conversión del programa para adaptarlo a estas ideas, podría verse como sigue:

```
author = 'Catedra de Algoritmos y Estructuras de Datos'

# titulo general...
print('Deteccion de palabras con la expresion "ta"')
print('Version 2: cargando todo el texto en una cadena...')

# inicializacion de flags y contadores...
# contador de letras en una palabra...
cl = 0

# contador total de palabras...
ctp = 0

# contador de palabras que empiezan con "p"...
cpp = 0

# contador de palabras que tienen la expresion "ta"...
cta = 0

# flag: la ultima letra vista fue una "t"?...
st = False

# flag: se ha formado la silaba "ta" al menos una vez?...
sta = False

# carga del texto completo...
cadena = input('Cargue el texto completo (finalizando con un punto): ')

# ciclo iterador para procesar el texto...
```

```

for car in cadena:
    # contar la letra actual...
    cl += 1

    # fin de palabra?
    if car == ' ' or car == '.':
        # 1.) contar la palabra solo si hubo al menos una letra...
        if cl > 1:
            ctp += 1

        # 2.) si hubo 'ta' contar la palabra...
        if sta:
            cta += 1

    # reiniciar contador de letras...
    cl = 0

    # reiniciar flags...
    st = sta = False

    # regresar al ciclo ahora mismo...
    continue

# 2.) detección de palabras que empiezan con "p"...
if cl == 1 and car == 'p':
    cpp += 1

# 3.) detección de expresión "ta"...
if car == 't':
    st = True
else:
    if car == 'a' and st:
        sta = True
    st = False

# análisis de los resultados...
print('1. Cantidad total de palabras:', ctp)
print('2. Cantidad de palabras que empiezan con "p":', cpp)
print('3. Cantidad de palabras con la expresión "ta":', cta)

```

Como se puede ver, el esquema lógico es exactamente el mismo. El uso de una cadena en lugar de la lectura "carácter a carácter" hace más sencillo el manejo del programa al ejecutarlo, y por lo tanto también se simplifica el proceso de prueba. Pero por lo demás, desde el punto de vista lógico, es tan válida una técnica como la otra.

## 5.] Tipos generales de ciclos y su implementación en Python.

Hemos indicado en la *Ficha 6* que el *ciclo while* de Python es un ciclo de la forma  $[0, N]$  y hemos explicado que esto quería decir que el bloque de acciones podría ejecutarse entre 0 y  $N$  veces: si la condición de control es falsa en la primera evaluación, el bloque del ciclo no se ejecutará, pero se ejecutará una cantidad finita  $N$  de veces mientras la condición de control sea cierta.

Si bien el ciclo *for* de Python tiene una estructura general orientada hacia el recorrido o iteración de estructuras de datos como tuplas, rangos, cadenas o listas, puede verse con relativa sencillez que este ciclo *también es de la forma  $[0, N]$* : intuitivamente, la primera aproximación se basa en lo que ocurre cuando se usa un *for* para intentar recorrer una

secuencia vacía: el bloque de acciones no se ejecuta y el programa salta directamente a la primera instrucción ubicada a la salida del ciclo. Podemos verlo claramente en este ejemplo:

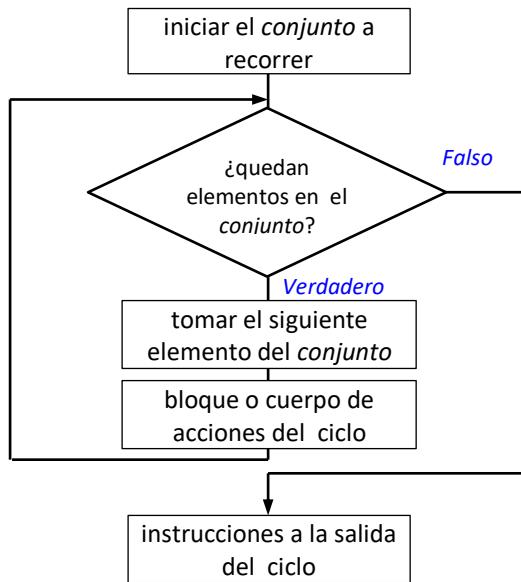
```
conjunto = ()
for x in conjunto:
    print('x:', x)

print('Programa terminado')
```

En el script anterior, la variable *conjunto* se inicializa como una *tupla vacía*, y luego se usa un ciclo *for* para recorrerla con la variable *x* como iteradora. En el bloque del ciclo se intentan mostrar los elementos de la tupla, uno a uno, y al terminar el ciclo se muestra un mensaje de terminación. Sin embargo, como la tupla *conjunto* está vacía, el ciclo no ejecuta nunca el bloque de acciones y sólo muestra el mensaje de terminación al final.

Sea cual fuese la estructura o colección a iterar, el ciclo *for* no ejecutará su bloque de acciones si la misma estuviese vacía. En el ejemplo anterior, si la variable *conjunto* se inicializa como una cadena vacía (*conjunto = ''*) o bien como un rango vacío (*conjunto = range(1, 0)*) el efecto será el mismo. En términos muy generales, el ciclo *for* entonces tiene una lógica esencial que puede graficarse de esta forma (que es claramente  $[0, N]$ ), usando la diagramación clásica:

**Figura 3: Diagrama de flujo de un ciclo *for* (diagramación clásica) como ciclo  $[0, N]$ .**



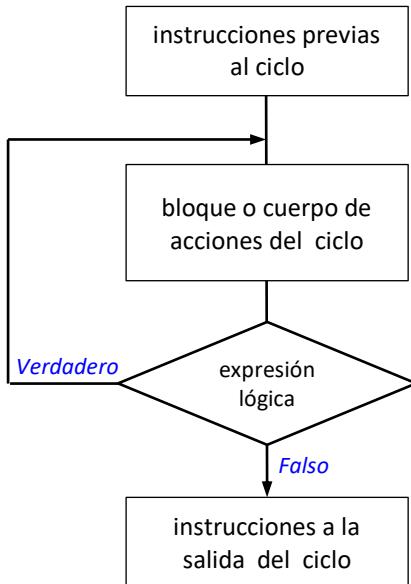
Como se puede ver, si el conjunto de datos a recorrer estuviese inicialmente vacío, la condición de control del ciclo entregaría un *falso* en el primer chequeo, ya que el conjunto no tendría en ese caso ningún elemento.

Por lo tanto, los dos tipos de instrucciones repetitivas provistas por Python son de la forma  $[0, N]$ . Sin embargo, existe al menos un segundo tipo de ciclo general (no provisto en forma directa por Python) que se conoce como ciclo  $[1, N]$ . En esencia, un ciclo  $[1, N]$  es aquel que *siempre ejecuta su bloque de acciones, al menos una vez* (y de allí la designación de  $[1, N]$ ).

Para que al menos una vez el bloque de acciones del ciclo se ejecute siempre, los ciclos de este tipo se plantean en forma ligeramente diferente a los ciclos  $[0, N]$ : en los  $[0, N]$  el

chequeo de la expresión lógica de control se escribe **antes** que el bloque de acciones, mientras que en los ciclos  $[1, N]$  se escribe **después** del bloque. El siguiente diagrama en forma clásica muestra la lógica esencial de funcionamiento de un *ciclo  $[1, N]$*  [2]:

Figura 4: Diagrama de flujo de un ciclo  $[1, N]$  (diagramación clásica).



El hecho es que no existe en Python una forma directa de ciclo  $[1-N]$  (como sería el *do – while* de C, C++ y Java o el *repeat – until* de Pascal), aunque eso no representa un problema ya que *siempre se puede emular un ciclo  $[1-N]$  desde un ciclo  $[0-N]$  forzando a que la expresión lógica de control sea cierta la primera vez que se evalúa*, como en el ejemplo siguiente, que carga una sucesión de números positivos hasta que se ingrese un número impar, mostrando al final la suma de los valores pares:

```

sp = 0

hecho = False
while hecho == False:
    n = int(input('Cargue un número (con impar corta:'))
    if n%2 == 0:
        sp += n
    else:
        hecho = True

print('La suma de los valores pares es:', sp)
  
```

Si bien el ciclo usado en el script anterior es un *while*, que en Python es  $[0, N]$ , el esquema de repetición que se muestra se comporta como  $[1, N]$  en la forma en que está planteado: la variable *hecho* se inicializa en *False* inmediatamente antes de iniciar el ciclo, y en la expresión lógica de control del ciclo se pregunta si *hecho* es *False*. Así escrito, *no hay ninguna oportunidad de que la expresión lógica sea falsa en el primer chequeo*, por lo que el bloque de acciones del ciclo se ejecutará inexorablemente al menos una vez. Si en la primera vuelta del ciclo se cargase por teclado un valor impar en la variable *n*, el valor de la variable *hecho* cambiaría a *True* y el ciclo cortaría... pero al menos una vez el bloque ya habrá sido ejecutado.

La aplicación de un tipo de ciclo u otro en un programa depende de las necesidades del programador: sabemos que si se conoce la cantidad exacta de repeticiones a realizar, suele

ser conveniente el uso de un *for*, pero nada obliga a ello ya que hemos visto que un *while* puede usarse también. Si se desconoce la cantidad de repeticiones, el *while* suele ser el ciclo más cómodo y esto es efectivamente así en Python, pero en otros lenguajes cualquier ciclo puede usarse en lugar de cualquier otro. Ahora sabemos que también puede plantearse un ciclo *while* para que se comporte en forma  $[1, N]$ . Y de nuevo, la decisión de hacer una u otra cosa depende del programador.

Sabiendo lo anterior, nada impide que nos preguntemos en qué situaciones reales un programador podría preferir la aplicación de un ciclo  $[1, N]$  en lugar de un  $[0, N]$ . Digamos que hay por lo menos una: la carga de valores desde el teclado, *validando esa carga* para impedir que se ingresen valores incorrectos.

Es muy común que en ciertas ocasiones se requiera que un programa cargue datos por teclado, pero de tal forma que se garantice que esos datos sean correctos. Por ejemplo: si en un programa se pide cargar el sueldo de un empleado, o la nota obtenida por un alumno en una materia, es de esperar que el valor que se cargue por teclado no sea negativo (un sueldo negativo o una nota negativa son situaciones que están claramente fuera de las hipótesis válidas para los datos del programa) [2].

La salida más cómoda es suponer que el usuario *no cargará* valores inadecuados, pero la solución más profesional es que el programa de alguna forma haga un control de los valores que se cargan (lo que forma parte de una estrategia que suele designarse como *programación defensiva*). Si el usuario carga un valor negativo cuando se esperaba que fuera cero o positivo, el programa *debería rechazar dicho valor, y volver a pedir la carga* del mismo hasta que finalmente el usuario introduzca un valor correcto. Ese proceso de verificación de un dato cargado por teclado, suele llamarse *proceso de validación de datos* y es usual realizarlo con un ciclo  $[1, N]$ : esto es así porque se espera que el usuario cargará *al menos un valor*. Si el primer valor cargado es correcto, el ciclo no pedirá otro valor y continuará el programa normalmente. Pero si el primer valor cargado fuera incorrecto, el ciclo hará otra repetición y volverá a pedir un valor, y así seguirá hasta que el valor sea correcto.

En el siguiente programa, se aplican *dos procesos de validación* para cargar por teclado el *sueldo* de un empleado y la *edad* de ese empleado, asegurando que ninguno sea negativo ni inapropiado. Ambos procesos validan la carga sólo terminan cuando el valor cargado finalmente se ingresa en forma correcta (aunque la lógica de cada validación es diferente en cada esquema):

```
__author__ = 'Catedra de AED'

print('Validación de carga de datos')

nombre = input('Ingrese su nombre: ')

edad = -1
while edad < 0 or edad >= 120:
    edad = int(input('Edad (mayor o igual a 0 y menor a 120, por favor): '))
    if edad < 0 or edad >= 120:
        print('Incorrecto... se pidió >= 0 y < 120... cargue otra vez...')

sueldo = 0
while sueldo <= 0:
    sueldo = int(input('Ingrese su sueldo (mayor a 0, por favor): '))
    if sueldo <= 0:
        print('Incorrecto... se pidió mayor a 0... cargue otra vez...')
```

```
print('Los datos registrados son: ')
print('Nombre:', nombre, '- Edad:', edad, '- Sueldo:', sueldo)
```

Ambos procesos de validación usan un ciclo *while* forzándolo a actuar en modo  $[1, N]$ . Para la carga de la *edad* se inicializa la variable *edad* con el valor -1 para que la condición de control del ciclo sea verdadera en el primer chequeo, mientras que en la carga del *sueldo* se inicializa la variable *sueldo* en 0 para lograr el mismo efecto. Dentro del bloque de acciones de cada ciclo se carga un valor por teclado para esas variables, los que serán controlados por cada ciclo, y si fuesen incorrectos se volverán a pedir. Para reforzar al usuario el hecho de haber cometido un error (y que no lo deje pasar en forma desapercibida), el bloque de acciones de cada ciclo contiene también una instrucción *if* que hace el mismo chequeo que el ciclo, pero en este caso sólo para activar un *print()* avisando del error e invitando al usuario a volver a ingresar. La repetición del proceso de carga es implementado por el ciclo, pero el mensaje de error surge de la instrucción condicional.

## Bibliografía

---

- [1] Python Software Foundation, "Python Documentation", 2021. [Online]. Available: <https://docs.python.org/3/>.
- [2] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

# Ficha 8

## Estructuras Repetitivas: Variantes

### 1.] Ciclos en Python: Variantes y elementos de control adicionales.

El bloque de acciones de un ciclo (*while* o *for*) en Python puede incluir una instrucción *break* para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control, en forma similar a otros lenguajes [1]. El siguiente script tiene el objetivo de cargar por teclado cinco números positivos y calcular la suma o acumulación de todos ellos. Pero el script contiene un *ciclo while* que se interrumpirá con una instrucción *break* si se carga por teclado un número cero o negativo, *aun cuando no se haya llegado a las 5 repeticiones que se esperaría en el ciclo*:

```
suma, i = 0, 1
while i <= 5:
    n = int(input('Ingrese un número mayor a cero: '))
    if n <= 0:
        break
    suma += n
    i += 1

print('La suma de los números ingresados es:', suma)
```

En el ejemplo anterior, el bloque de acciones del *ciclo while* carga por teclado un número *n*, y chequea con un *if* si ese número es cero o negativo. En caso de serlo, se ejecuta la instrucción *break* y su efecto es *cancelativo con respecto al ciclo*: el ciclo se interrumpe, y el programa continúa con la ejecución del *print()* que se encuentra a la salida del ciclo. Insistimos: se ejecuta el *break* y el ciclo se interrumpe, sin volver a la cabecera para chequear la expresión lógica de control del ciclo, por lo que su valor es ignorado en ese caso. Si el conjunto de números a procesar fuese {2, 4, 5, -2, 7, 1} este script cargaría los tres primeros y los acumularía sin problemas con el ciclo, pero el ciclo se interrumpiría en el cuarto (el -2) pues siendo negativo activaría la instrucción *break*. La salida de este programa para esa secuencia de números de entrada, sería:

La suma de los números ingresados es: 11

que es la suma  $2 + 4 + 5$ , sin incluir al resto de los positivos del conjunto.

El siguiente programa pretende procesar uno por uno los caracteres de una cadena. Contará aquellos que representen letras minúsculas, y por separado contará los que representen mayúsculas. Los caracteres que no sean letras serán simplemente ignorados. Pero si aparece un punto (".") el *ciclo for* que itera sobre la cadena se interrumpirá con un *break*, aun cuando no se haya alcanzado todavía el último carácter de la cadena:

```
cadena = input('Ingrese una cadena: ')
minusculas = mayusculas = 0

for car in cadena:
```

```

if car == '.':
    break

if 'a' <= car <= 'z':
    minusculas += 1

elif 'A' <= car <= 'Z':
    mayusculas += 1

print('Cantidad de minúsculas:', minusculas)
print('Cantidad de mayúsculas:', mayusculas)

```

Si la cadena ingresada por teclado fuese *ABCDabc34#.ABad* este programa mostraría una salida de la forma:

```

Ingrese una cadena: ABCDabc34#.ABad
Cantidad de minúsculas: 3
Cantidad de mayúsculas: 4

```

lo cual está de acuerdo con el objetivo: antes del *punto* hay cuatro letras mayúsculas y tres minúsculas. También hay dos números y un carácter numeral, pero tanto los números como el numeral son ignorados. Luego aparece el *punto* que da por terminado el *ciclo for* al ejecutar la instrucción *break*. Y todo carácter que viniese después del *punto*, será ignorado (sea letra o no) ya que el ciclo de iteración terminó.

De forma similar, pero a la inversa, un ciclo cualquiera puede incluir una instrucción *continue* para *forzar una repetición del ciclo* sin terminar de ejecutar las instrucciones que queden por debajo de la invocación a *continue* [1]. El siguiente ejemplo es una variante del que mostramos antes para cargar números positivos con un *while*, pero haciendo ahora que el ciclo *fuerce la siguiente vuelta* si el valor cargado fue cero o negativo (pero observe que entonces en este caso, el script *siempre* pedirá los cinco números positivos, incluso si en el medio se cargó alguno negativo o cero):

```

suma = 0
i = 1

while i <= 5:
    n = int(input('Ingrese un número mayor a cero: '))
    if n <= 0 :
        continue
    suma += n
    i += 1

print('La suma de los números ingresados es:', suma)

```

Al ejecutarse la instrucción *continue*, el ciclo no cortará su ejecución: volverá a la cabecera para volver a chequear la expresión lógica de control, pero no ejecutará en ese caso las instrucciones *suma += n* ni *i += 1*. Como de esta forma el contador *i* sólo se incrementa si entró un positivo, entonces ese ciclo se detendrá sólo si alguna vez se cargan cinco positivos.

A diferencia de otros lenguajes, en Python un ciclo (*for* o *while*) puede llevar opcionalmente una cláusula *else* en forma similar a una instrucción condicional, aunque el efecto de este *else* en un ciclo es bastante diferente a lo que ocurre con una condición común:

- ✓ En un ciclo *while*, las instrucciones de la rama *else* se ejecutan en el momento en que la expresión de control del ciclo se evalúa en *False* (sin importar en qué vuelta se obtuvo el *False*).
- ✓ En un ciclo *for*, las instrucciones de la rama *else* se ejecutan cuando el *for* termina de iterar sobre *todos* los elementos de la colección o secuencia dada.
- ✓ En ambos casos (tanto en un *while* como en un *for*) la rama *else* no será ejecutada si el ciclo terminó por acción de una instrucción *break*.

El siguiente programa es una variante del que mostramos antes para contar letras. Muestra el mismo ciclo *for* anterior, pero ahora con un *else*. Los resultados del conteo de letras se mostrarán sólo si el ciclo logró procesar toda la cadena (lo cual ocurrirá si la misma no tenía un punto): esas visualizaciones ahora están en el bloque *else* del *for*, por lo que sólo se ejecutarán si el ciclo cortó sin recurrir al *break*:

```

cadena = input('Ingrese una cadena: ')
minusculas = mayusculas = 0
for car in cadena:
    if car == '.':
        break

    if 'a' <= car <= 'z':
        minusculas += 1

    elif 'A' <= car <= 'Z':
        mayusculas += 1

else:
    print('Resultados (la cadena no contenía un punto):')
    print('Cantidad de minúsculas:', minusculas)
    print('Cantidad de mayúsculas:', mayusculas)

print('Proceso terminado')

```

Por último, veamos que en ocasiones el programador necesita *dejar vacío* el bloque de acciones de un ciclo o una rama de una condición y para casos así Python prevé el uso de la instrucción *pass*. Esa instrucción sirve para indicar al intérprete que simplemente considere vacío el bloque que la contiene:

```

# un ciclo que hace 10000 repeticiones sin bloque de acciones...
for i in range(10000):
    pass

```

En el ejemplo anterior, el ciclo *for* efectivamente hace 10000 repeticiones, pero no ejecuta ninguna acción adicional en cada una de ellas. El programa insumirá cierto tiempo en terminar de ejecutar este ciclo, y luego continuará normalmente con las instrucciones que sigan. Usar un ciclo con un elevado número de repeticiones pero con bloque de acciones vacío suele ser un truco empleado para provocar un breve retardo o *delay* en la ejecución del programa si el programador lo cree necesario.

La instrucción *pass* también puede usarse para dejar vacía una rama de una condición, como se ve en los siguientes ejemplos:

```

# una condición con rama verdadera en blanco...
if n1 > n2:
    pass

```

```

else:
    print('El primero no es el mayor')

```

## 2.] Programas controlados por menú de opciones.

Analizaremos ahora la forma de plantear lo que se conoce como un *programa controlado por menú de opciones*, que es en realidad la formalización de una técnica o esquema de *interfaz elemental de usuario*. Abordamos el tema a partir del siguiente enunciado:

**Problema 22.) Desarrollar un Programa Controlado por Menú de Opciones, que incluya opciones para realizar las siguientes tareas:**

1. *Cargar un valor entero n por teclado, y obtener la suma de los enteros del 1 al n.*
2. *Cargar un valor entero n por teclado, y obtener su factorial.*
3. *Cargar por teclado los coeficientes a, b, y c de un polinomio de segundo grado, y obtener el valor del polinomio en el punto x, siendo x un valor que también se carga por teclado.*

**Discusión y solución:** Un *programa controlado por menú de opciones* es aquel que, al comenzar, presenta en pantalla una lista de opciones (que los programadores designan justamente como un *menú de opciones*) para que el usuario del programa elija la acción que desea llevar a cabo. La mecánica básica es que al seleccionar una opción, el programa desarrolla la misma y luego *vuelve* a presentar el menú en pantalla, de modo que el usuario puede elegir otra opción (o la misma anterior si lo desea). El programa incluye una opción para finalizar la ejecución y *sólo finaliza si alguien selecciona esa opción de terminación* [2].

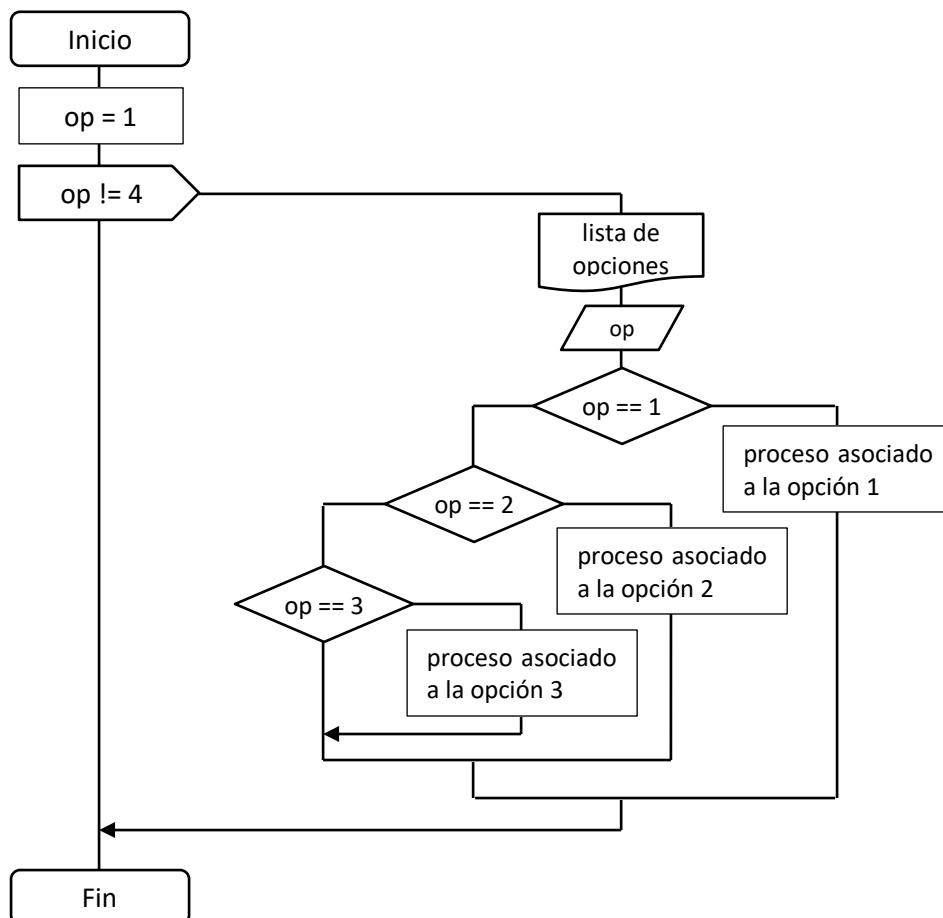
Si bien no es obligatorio, el hecho es que estos programas suelen usar un ciclo [1, N] para controlar todo el funcionamiento, debido a que la pantalla del menú principal debe mostrarse *al menos una vez* cuando el programa empieza, y luego *repetir* la presentación al terminar de desarrollar cada opción elegida por el usuario. Además, dentro del ciclo se utiliza un esquema de instrucciones condicionales anidadas, para chequear cuál fue la opción ingresada por el usuario y poder responder a ella con el proceso que corresponda.

Por lo tanto, nuestro programa usará un *ciclo while ajustado a operar en forma [1, N]* (esto es, planteado de forma tal que nos aseguremos que al chequear la condición de control la primera vez, esta sea verdadera) para controlar la repetición y un *esquema de condiciones if – elif* para determinar cuál es la opción elegida por el usuario.

El programa para el enunciado propuesto presentará una pantalla de *cuatro opciones*: las tres primeras (numeradas del 1 al 3) son las pedidas por el propio enunciado del ejercicio (y son llamadas en general *opciones operativas*) y la cuarta (numerada con el número 4) es la *opción de salida*. De este modo, el ciclo *while* controla que la opción ingresada sea distinta de 4, y en caso afirmativo vuelve a mostrar el menú. Sólo corta el proceso (y en este caso también el programa) cuando se selecciona la opción 4. Ante cualquier selección incorrecta, el programa vuelve a mostrar el menú, ignorando simplemente la entrada errónea. En la *Figura 1* (página 166) se muestra el diagrama de flujo general sugerido para el programa (sin detallar los procesos a realizar en cada rama).

Los procesos asociados a cada una de las opciones (salida verdadera de cada una de las condiciones del diagrama) son directos, y los analizaremos desde su planteo en código fuente, uno por uno.

Figura 1: Diagrama de flujo general sugerido para el programa controlado por menú de opciones.



En este momento, conviene aplicar un ya conocido detalle de control: Tanto para el cálculo de la suma (en la opción 1) como en el factorial (opción 2), se esperaría que el valor cargado por teclado para hacer el cálculo sea mayor o igual a 0. En ninguno de ambos casos sería admisible un valor negativo, y para controlar que eso no ocurra, podemos incluir un proceso de **validación** que realice la carga por teclado, pero verificando que el número cargado no sea negativo, pidiéndolo nuevamente en caso de serlo:

```

n = -1
while n < 0:
    n = int(input('Ingrese n (no negativo, por favor): '))
    if n < 0:
        print('Error... se pidió no negativo... cargue de nuevo...')
  
```

Ahora sí, comenzemos por el proceso asociado a la opción 1: de acuerdo al enunciado, si se elige la opción 1 se debe cargar por teclado un número entero  $n$ , y calcular la suma de los números enteros desde el 1 hasta el  $n$ . Por ejemplo, para  $n = 5$  la suma pedida es  $1 + 2 + 3 + 4 + 5 = 15$ . En principio, podría calcularse usando un ciclo *for* ajustado para recorrer el intervalo  $[1, n]$  con una variable  $i$ , y acumular los valores de  $i$ . Un proceso que aplique la idea podría verse así:

```

ac = 0
for i in range(1, n+1):
    ac += i
  
```

Sin embargo, el proceso indicado demoraría lo que demore el ciclo *for* en terminar de recorrer el *range* de control, y si bien eso puede parecer despreciable, el hecho es que para un valor realmente grande de *n* la demora en el tiempo de ejecución comenzará a notarse... incluso una computadora se las verá en problemas cuando deba ejecutar un proceso con una enorme cantidad de pasos.

Es conveniente, para una buena formación a futuro, que el programador se acostumbre a pensar todo el tiempo en soluciones mejores (en este caso, más rápidas)... incluso cuando podría parecer que en el contexto del problema no vale la pena el esfuerzo. En este caso, existe al menos una manera mucho más rápida de hacer el cálculo, recurriendo a una fórmula directa para calcular esa suma. Puede demostrarse (por inducción matemática) [3] que:

$$1 + 2 + 3 + 4 + \dots + (n-2) + (n-1) + n = n * (n+1) / 2$$

Sólo a modo de ejemplo, si *n* = 5 entonces  $1 + 2 + 3 + 4 + 5 = 5 * 6 / 2 = 30 / 2 = 15\dots$  y con este resultado a mano, ya no necesitamos perder el tiempo recorriendo un *range*, sino mostrar directamente el resultado del cálculo:

```
s = n * (n+1) // 2
print('Suma:', s)
```

La versión original del proceso que hacía la suma con un ciclo tiene un *tiempo de ejecución que crece en forma directamente proporcional al valor de n*. Pero la segunda versión se ejecuta en *tiempo constante*: no importa cuál sea el valor de *n*, el tiempo insumido en el cálculo es siempre el mismo.

El proceso a desarrollar para la opción 1 entonces quedar finalmente planteado así:

```
if opcion == 1:
    n = -1
    while n < 0:
        n = int(input('Ingrese n (no negativo, por favor): '))
        if n < 0:
            print('Error... se pidió no negativo... cargue de nuevo...')

    s = n * (n+1) // 2
    print('Suma de los enteros del 1 al', n, ':', s)
```

Respecto del proceso para la opción 2, el planteo es muy similar al de la opción 1, pero ahora se pide el *factorial* de *n* (y no la suma). El problema del *factorial* de *n* fue analizado y resuelto en una Ficha anterior, por lo que aquí simplemente reutilizaremos el proceso que propusimos en ese momento, con un único pequeño ajuste: el *range* a recorrer para obtener los factores del cálculo, está ajustado a *range(2, n+1)* en lugar de *range(1, n+1)*... ya que la multiplicación por 1 en realidad no aporta nada al producto final:

```
if opcion == 2:
    n = -1
    while n < 0:
        n = int(input('Ingrese n (no negativo, por favor): '))
        if n < 0:
            print('Error... se pidió no negativo... cargue de nuevo...')

    f = 1
    for i in range(2, n+1):
        f *= i

    print('Factorial de', n, ':', f)
```

Y en relación a la opción 3, el requerimiento es cargar por teclado los coeficientes  $a$ ,  $b$ , y  $c$  de un polinomio  $p$  de segundo grado, más el valor de  $x$  en el cual se quiere evaluar ese polinomio, y simplemente retornar el valor  $p(x)$ . No se requiere en este caso validar las cargas por teclado, ya que ahora son admisibles números negativos, cero o positivos. Aquí también el cálculo es directo:

```
if opcion == 3:
    a = float(input('a: '))
    b = float(input('b: '))
    c = float(input('c: '))
    x = float(input('x: '))
    p = a*pow(x, 2) + b*x + c
    print('Valor de p(', x, '):', p)
```

Mostramos a continuación (por fin...) el programa completo que incluye el ciclo del menú de opciones:

```
op = 1
while op != 4:
    # visualizacion de las opciones...
    print('1. Suma de 1 al n')
    print('2. Factorial de n')
    print('3. Polinomio valuado en x')
    print('4. Salir')
    op = int(input('Ingrese el numero de la opcion elegida: '))

    # chequeo de la opcion elegida...
    if op == 1:
        # Calculo de la suma de 1 a n...
        n = -1
        while n < 0:
            n = int(input('Ingrese n (>=0, por favor): '))
            if n < 0:
                print('Error... se pidio >=0... cargue de nuevo...')

        s = n * (n+1) // 2
        print('Suma de los enteros del 1 al', n, ':', s)

    elif op == 2:
        # Calculo del factorial de n...
        n = -1
        while n < 0:
            n = int(input('Ingrese n (>=0, por favor): '))
            if n < 0:
                print('Error... se pidio >=0... cargue de nuevo...')

        f = 1
        for i in range(2, n+1):
            f *= i

        print('Factorial de', n, ':', f)

    elif op == 3:
        # Calculo del valor de un polinomio...
        a = float(input('a: '))
        b = float(input('b: '))
        c = float(input('c: '))
        x = float(input('x: '))
        p = a*pow(x, 2) + b*x + c
        print('Valor de p(', x, '):', p)
```

### 3.] Implementación de un juego sencillo: el *Número Secreto*.

Está claro que uno de los campos de aplicación más populares de las computadoras es el desarrollo de juegos. Ese segmento del mercado informático mundial mueve miles de millones de dólares al año<sup>1</sup>. Nuestra humilde contribución a ese gigantesco mercado comenzará con la implementación del juego del *Número Secreto*, en base al siguiente enunciado general:

**Problema 23.)** *El Juego del Número Secreto consiste en lo siguiente: la computadora tiene un número guardado (que el jugador obviamente no conoce) y el jugador debe tratar de adivinarlo. Si lo logra, gana el juego y la computadora le avisa en cuántos intentos lo hizo. Si no lo logra en cierta cantidad predefinida intentos, el juego termina y se avisa que el jugador ha perdido. La cantidad máxima de intentos que el jugador tendrá a su disposición es un número que debe cargarse por teclado antes de comenzar a jugar, al igual que el límite derecho del intervalo que contendrá al número secreto elegido por el computador (es decir, el usuario debe poder indicar el número estará entre 1 y 30 o bien entre 1 y 50 o bien en el intervalo que el propio usuario decida). Desarrolle un programa completo que implemente este juego.*

**Discusión y solución:** La primera cuestión a resolver es cómo hacer que la máquina piense un número sin que lo sepa el jugador, y de forma tal que cada vez que el programa se ejecute nuevamente el número pensado sea diferente... (de lo contrario, el segundo o tercer jugador tendría muy fácil el juego... sólo debería saber qué número le tocó al anterior jugador...) Y es obvio que lo único que tenemos que hacer es generar ese número en forma aleatoria, con alguna de las funciones que Python provee para ello en sus librerías estándar, como las funciones *random.random()*, *random.randrange()* o *random.randint()* (ver Ficha 4) [1] [4]. En el juego a implementar, queríamos que el número a adivinar sea entero y en el intervalo pedido por el usuario (por ejemplo, entre 1 y 50). Y como sabemos, la forma más simple de obtener un número entero aleatorio en el intervalo  $[a, b]$  en Python, consiste en invocar a la función *random.randint(a, b)*. Por ejemplo, la invocación

```
x = random.randint(1, 50)
```

en Python, hará que la variable *x* sea asignada con un número aleatoriamente elegido entre 1 y 50 (ambos incluidos).

En cuanto al programa que implementa el juego, la idea final es que al inicio se muestre un mensaje indicando entre qué valores está el número secreto. Por ejemplo, suponga que el número secreto guardado por la computadora es el 14. Entonces una sesión de juego podría ser la siguiente, suponiendo que la cantidad máxima de intentos sea 5 y el límite derecho del intervalo elegido por el usuario sea 50:

```
El número está entre 1 y 50.  
[Intento: 1] ==> Ingrese: 30
```

<sup>1</sup> Son numerosas y muy conocidas las películas cuyo tema central gira en torno a la visión fantástica que la industria del cine tiene acerca de los juegos de computadoras. Una de las primeras (ya algo vieja...) fue la película *WarGames* (en español conocida como *Juegos de Guerra*) de 1983, protagonizada por un muy joven *Matthew Broderick* y dirigida por *John Badham*. En esa película se narra la historia de un joven hacker que casi provoca una guerra mundial termonuclear, pero que finalmente termina evitándola... haciendo que la supercomputadora que estaba por lanzar el ataque juegue contra si misma una eterna serie de partidas empatadas de Tic-Tac-Toe (o TaTeTi) hasta que la máquina "aprendió" que en algunas situaciones no hay ganadores... y lo mejor entonces es no hacer nada.

El número está entre 1 y 30.  
 [Intento: 2] ==> Ingrese: 10

El número está entre 10 y 30.  
 [Intento: 3] ==> Ingrese: 15

El número está entre 10 y 15.  
 [Intento: 4] ==> Ingrese: 14

Acertó!!! (en 4 intentos)

Un programa completo en Python que implementa este juego, se muestra a continuación:

```
import random

# Títulos y carga de datos básicos...
print('Juego del Número Secreto... Configuración Inicial...')
limite_derecho = int(input('El número secreto estará entre 1 y: '))
cantidad_intentos = int(input('Cantidad máxima de intentos: '))

# límites iniciales del intervalo de búsqueda...
izq, der = 1, limite_derecho

# contador de intentos...
intentos = 0

# bandera de estado: si es False, el
# número aún no ha sido encontrado...
encontrado = False

# el numero secreto...
secreto = random.randint(1, limite_derecho)

# el ciclo principal... siga mientras no
# haya sido encontrado el número, y la
# cantidad de intentos máxima no sea superada...
while not encontrado and intentos < cantidad_intentos:
    intentos += 1
    print('\nEl número está entre', izq, 'y', der)

    # un valor para forzar al ciclo a ser [1, N]...
    # ... ver Ficha 7.
    num = izq - 1

    # carga y validación del número sugerido por el usuario...
    while num < izq or num > der:
        num = int(input('[Intento: ' + str(intentos) + '] ==> Ingrese: '))
        if num < izq or num > der:
            println('Error... le dije entre', izq, 'y', der, '...')

    # controlar si num es correcto y avisar en ese caso...
    if num == secreto:
        encontrado = True

    # ... pero si no lo es, ajustar los límites
    # del intervalo de búsqueda...
    elif num > secreto:
        der = num
    else:
        izq = num
```

```
# control final...
if encontrado:
    print('\nGenio!!! Acertaste en', intentos, 'intentos')
else:
    print('\nLo siento!!! Acabaron los intentos. El número era:', secreto)
```

Al inicio del programa se cargan dos valores que permiten configurar el arranque del juego:

- ✓ **límite\_derecho**: el límite derecho del intervalo donde estará el número secreto.
- ✓ **cantidad\_intentos**: la máxima cantidad de intentos que tendrá disponible el jugador.

Ambos valores son cargados por teclado antes de comenzar el ciclo general que controla el juego. Ese ciclo es el *motor del juego* propiamente dicho: muestra los mensajes de ayuda en pantalla a través de los valores de *izq* (cuyo valor inicial es 1) y *der* (valor inicial: *límite\_derecho*). Dentro del ciclo se usa una *bandera de estado* llamada *encontrado* (valor inicial *False*) para marcar en todo momento si el número secreto fue encontrado o no.

El jugador carga un número en la variable *num* y se compara contra el número secreto (generado con *random.randint()* y almacenado en la variable *secreto*). Si *num* es igual a *secreto*, se marca ese hecho volviendo a *True* la bandera *encontrado*, y termina el juego. Si no, se determina si *num* es mayor o menor que *secreto*. Si fuera mayor, se ajusta el valor de *der* haciéndolo igual al número cargado (se reduce el intervalo de búsqueda desde la derecha). Y si fuera menor, se ajusta el valor de *izq* para hacerlo igual al número cargado (se reduce el intervalo de búsqueda desde la izquierda).

Un ciclo *while* forzado a ser  $[1, N]$  (vea el tema de los ciclos  $[1, N]$  en la Ficha 7) controla si el número fue adivinado o si se llegó al límite de intentos. En cualquiera de los dos casos, el proceso termina y se muestra el resultado adecuado en pantalla.

#### 4.] Otro juego sencillo: Piedra, Papel y Tijera.

Hemos dicho que la implementación de juegos de computadoras es una de las actividades más desarrolladas (y rentables) del mundo, y que varios miles de millones de dólares se invierten y se ganan en esta actividad año tras año (a pesar de las copias ilegales distribuidas por la piratería). En la sección anterior hemos realizado un sencillo aporte a este segmento con el *Juego del Número Secreto*, y ahora proponemos la implementación de un clásico juego de manos: El *Juego de Piedra, Papel y Tijera*. El enunciado formal puede ser el siguiente:

**Problema 24.) Desarrollar un programa que implemente el conocido juego de manos llamado "Piedra , Papel y Tijera". En este juego participan dos jugadores, uno contra el otro. Cada uno de ellos, al mismo tiempo que el otro, debe mostrar con una de sus manos, alguna de las tres figuras básicas llamadas Piedra (la mano cerrada), Papel (la mano abierta y extendida) o Tijera (los dedos de la mano formando una V). Luego se comparan las figuras que cada uno mostró, y se determina el ganador de acuerdo a la siguiente secuencia de reglas generales:**

- *Piedra vence a Tijera (ya que Tijera se rompe si intenta cortar a Piedra)*
- *Tijera vence a Papel (ya que Tijera corta a Papel)*
- *Papel vence a Piedra (ya que Papel envuelve a Piedra)*

*Típicamente, se juega "a la mejor de tres": los jugadores se enfrentan en tres jugadas, y se declara ganador al que gane en dos o más de esas tres.*

**Discusión y solución:** El primer desafío en este tipo de programas es decidir la forma en que será modelado el sistema de pantallas para que el programa informe al usuario sobre lo que está ocurriendo y la forma de hacer las cargas de datos cuando el programa lo requiera (es decir, se debe decidir el modelado de lo que se conoce como la *Interfaz de Usuario* del programa o *User Interface (UI)* en inglés). En general, si la *UI* contendrá elementos visuales de alto nivel, como ventanas, botones, gráficos, posibilidad de uso del mouse u otros elementos de interacción, entonces se habla (en español) de la *Interfaz Gráfica de Usuario (IGU)* o bien (en inglés) de la *Graphic User Interface (GUI)*.

Obviamente, en las primeras semanas de un curso de introducción a la programación no se cuenta aún con conocimientos y formación en cuanto al empleo de elementos visuales de alto nivel (que Python provee), por lo que nuestro diseño de interfaz de usuario estará simplemente basado en la consola estándar.

En nuestra versión del juego, supondremos que *uno de los jugadores será el usuario humano, y el segundo jugador será la computadora*. Los estudiantes podrán luego dedicarse a intentar modificar esta versión para incluir la posibilidad de partidas "humano-humano" o "computadora-computadora". La idea entonces será la siguiente: cuando deba jugar el humano, el programa deberá solicitar que cargue por teclado un número entero, cuyo valor identificará a la figura que el humano quiere jugar (por ejemplo: 1 – Piedra, 2 – Papel, 3 – Tijera). Los nombres de las tres figuras serán cadenas de caracteres en una tupla definida como una variable *descripcion* de uso general:

```
descripcion = 'Piedra', 'Papel', 'Tijera'
```

De este modo, el componente *descripcion[0]* queda asignado con la cadena '*Piedra*', mientras que *descripcion[1]* queda asignado con '*Papel*' y *descripcion[2]* con '*Tijera*'. Sabiendo que la variable *descripción* existe, entonces el siguiente esquema de código permite hacer la carga por teclado de la figura elegida por el jugador humano:

```
humano = int(input('Ingrese 1 - Piedra, 2 - Papel o 3 - Tijera: '))
print('Usted eligió:', descripcion[humano - 1])
```

En este esquema, la variable *humano* se usa para almacenar el valor cargado por el usuario, que será un 1, un 2 o un 3. Luego de cargar ese valor, se muestra un mensaje informando cuál fue efectivamente la figura elegida. Note que el número que cargó el usuario en la variable *humano*, se usa como índice para entrar a la cadena que describe a la figura dentro de la tupla *descripcion*, pero restando uno al valor (ya que en una tupla, los índices comienzan desde el valor 0 y no desde el 1) [1]. Obviamente, en la versión definitiva del programa la carga de la jugada del jugador humano será realizada con un proceso de validación, para evitar que se ingresen números diferentes de 1, 2, o 3.

Una vez que se tiene cargada la jugada del usuario humano, debe jugar la computadora. La elección de la figura será realizada en este caso en forma aleatoria mediante la función *random.randint()* que hemos presentado en la Ficha anterior. El sencillo par de instrucciones que sigue muestra el mecanismo:

```
computadora = random.randint(1, 3)
print('La computadora eligió:', descripcion[computadora - 1])
```

La variable *computadora* se usa para almacenar el número que representa a la figura elegida en forma aleatoria. Ese número se obtiene con la invocación *random.randint(1, 3)* que obtiene un número entero aleatorio en el intervalo [1, 3]. Igual que antes, se muestra luego el nombre de la figura elegida, con la misma técnica usada en la elección del jugador humano.

El paso siguiente es determinar si hubo un ganador, y en ese caso, cuál de los dos jugadores ganó en esa jugada. Recordando que las variables *humano* y *computadora* contienen el *número* de la figura elegida por cada jugador, el siguiente fragmento de código compara esos números y determina el ganador:

```
if humano != computadora:
    if (humano == 1 and computadora == 3) \
        or (humano == 3 and computadora == 2) \
        or (humano == 2 and computadora == 1):
        ganador = 1
    else:
        ganador = -1
else:
    ganador = 0
```

La variable *ganador* se usa para guardar el resultado, codificándolo de la siguiente forma: si hubo un empate, el resultado se codifica como un 0. Si en cambio ganó el jugador humano, el resultado se almacena en la variable *ganador* como un 1. Y si el ganador fue la computadora, se almacena el valor -1 en la misma variable *ganador*. Esta manera de proceder asociando cada estado posible a distintos números enteros, es muy común en programación y de hecho lo hemos aplicado ya en este mismo programa al enumerar las descripciones de la figuras.

La instrucción condicional que determina si ganó el usuario humano o el usuario computadora, está basada en los números de las figuras. Sabemos que Piedra es 1 y que Tijera es 3, por lo cual si la expresión:

```
humano == 1 and computadora == 3
```

fuese cierta, significaría que el humano (1: *Piedra*) le ha ganado en esa jugada a la computadora (3: *Tijera*). La condición evaluada es más extensa, ya que controla todas las combinaciones posibles de valores entre las variables *humano* y *computadora*, pero la lógica esencial es la misma que la que hemos descripto aquí. Confiamos en que los estudiantes podrán analizar el resto de esas combinaciones y asegurarse de que funcionan correctamente.

Note que en el planteo del código fuente la instrucción condicional completa aparece escrita en tres líneas diferentes. Esto es así debido a que la instrucción completa es tan larga que no cabe en una línea regular de 80 caracteres de ancho (que según las *recomendaciones PEP 8* debería ser el límite a respetar en una línea de código fuente). Si el programador decide partir una instrucción y seguir escribiendo la misma en la línea siguiente, se usa el separador \ (barra invertida) para hacer ese corte, en la forma que se mostró [1] [4].

Una vez que el programa ha determinado si hubo un ganador en una jugada, lo que sigue es llevar la cuenta del puntaje de cada jugador para poder al final determinar si hubo un ganador de la partida completa (recuerde: se hacen tres jugadas, y se declara ganador al que gane en dos o más de ellas). El script que sigue es el que lleva la cuenta:

```

if ganador == 1:
    contar_ganadas = contar_ganadas + 1
elif ganador == -1:
    contar_perdidas = contar_perdidas + 1

```

La idea general es simple: si la variable *ganador* quedó valiendo 1 entonces el ganador fue el humano y debe sumársele un punto. Pero si *ganador* quedó valiendo -1, el ganador de la jugada fue la computadora y el punto se debe sumar a ella. Las variables *contar\_ganadas* y *contar\_perdidas* son contadores que se usan para sumar los puntos que el humano ganó y perdió (respectivamente). Esas mismas variables permiten saber lo que ganó o perdió la computadora, tomándolas en forma invertida (lo que ganó el humano lo perdió la computadora, y lo que perdió el humano lo ganó la computadora: no es necesario usar dos variables adicionales). Obviamente, asumimos que el valor inicial de ambas variables misma es 0 (cosa que efectivamente haremos en el programa completo). Recuerde además, que la expresión *contar\_ganadas = contar\_ganadas + 1* es equivalente a *contar\_ganadas += 1*.

El script anterior entonces, está usando dos contadores para llevar la cuenta de los puntos ganados o perdidos por el humano. Es importante que se comprenda que si se va a usar un contador en un programa, ese contador debe ser asignado con algún valor inicial válido (normalmente el 0) para garantizar que el conteo comience desde donde debe hacerlo y no desde un valor indefinido, o bien para garantizar que esa variable realmente exista antes de comenzar a contar. En nuestro caso *el valor inicial 0* de cada una se asigna (en principio) en el *programa completo*, en el mismo lugar donde se define la variable *descripción*:

```

print('Bienvenido al juego de Piedra - Papel - Tijera')

# inicializacion de variables descriptivas y contadores...
descripcion = 'Piedra', 'Papel', 'Tijera'
contar_ganadas = contar_perdidas = 0

```

Con todos estos procesos ya definidos, podemos controlar totalmente *una* jugada o ronda. Pero el juego consta de *tres* rondas, *con resultado al mejor de tres*. La simulación de las tres rondas (una luego de la otra), puede hacerse sin problema mediante un *ciclo for que ejecute tres iteraciones*:

```

# Inicializacion de variables descriptivas y contadores...
descripcion = 'Piedra', 'Papel', 'Tijera'
ganadas = perdidas = 0

# Simulacion de las tres rondas...
for ronda in range(1, 4):
    print('\nRonda', ronda)

    # Juega el humano...
    humano = 0
    while humano < 1 or humano > 3:
        humano = int(input('Ingrese 1 - Piedra, 2 - Papel o 3 - Tijera: '))
        if humano < 1 or humano > 3:
            print('Error... se pidió entre 1 y 3... cargue de nuevo...')
    print('Usted eligió:', descripcion[humano - 1])

    # Juega la computadora...
    computadora = random.randint(1, 3)
    print('La computadora eligió:', descripcion[computadora - 1])

    # Determinar si hubo un ganador y mostrar...

```

```

if humano != computadora:
    if (humano == 1 and computadora == 3) \
        or (humano == 3 and computadora == 2) \
        or (humano == 2 and computadora == 1):
        ganador, mensaje = 1, 'Punto para el humano...'
    else:
        ganador, mensaje = -1, 'Punto para la computadora...'
else:
    ganador, mensaje = 0, 'Empate en esta ronda...'
print(mensaje)

# Llevar la cuenta de los puntos ganados o perdidos...
if ganador == 1:
    ganadas += 1
elif ganador == -1:
    perdidas += 1

```

El proceso comienza definiendo las variables más generales (la tupla con la descripción de las figuras, y los dos contadores). Luego se activa el ciclo *for* para repetir tres veces tres veces la ejecución de una ronda completa (con lo que se logra entonces la simulación de las tres rondas del juego) y antes de terminar simplemente controla si alguno de los dos jugadores sumó al menos dos puntos, mostrando un mensaje con el resultado. Note que según el enunciado, sólo hay un ganador si alguno de los jugadores hizo dos o más puntos: en casos como que un jugador gane una ronda y se empate en las otras dos, el programa anunciará *que no hay ganador*.

Para terminar, note que si el segmento de código anterior hace todo este trabajo, *entonces en el programa completo lo único que nos restaría por hacer es controlar el resultado final al terminar el ciclo, e informar al ganador*. Mostramos el programa completo a continuación:

```

import random

# Titulo general...
print('Bienvenido al juego de Piedra - Papel - Tijera')

# Inicializacion de variables descriptivas y contadores...
descripcion = 'Piedra', 'Papel', 'Tijera'
ganadas = perdidas = 0

# Simulacion de las tres rondas...
for ronda in range(1, 4):
    print('\nRonda', ronda)

    # Juega el humano...
    humano = 0
    while humano < 1 or humano > 3:
        humano = int(input('Ingrese 1 - Piedra, 2 - Papel o 3 - Tijera: '))
        if humano < 1 or humano > 3:
            print('Error... se pidió entre 1 y 3... cargue de nuevo...')
    print('Usted eligió:', descripcion[humano - 1])

    # Juega la computadora...
    computadora = random.randint(1, 3)
    print('La computadora eligió:', descripcion[computadora - 1])

    # Determinar si hubo un ganador y mostrar...
    if humano != computadora:
        if (humano == 1 and computadora == 3) \

```

```

        or (humano == 3 and computadora == 2) \
        or (humano == 2 and computadora == 1):
            ganador, mensaje = 1, 'Punto para el humano...'
        else:
            ganador, mensaje = -1, 'Punto para la computadora...'
    else:
        ganador, mensaje = 0, 'Empate en esta ronda...'
    print(mensaje)

# Llevar la cuenta de los puntos ganados o perdidos...
if ganador == 1:
    ganadas += 1
elif ganador == -1:
    perdidas += 1

# Determinación del resultado final del juego...
if ganadas >= 2:
    print('\nWinner!! Usted ganó', ganadas, 'a', perdidas)
elif perdidas >= 2:
    print('Loser... Usted perdió', perdidas, 'a', ganadas)
else:
    print('\nNo hay ganador... jueguen de nuevo para decidir')

# cierre...
print('Fin del programa')

```

## 5.] Medición del tiempo de ejecución de un proceso en Python.

Si bien los programas que hasta ahora hemos desarrollado en el curso no tienen la complejidad ni el volumen de datos como para esperar tiempos de ejecución elevados o grandes demoras, el hecho es que a medida que se avance y se incorporen nuevas herramientas y técnicas para el desarrollo de programas (como ahora las estructuras repetitivas y más adelante la recursividad) esa situación irá cambiando.

Muchos programas tenderán a demorar cada vez más su tiempo de ejecución a medida que el tamaño del conjunto de datos crezca. Y si bien existen técnicas matemáticas formales (que oportunamente veremos) para estudiar el comportamiento de un algoritmo en circunstancias diversas, por ahora nos quedaremos en un plano más técnico y aplicativo como es el de *medir directamente el tiempo de ejecución de un proceso cualquiera en Python*.

En general, todo lenguaje provee funciones en sus librerías estándar que de una forma u otra permiten medir el tiempo que demora en ejecutarse un proceso. En Python, las funciones estándar provistas para esta tarea (y para otras relativas al manejo del tiempo) se encuentran incluidas en el módulo (o *librería de funciones*) llamado *time* [1]. Por lo tanto, un programa que necesite acceso a esas funciones deberá contener una *instrucción de importación* para ese módulo:

```
import time
```

Para proceder a la medición del tiempo que demora en ejecutarse un proceso cualquiera, los lenguajes de programación suele basase en lo que se designa como el *momento de origen* o *epoch*, que depende del sistema operativo, pero que en general suele ser el 1 de Enero de 1970, a las 0 horas. Si quiere conocer con precisión cuál es el *epoch* en su sistema, puede usar la función *time.gmtime()* pasándole un 0 como parámetro:

```
import time
print(time.gmtime(0))
```

La función `time.gmtime(t)` retorna una colección de valores que representan la fecha dada por la cantidad de segundos  $t$  tomada como parámetro. Si  $t = 0$ , entonces la fecha pedida coincide con el *momento epoch*. El pequeño script anterior, mostrará la siguiente salida (que aquí se muestra en dos líneas por razones de espacio) [1]:

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=0,
tm_wday=3, tm_yday=1, tm_isdst=0)
```

De todos modos, y como veremos, no debe preocuparnos mucho cuál fue exactamente el *momento epoch*.

Hasta la versión 3.3 de Python se contaba con la función `time.clock()` que retornaba un número en coma flotante indicando la cantidad de segundos que transcurrieron entre el *epoch* del sistema hasta el momento en que se invocó a la función. Pero a partir de la versión 3.3 de Python, esa función fue desestimada (se dice que se marcó como *obsoleta* o *deprecated*) e incluso luego fue eliminada del módulo `time`.

A partir de la versión 3.3 de Python, se incluyó la función `time.perf_counter()` como una opción para reemplazar a `time.clock()`. La función `time.perf_counter()` retorna un valor flotante que indica la cantidad de segundos que representa en ese instante el estado del reloj del sistema, pero no referido al *epoch*, sino al momento en que el procesador fue reiniciado. Por lo tanto, `time.clock()` retornaba un indicador de tiempo referido a un momento *único y real* (el momento *epoch*), pero `time.perf_counter()` retorna un lapso de tiempo que es relativo al momento en que el procesador activó su contador de tiempo (y ese contador se reinicia cada vez que el procesador se vuelve a encender o se reinicia).

Como sea, en la práctica la técnica básica para medir el tiempo que demora en ejecutarse un proceso consiste en invocar dos veces a `time.perf_counter()`: la primera vez inmediatamente antes de lanzar el proceso, y la segunda, inmediatamente después de finalizado el mismo. El tiempo total de ejecución del proceso medido, será entonces igual a la diferencia entre esos tiempos. Veamos un ejemplo muy simple:

```
import time

# medición del tiempo del proceso...
t1 = time.perf_counter()
x = 3 + 4**2
t2 = time.perf_counter()

# obtención del tiempo total...
tt = t2 - t1

# visualización del tiempo final
print('Tiempo total:', tt)
```

En el programa anterior, el proceso cuyo tiempo de ejecución se quiere medir es la sencilla instrucción `x = 3 + 4**2`. Para hacer la medición, se invoca a `time.perf_counter()` justo antes de lanzar esa instrucción, y también justo después de terminar de ejecutarla, almacenando los resultado en las variables `t1` y `t2`. La primera (`t1`) nos dice entonces cuánto tiempo (en segundos) transcurrió en el sistema hasta la primera llamada a `time.perf_counter()`; y la segunda (`t2`) nos indica el tiempo transcurrido hasta la segunda invocación. Por lo tanto, el cálculo de la diferencia `tt = t2 - t1` nos dará el tiempo exacto transcurrido entre ambas

llamadas, que no es otra cosa que el tiempo que demore en ejecutarse el proceso  $x = 3 + 4^{**2}$ . En este caso, la ejecución del programa producirá una salida parecida a la siguiente:

```
Tiempo total: 4.105230186414398e-07
```

que como veremos en una Ficha posterior, equivale a un tiempo de  $4.105230186414398 * 10^{-7}$  segundos, que es lo mismo que 0.0000004105230186414398 segundos. Aquí debe entender que este número muy posiblemente será diferente cuando haga sus propias pruebas, ya que el tiempo de ejecución dependerá de muchos factores (el tipo de procesador, el nivel de ocupación del mismo al momento de ejecutar el script, etc.), pero puede esperar que el orden de magnitud del número obtenido sea muy similar (7 u 8 dígitos de mantisa con valor 0).

La técnica general para medir el tiempo de un proceso es esencialmente la que hemos mostrado, pero podemos ir un poco más lejos. Por lo pronto, está claro que el proceso a medir no tiene por qué ser una instrucción simple: puede medirse (por ejemplo) el tiempo de ejecución de una llamada a una función o de cualquier segmento de programa que pueda incluir entre dos llamadas a `time.perf_counter()`. El programa que sigue es una variante simple del programa anterior, y permite medir el tiempo de ejecución de un proceso que calcula y muestra el mayor entre dos números  $n1$  y  $n2$ :

```
import time

n1, n2 = 20, 10

# medición del tiempo del proceso...
t1 = time.perf_counter()
if n1 > n2:
    m = n1
else:
    m = n2
t2 = time.perf_counter()

# visualización del resultado del proceso...
print('Mayor:', m)

# obtención del tiempo total...
tt = t2 - t1

# visualización del tiempo final
print('Tiempo total:', tt)
```

La ejecución del programa produce una salida similar (en orden de magnitud) a la que mostramos aquí:

```
Tiempo total: 8.210460372828794e-07
```

Se puede apreciar que ahora la cantidad de dígitos con valor 0 a la derecha del punto sigue siendo 7, pero el primer decimal significativo vale ahora 8 (en lugar del 4 del ejemplo anterior). Conclusión: en este caso puntual, la ejecución de un proceso completo que incluye a una condición, le llevó a Python *el doble de tiempo* que ejecutar una instrucción simple que asigne en una variable el resultado de una expresión aritmética. Esto era esperable: no es lo mismo una sola instrucción simple, que una instrucción compuesta que incluye el chequeo de una condición y luego una asignación.

Aún más amplio resulta considerar la función `eval()` de la librería estándar de Python [1]. En su forma más simplificada, esta función toma como parámetro una *cadena de caracteres que exprese una instrucción* válida en Python, analiza esa expresión, y si efectivamente es válida la ejecuta y retorna el resultado de esa expresión. Ejemplo:

```
x = eval('3 + 4**2')
print('x:', x)
```

La salida en consola del script anterior será algo como `x: 19`.

De esta forma, podemos plantear un script que tome una cadena cualquiera (asignada o cargada por teclado) pero que represente una expresión válida en Python, y retorne el tiempo de ejecución de la expresión:

```
import time

expr = 'min(3, 7)'
t1 = time.perf_counter()
r = eval(expr)
t2 = time.perf_counter()
print('Menor:', r)
print('Tiempo de ejecución:', t2 - t1)

expr = input('Ingrese una expresión válida en Python: ')
t1 = time.perf_counter()
r = eval(expr)
t2 = time.perf_counter()
print('Resultado:', r)
print('Tiempo de ejecución:', t2 - t1)
```

Si se ejecuta el script anterior, comenzará mostrando una salida de la parecida a esta:

```
Menor: 3
Tiempo de ejecución: 3.489491498982916e-05
```

Luego pedirá que se cargue por teclado una expresión válida en Python. Si el usuario carga una cadena cualquiera que efectivamente represente una expresión ejecutable en Python (como puede ser la expresión lógica `'3 >= 8'`) entonces el programa ejecutará esa expresión, mostrará el resultado de la misma (*False* en este caso...) y luego el tiempo insumido en esa ejecución:

```
Ingrese una expresión válida en Python: 3 >= 8
Resultado: False
Tiempo de ejecución: 7.389511409705563e-05
```

## Bibliografía

- [1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [2] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [3] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.
- [4] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 9

## Subproblemas y Funciones

### 1.] Introducción al concepto de subproblema.

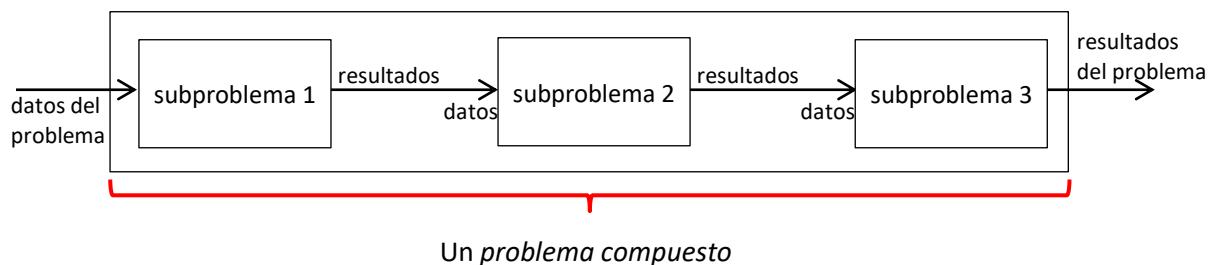
La mayoría de los problemas que normalmente se enfrentan en la práctica son de *estructura compuesta*, es decir, son problemas que pueden ser divididos en subproblemas cada vez menores en complejidad, hasta llegar a subproblemas que no necesiten nuevas subdivisiones (y que por eso se designan como *problemas simples*). Básicamente, un *subproblema* es un problema incluido dentro de otro de estructura más compleja.

Un principio básico en el planteo de algoritmos para resolver problemas, consiste justamente en tratar de identificar los subproblemas simples de un problema compuesto, considerando que cada subproblema simple es también un problema que admite datos, desarrolla procesos y genera resultados.

Esta forma básica de proceder, *centrando el análisis de un problema en los subproblemas que pudiera contener*, ha dado origen a la técnica o paradigma de programación que usaremos y que generalmente se conoce como *programación estructurada*<sup>1</sup>, en la que justamente se trabaja de forma que los programadores descomponen un problema en problemas menores, programan por separado los procesos que resuelven esos subproblemas, y luego unen todo para formar el programa completo [1].

Idealmente, y en forma muy general, un problema compuesto se dividirá en tantos subproblemas como se hayan detectado de forma que cada uno de ellos requerirá datos y entregará resultados. Los resultados que cada uno genere, serán datos para los subproblemas siguientes o ya serán los resultados finales esperados, como lo que se ve en la Figura 1 en la cual se supone un problema compuesto con tres subproblemas incluidos en él.

Figura 1: Esquema general de un *problema compuesto* que incluye tres *subproblemas*.



<sup>1</sup> Un conjunto de reglas y convenciones para trabajar en cierto contexto se designa en general como un *paradigma*. En consecuencia, la técnica de Programación Estructurada se puede designar también en como el *Paradigma de Programación Estructurada*, en contraposición a otras técnicas y formas de trabajo que constituyen otros paradigmas, como el *Paradigma de Programación Orientada a Objetos* o el *Paradigma de Programación Funcional*, entre otros.

La idea de esa figura, (insistimos: muy general y esquemática) es que los datos del problema compuesto original serán tomados a su vez como datos por alguno de sus subproblemas, el cual los procesará y obtendrá ciertos resultados parciales, que serán entregados como datos al siguiente subproblema. En algún momento, alguno de los subproblemas obtendrá y entregará los resultados finales que se esperaban para el problema compuesto original. Los procesos planteados en cada subproblema se pueden considerar independientes entre ellos, pero se terminan relacionando por esta secuencia de resultados – datos compartidos.

Está claro que en una situación real, este esquema puede variar y complicarse mucho más: los datos originales del problema compuesto podrían ser tomados por dos o más subproblemas (y no por uno sólo); los resultados finales podrían llegar a ser obtenidos por más de un subproblema (y no sólo por uno); y los resultados parciales de un subproblema podrían ser tomados como datos o entradas por más de un subproblema posterior (y no sólo por uno...) O incluso más: cualquiera de los subproblemas podría a su vez ser compuesto, dividiéndose en dos o más subproblemas él mismo. Lo que intenta rescatar la *Figura 1* es el fundamento conceptual básico: un problema compuesto contiene subproblemas que aprovechan mutuamente los resultados parciales obtenidos para llegar en algún momento al resultado final esperado.

A modo de ejemplo que permita aclarar el tema, analicemos ahora el siguiente problema de aplicación (la lógica requerida para resolverlo es muy simple, ya que la idea es sólo usarlo como modelo para entrar en el tema de los subproblemas):

**Problema 25.)** *En el contexto de un estudio estadístico, se requiere un programa que cargue tres números por teclado y luego proceda a determinar el menor de ellos y calcular el cuadrado y el cubo del mismo.*

a.) Identificación de componentes:

- **Resultados:** El menor, su cuadrado y su cubo. (*men, cuad, cubo: int*)
- **Datos:** Tres números distintos. (*a, b, c: int*)
- **Procesos:** El problema puede ser dividido en dos *subproblemas*: uno de ellos tiene como objetivo calcular el *menor de los tres valores de entrada*, y el segundo busca *calcular el cuadrado y el cubo de ese valor*. En vez de intentar realizar todo a la vez, lo cual crearía eventuales confusiones, se trata de plantear cada subproblema por separado, y luego unir las piezas para obtener el algoritmo completo que permita desarrollar un programa.

Como cada subproblema es él mismo un problema, entonces cada uno tiene su propio modelo de datos, procesos y resultados. Y en ese sentido, el planteo de cada subproblema puede hacerse como sigue:

**Subproblema 1:** *Determinar el menor de los tres valores de entrada.*

- **Resultados:** El menor de tres números. (*men: int*)
- **Datos:** Tres números. (*a, b, c: int*)
- **Procesos:** El proceso de este subproblema consiste en plantear el esquema de condiciones que permita encontrar el menor entre los valores de las variables *a, b, y c*, lo cual puede hacer en base al siguiente modelo de pseudocódigo:

```
si   a < b   y   a < c:
    men = a
sino:
```

```

    si b < c:
        men = b
    sino:
        men = c

```

**Subproblema 2:** *Calcular el cuadrado y el cubo del menor encontrado.*

- **Resultados:** El cuadrado y el cubo del menor. (*cuad, cubo: int*)
- **Datos:** El menor de los valores de entrada. (*men: int*)
- **Procesos:** En este subproblema, todo el trabajo consiste en calcular el cuadrado y el cubo del menor:

```

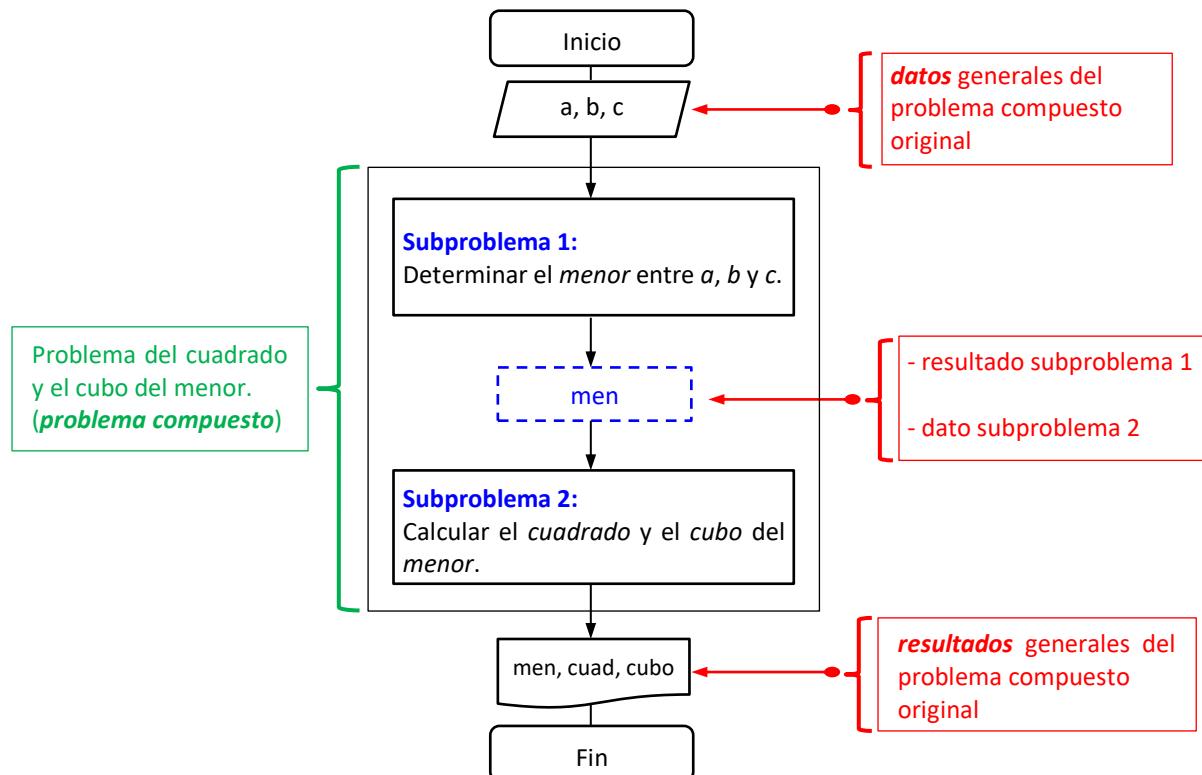
cuad = men ** 2
cubo = men ** 3

```

Aquí debe notarse que este subproblema toma como dato al valor *men* que fue el resultado del subproblema anterior. Pero eso no significa que el valor de *men* debe ser cargado por teclado: su valor surge de los procesos del subproblema anterior, en particular de las condiciones y asignaciones usadas para calcular el valor menor.

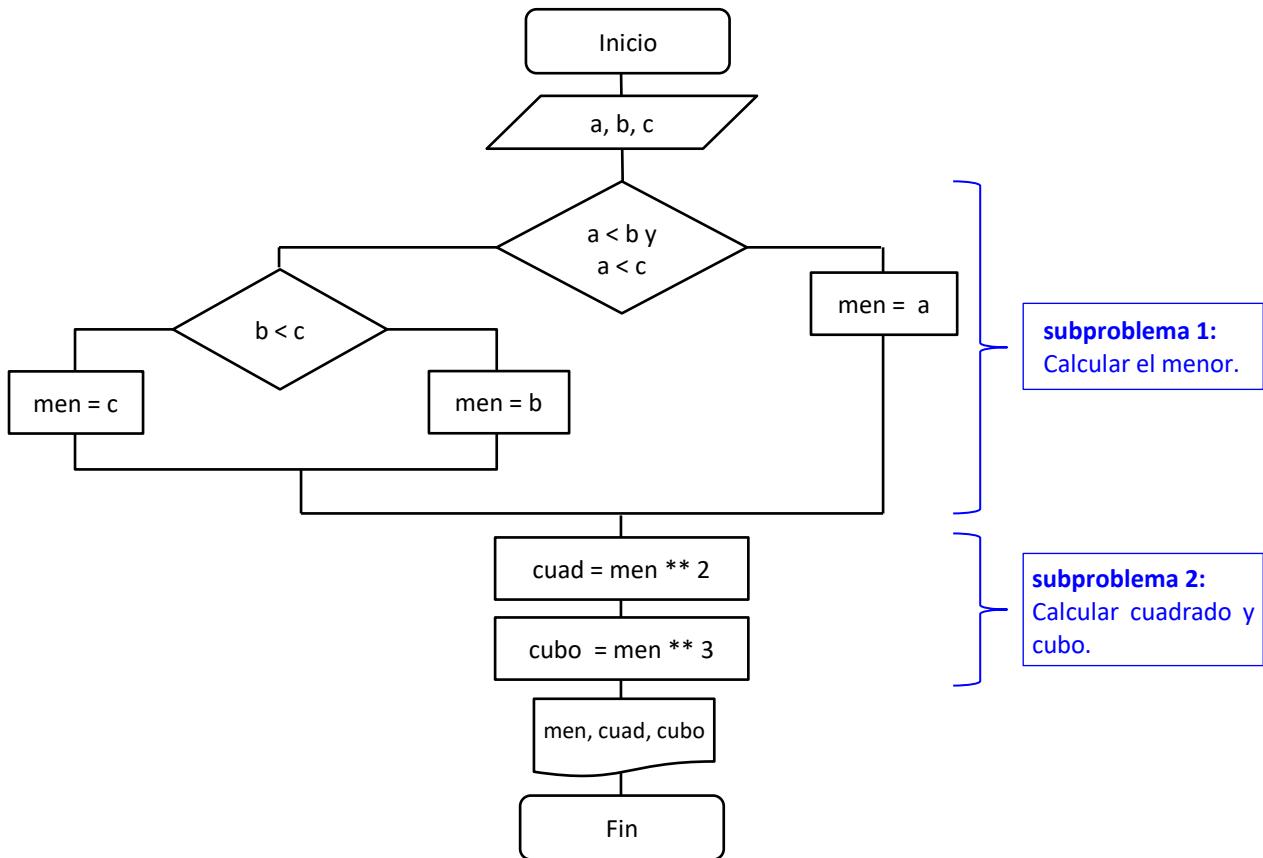
Es fácil ver que con estos dos subproblemas detectados y analizados, la estructura gráfica completa del problema original puede representarse a través del modelo que se ve en la *Figura 2*:

**Figura 2:** Modelo de subproblemas para el *problema del cuadrado y el cubo del menor*.



- b.) **Planteo del algoritmo:** A partir de la discusión hecha en la identificación de componentes, y tomando como base el modelo de subproblemas de la *Figura 2*, se puede plantear un diagrama de flujo que contemple implícitamente a esos subproblemas, los cuales se muestran *remarcados con llaves de color azul* para facilitar su identificación (ver *Figura 3*):

Figura 3: Diagrama del flujo del problema del cubo y el cuadrado del menor, marcando subproblemas.



c.) Desarrollo del programa: Como siempre, en base al diagrama el script o programa en Python se deduce en forma inmediata:

```

__author__ = 'Cátedra de AED'

# títulos y carga de datos...
print('Determinación del cuadrado y el cubo del menor')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))
c = int(input('Tercer número: '))

# procesos...
# subproblema 1: determinar el menor...
if a < b and a < c:
    men = a
else:
    if b < c:
        men = b
    else:
        men = c

# subproblema 2: calcular el cuadrado y el cubo del menor...
cuad = men ** 2
cubo = men ** 3

# visualización de resultados...
print('El menor es:', men)
print('Su cuadrado es:', cuad)
print('Su cubo es:', cubo)

```

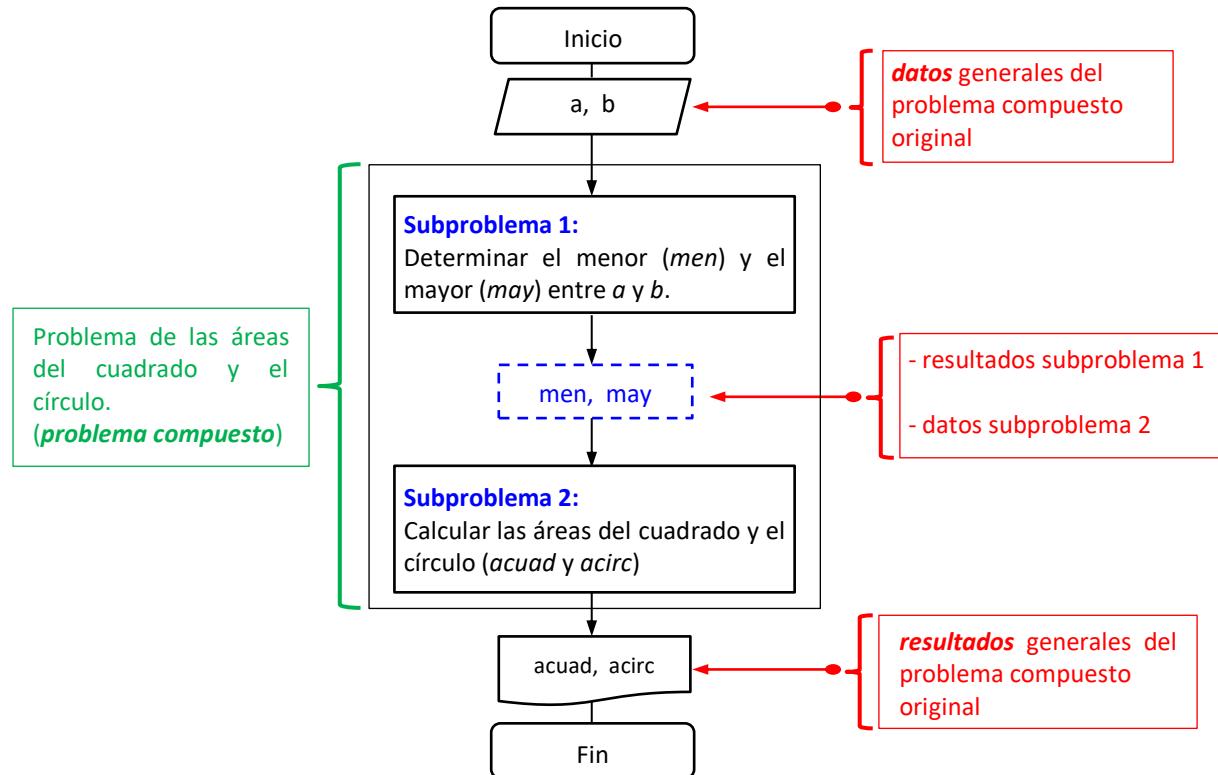
Como ya habrá podido notar, a medida que se estudian problemas de estructura más compleja y se incorporan nuevos elementos (como instrucciones condicionales, instrucciones repetitivas y subproblemas), se hace también cada vez más extenso el análisis del problema a nivel de identificación de datos, resultados y procesos. Hemos indicado en una ficha anterior que en la práctica, el programador hace gran parte de este proceso mentalmente, sin tanto rigor descriptivo previo. Y eso es lo que en general hemos hecho en cada una de las Fichas anteriores: para cada problema, discutir una brevíssima identificación de datos y resultados, para pasar luego al planteo del diagrama de flujo o el pseudocódigo., y si fuese necesaria una discusión referida a ciertos aspectos lógicos de la solución, se incorpora brevemente antes de plantear el diagrama [1].

Considere entonces un nuevo ejemplo muy simple de aplicación:

**Problema 26.)** *Se cargan por teclado dos números. Calcular la superficie de un cuadrado, suponiendo como lado del mismo al mayor de los números dados y la superficie de un círculo suponiendo como radio del mismo al menor de los números dados.*

**Discusión y solución:** En este problema también se presentan dos subproblemas: en el primero se debe buscar el mayor y el menor entre dos valores, y en el segundo se deben calcular las áreas indicadas. Como el cálculo de las áreas no puede hacerse sin conocer cuál de los valores es el mayor y cuál es el menor, es obvio entonces que el subproblema de buscar el mayor y el menor debe resolverse antes que el cálculo de las áreas. Un esquema general de subproblemas podría ser el siguiente:

Figura 4: Modelo de subproblemas para el problema de las áreas del cuadrado y el círculo.



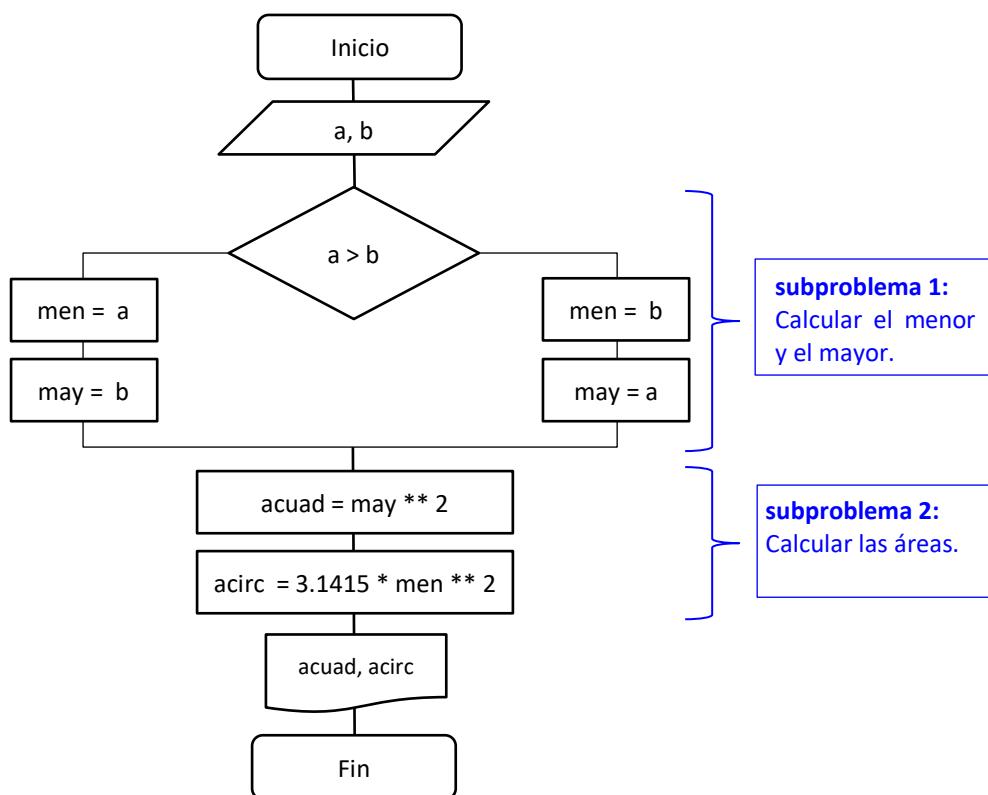
El *subproblema 1* tomará los valores de las variables *a* y *b* cargados por teclado, determinará el menor y el mayor y los copiará a su vez en las variables *temporales men* y *may*. De esta forma, si luego se requiere volver procesar estos valores en diferentes lugares del programa,

no será necesario preguntar nuevamente cuál es el mayor y cuál el menor: a partir del ordenamiento de variables producido por este subproblema, el programa trabajará con las variables *men* y *may* en lugar de *a* y *b*.

El *subproblema 2* simplemente debe calcular las dos áreas. El enunciado del problema indica que debe suponerse un cuadrado cuyo lado sea igual al mayor de los números de entrada (que tenemos copiado en *may*) y debe suponerse también un círculo cuyo radio sea igual al menor de esos datos (que tenemos en *men*). Por lo tanto, aplicando fórmulas muy conocidas de la geometría, el área del cuadrado (*acuad*) será  $acuad = may ** 2$  (el cuadrado del lado) y el área del círculo (*acirc*) será  $acirc = 3.1415 * men ** 2$  (*Pi* por radio al cuadrado).

El diagrama de flujo puede verse en la *Figura 5*:

**Figura 5: Diagrama de flujo del problema de la superficie del cuadrado y del círculo.**



- c.) **Desarrollo del programa:** El script en Python no presenta dificultades. Sólo recuerde la importancia de respetar la indentación, que en el caso de las ramas de la condición en este caso es fundamental [2]:

```

__author__ = 'Cátedra de AED'

# títulos y carga de datos...
print('Cálculo de áreas de un cuadrado y un círculo...')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))

# procesos...
# subproblema 1: determinar el menor y el mayor...
if a > b:
    men = b
    may = a

```

```

else:
    men = a
    may = b

    # subproblema 2: calcular las áreas...
    acuad = may ** 2
    acirc = 3.1415 * men ** 2

    # visualización de resultados...
    print('Área del cuadrado:', acuad)
    print('Área del círculo:', acirc)

```

## 2.] Subrutinas: introducción y conceptos generales.

Como vimos, a partir de la noción de *subproblema* un problema principal puede descomponerse en partes o subproblemas más sencillos para facilitar su planteo y luego unir las piezas para lograr el planteo final. Sin embargo, en la forma en que aquí se trabajó, la división en subproblemas resultó ser un simple planteo de tipo mental: ni el diagrama de flujo del problema ni el programa o script hecho en Python muestran en forma clara cuales son los subproblemas en los que *se supone* está dividido el problema (a lo sumo, hemos incluido llaves y etiquetas de colores para remarcar la presencia de cada subproblema, pero sólo a modo de recurso informal). Y esa es la cuestión que aquí analizamos: se asume que quien estudie el diagrama de flujo y el programa en Python, comprende (o al menos intuye) la división en subproblemas que el programador definió.

Pero ¿por qué debe ser así? ¿No pueden plantearse el diagrama de flujo y el programa de forma tal que los subproblemas en que fueron divididos queden evidenciados *físicamente*, y no ya *intuitivamente*? En otras palabras: ¿cómo puede representarse un subproblema en un diagrama de flujo? ¿y en un programa hecho en Python? La respuesta a ambas cuestiones es usar lo que genéricamente se denominan *subrutinas*, y que se implementan mediante recursos que se designan de distintas maneras en cada lenguaje. En Python, las subrutinas se implementan mediante *funciones* [3].

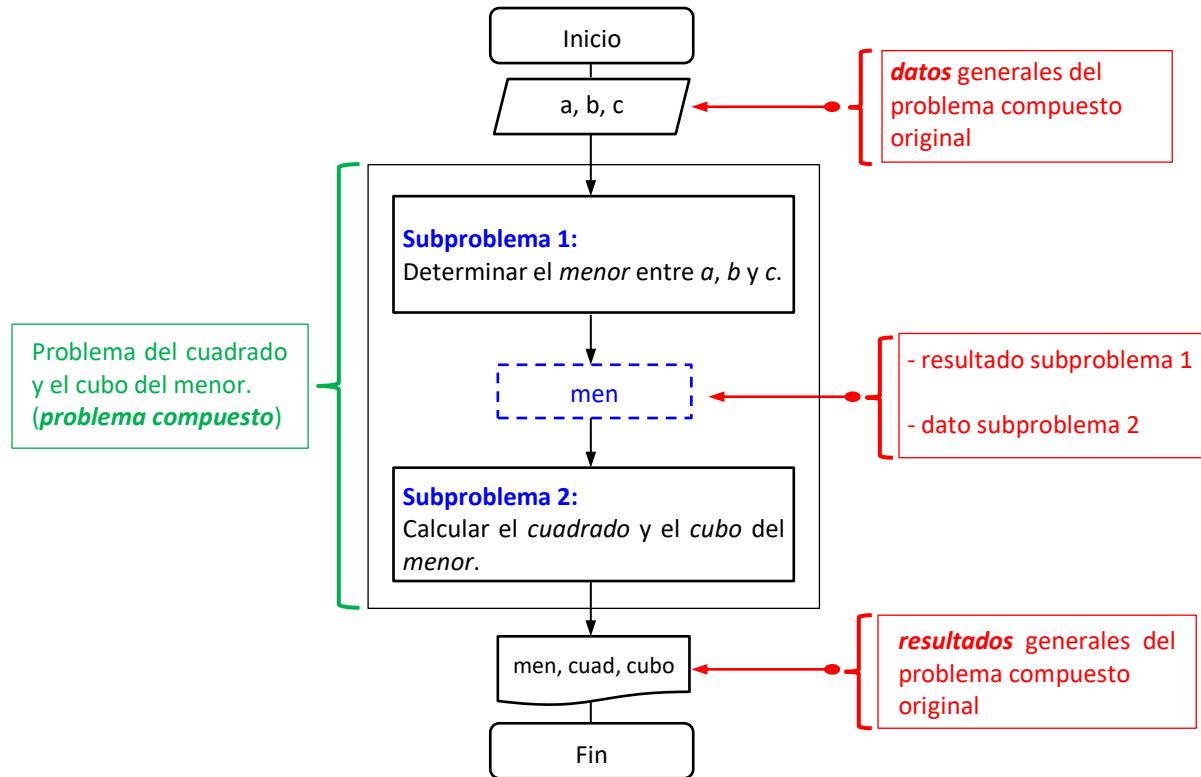
Esencialmente, una *subrutina* (o *subproceso*) es un segmento de un programa que se escribe en forma separada del programa principal. A cada subrutina el programador asocia un nombre o identificador y mediante ese identificador la subrutina puede ser activada todas las veces que sea necesario. Esto puede parecer complicado pero piense el estudiante que, aún sin saberlo, ya ha estado usando subrutinas: la función *print()* que utilizaba para mostrar salidas por consola, es una subrutina que ya viene lista para usar con el lenguaje. En ese sentido, como ya se sugirió, una *función* es una *subrutina* escrita en Python. También son ejemplos de subrutinas implementadas como funciones en Python [3] algunas otras conocidas, como *int()*, *float()*, *input()*, *pow()*, etc.

El lenguaje Python ya provee una serie de subrutinas listas para ser usadas, en forma de funciones incluidas en varias *librerías de funciones*, pero la idea ahora es que el programador desarrolle sus propias subrutinas, en forma de funciones propias definidas y escritas por el propio programador. Aunque lo que sigue no es una regla taxativa, lo que básicamente suele hacerse en el *Paradigma de la Programación Estructurada*, es definir una subrutina (en Python, una función) por cada subproblema que el programador detecte.

A modo de ejemplo podemos volver sobre el *problema 25* de esta misma Ficha, cuyo enunciado era básicamente: *cargar tres números por teclado, determinar el menor de ellos y*

calcular el cuadrado y el cubo del mismo. Como se vio oportunamente, este problema incluía dos subproblemas: el primero era determinar el menor, y el segundo era calcular el cuadrado y el cubo de ese menor. El gráfico que sigue en la *Figura 6*, es el mismo que se mostró en la *Figura 2*, y expone la estructura general de subproblemas que se sugirió para el planteo de la solución:

**Figura 6: Modelo de subproblemas - Problema del cuadrado y el cubo del menor.**



En base a lo anterior, el algoritmo general contendría entonces dos *subrutinas*: una para calcular el mayor y el menor entre los dos números de entrada, y otra para calcular el cuadrado y el cubo.

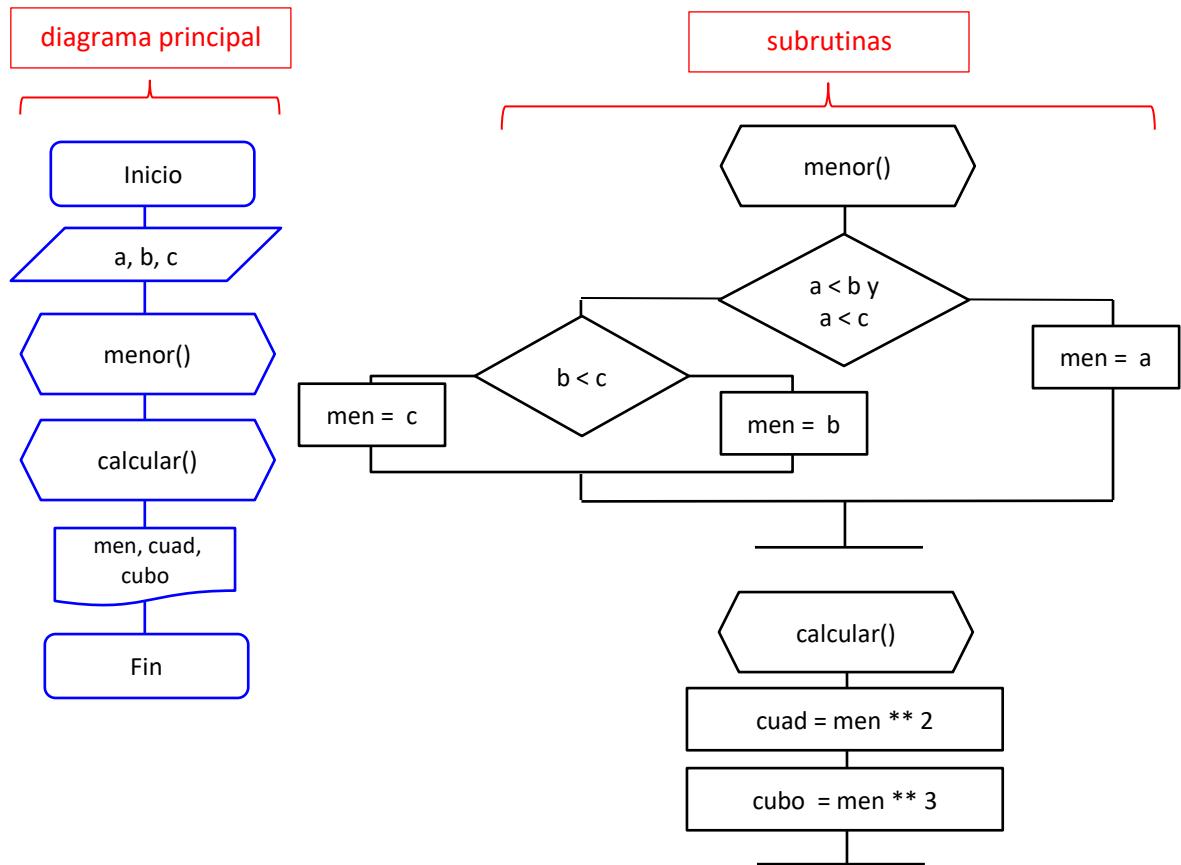
En un diagrama de flujo es especialmente fácil introducir nuevas reglas para denotar la presencia de una subrutina: simplemente se diagrama la misma por separado. El diagrama que se ve en la *Figura 7* (página 188) es el mismo que el que anteriormente mostramos en la *Figura 6* para el problema del cuadrado y el cubo, pero replanteado para incluir en forma explícita las subrutinas previstas por el programador [1].

Como puede verse, ahora hay un *diagrama principal* (que comienza y termina con los clásicos símbolos ovales de *Inicio* y *Fin*) y varios diagramas separados: uno por cada *subrutina* (es decir, uno por cada subproblema), pero de forma tal que el programador coloca un *nombre* o *identificador* a cada subrutina que grafica por separado. Por convención, y para anticiparnos a lo que luego se debe hacer en Python para declarar e invocar a una función, colocaremos al final de los nombres asociados a cada función un *par de paréntesis vacíos*.

En el *diagrama principal*, en lugar de graficar todos los procesos, sólo se dejan *indicados los nombres de las subrutinas* que los deben realizar, usando para ello el *símbolo de subrutina* (el hexágono de bases alargadas). Los nombres de cada subrutina son elegidos por el

programador, debiendo seguir para ello las mismas reglas y convenciones que valen para formar el nombre de una variable. Aquí, la primera subrutina se llama *menor()*, y la otra se llama *calcular()*:

**Figura 7: Diagrama del problema del cuadrado y el cubo, replanteado para incluir subrutinas.**



Observe que la lógica del diagrama no ha cambiado en absoluto, sino que sencillamente se ha estructurado el gráfico de manera diferente para que cada subproblema pueda ser entendido y estudiado por separado. De esta forma, si un problema es muy complicado y está dividido en muchos subproblemas, no será necesario plantear un "diagrama gigante" que abarque toda la lógica del mismo, sino que cada parte puede ser planteada y analizada por separado, incluso en momentos diferentes y por distintas personas.

### 3.] Funciones en Python - Parámetros y retorno de valores.

La división en subproblemas puede hacerse en un diagrama usando la técnica vista en la sección anterior. Si ahora se desea escribir el programa o script en Python, pero también dividiendo al mismo en subrutinas, puede hacerse (según ya dijimos) recurriendo a *funciones*. Como vimos, la idea es simple y directa: En Python, una *función* es una subrutina:

un segmento de programa que se codifica en forma separada, con un nombre asociado para poder activarla. En Python, una función tiene dos partes [3] [2]:

- La **cabecera**: también llamada *encabezado* de la función. Es la primera línea de la función, en ella se indica el nombre de la misma, entre otros elementos que oportunamente veremos (y que se designan como *parámetros*).
- El **bloque de acciones**: es la sección donde se indican los procesos que lleva a cabo la función. Este bloque debe comenzar a renglón seguido de la cabecera, y debe ir indentado hacia la derecha del comienzo de la cabecera. Es típico (como también veremos) que este bloque finalice con una instrucción de retorno de valor, llamada *return*.

A modo de ejemplo, veamos la forma que podría tener en Python la función que corresponde a la subrutina *menor()* de nuestro programa anterior:

```
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn
```

Aquí, la línea `def menor(n1, n2, n3)`: es la **cabecera** de la función. Las variables *n1*, *n2*, *n3* encerradas entre los paréntesis se designan como *parámetros*, y esencialmente contienen *los datos* que la función debe procesar. Como se ve, es también en la cabecera donde el programador indica el nombre que llevará la función: en este caso, el nombre elegido fue *menor*.

El resto de la función es el **bloque de acciones** de la misma. Como ya se dijo, el bloque se escribe a renglón seguido de la cabecera, e indentado hacia la derecha del comienzo de la misma. En ese bloque se escriben las instrucciones que la función debe ejecutar cuando sea invocada o activada desde otro lugar del programa. En este caso el bloque de la función contiene una *instrucción condicional anidada* para determinar cuál de los valores *n1*, *n2*, o *n3* es el menor, asignándolo en la variable *mn* de acuerdo al resultado de la verificación. En el ejemplo, la última instrucción del bloque de acciones es la instrucción *return mn*. Esta instrucción hace que la función, antes de finalizar, devuelva el valor contenido en la variable *mn* al punto desde el cual la función haya sido invocada. La función se dará por terminada cuando el intérprete Python encuentre la primera línea cuya columna de indentación vuelva a ser la que corresponde al inicio de la cabecera.

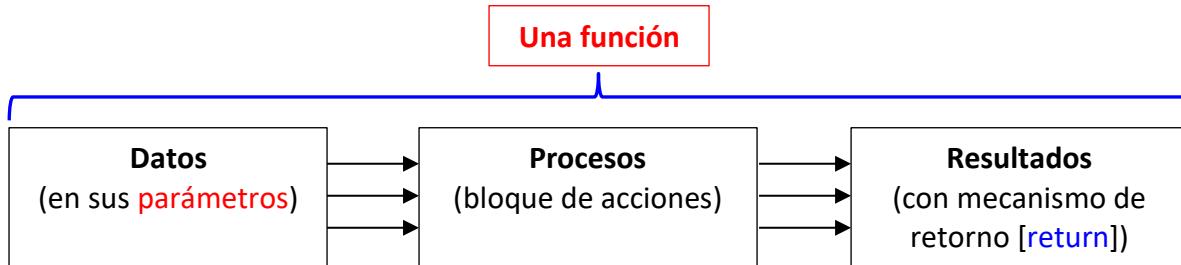
Intuitivamente, una función puede entenderse como un proceso separado (también designado como *caja negra* debido a que esos procesos permanecen ocultos para quien usa la función desde el exterior)<sup>2</sup> que acepta *entradas o datos* (los *parámetros*), procesa esos

---

<sup>2</sup> La idea de *caja negra* referida a un proceso oculto nos remite a la inquietante película canadiense *Cube* (conocida como *Cubo* en nuestro país) de 1997, dirigida por *Vincenzo Natali* y protagonizada (entre otros) por *Nicole de Boer*. Varias personas que no se conocen entre sí aparecen (sin explicación alguna) encerradas en una habitación con puertas en cada pared, en el piso y en el techo. Cada una de esas puertas lleva a su vez a otra habitación igual a la primera, y los enloquecidos prisioneros deben buscar la forma de salir de ese "cubo mágico" sin morir en el intento... ya que muchas de esas habitaciones tienen trampas mortales. Se convirtió en una película de culto del cine de ciencia ficción – terror, y derivó en una secuela: *Cube 2: Hypercube* [año 2002] y una precuela: *Cube Zero* [año 2004].

**datos** para obtener uno o más **resultados o salidas**, y devuelve esos **resultados** mediante la instrucción **return**. La figura que sigue es un esquema de esta idea:

Figura 8: Esquema general y conceptual de una función.



El programa completo que resuelve el problema pero ahora usando funciones en Python, podría tener el siguiente aspecto (analizaremos con detalle en las secciones que siguen de esta ficha la forma en que trabajan los **parámetros** y la instrucción **return**):

```

__author__ = 'Cátedra de AED'

# subproblema 1: determinar el menor...
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn

# subproblema 2: calcular cuadrado y cubo...
def calcular(mn):
    c2 = mn ** 2
    c3 = mn ** 3
    return c2, c3

# script principal...
# títulos y carga de datos...
print('Cálculo del cuadrado y el cubo del menor...')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))
c = int(input('Tercer número: '))

# procesos...
# invocar las funciones en el orden correcto de aplicación...
men = menor(a, b, c)
cuad, cubo = calcular(men)

# visualización de resultados...
print('Menor:', men)
print('Cuadrado:', cuad)
print('Cubo:', cubo)
  
```

La idea general es que por cada subrutina planteada en el diagrama de flujo o ideada por el programador se plantea una función, y la definición (el desarrollo de la cabecera y el bloque de acciones) de esas funciones debe hacerse en cualquier lugar que esté antes que el script desde donde se hacen las invocaciones a ellas: el intérprete lanzará un error si se intenta llamar a una función cuya definición aparezca más adelante (más abajo) de ese script en el código fuente. Por este motivo, el *script principal* (el bloque de código encargado de leer los valores de *a*, *b* y *c*, invocar a las funciones y luego mostrar los resultados), está *debajo* de las definiciones de ambas funciones.

En el *script principal* la línea `men = menor(a, b, c)` procede a *invocar* (o *llamar*) a la función *menor()*. Cuando decimos que una función es invocada o llamada, queremos decir que esa función es activada para que en ese momento se ejecuten las instrucciones que contiene en su bloque de acciones. Muy en general, cuando se invoca una función se escribe su nombre (*menor* en este caso) y luego entre paréntesis se escriben las variables o valores que se quieren enviar a la función a modo de datos. Esas variables se designan en general como **parámetros de la función**. En el programa anterior, al invocar a la función *menor()* se le están enviando tres parámetros: las variables *a*, *b* y *c* que se acaban de cargar por teclado en el script principal. La función toma los valores de esas variables, y automáticamente los asigna en las variables que ella misma tiene declaradas en su propia cabecera (en este caso, las variables *n1*, *n2* y *n3*). Como estas variables contienen ahora una copia de los valores que fueron enviados desde el script principal, la función puede operar con ellas y obtener el resultado esperado.

Cuando una función termina de ejecutarse, es común que disponga de uno o más valores obtenidos como resultado del proceso llevado a cabo por ella (en el caso del ejemplo, la variable *mn* con el valor del menor). Para que esos resultados sean *entregados* (o *retornados*) al punto desde el cual se llamó a la función, se usa la instrucción *return*. En la última línea de la función *menor()* puede verse la instrucción `return mn`, la cual esencialmente toma el valor de la variable *mn* y lo deja disponible para la instrucción desde la cual se haya invocado a la función. En este caso, recordemos que la función *menor()* fue invocada desde el script principal haciendo `men = menor(a, b, c)`. Esto quiere decir que la función tomará una copia de las variables *a*, *b* y *c*, buscará el menor entre esos valores, y retornará ese menor (almacenado internamente en la variable *mn*) para que sea asignado a su vez en la variable *men* del script principal. Así, si se hace una invocación de la forma:

```
a, b, c, = 3, 6, 2
men = menor(a, b, c)
```

entonces la función *automáticamente* asignará los valores *3*, *6* y *2* en las variables *n1*, *n2* y *n3* declaradas en su cabecera. Luego comprobará cuál de esas tres variables contiene el valor menor, el cual será asignado en la variable interna *mn*. Y finalmente, la instrucción *return mn* retornará el valor de *mn* que será asignado a su vez en la variable *men*. Por lo tanto, *men* quedará valiendo *2*. La Figura 9 (página 192) muestra en forma esquemática lo que ocurre al invocar a la función.

Es interesante notar que este proceso de toma de parámetros y retorno de valor ocurre en forma automática: es el propio Python el que copia los valores de las variables enviadas (*a*, *b*, y *c* en este caso) en las variables declaradas en la cabecera de la función (*n1*, *n2* y *n3*) y luego es también Python el que controla el retorno del resultado con la instrucción *return*. Y por cierto, esos mecanismos se activarán de la misma forma si la función es invocada para

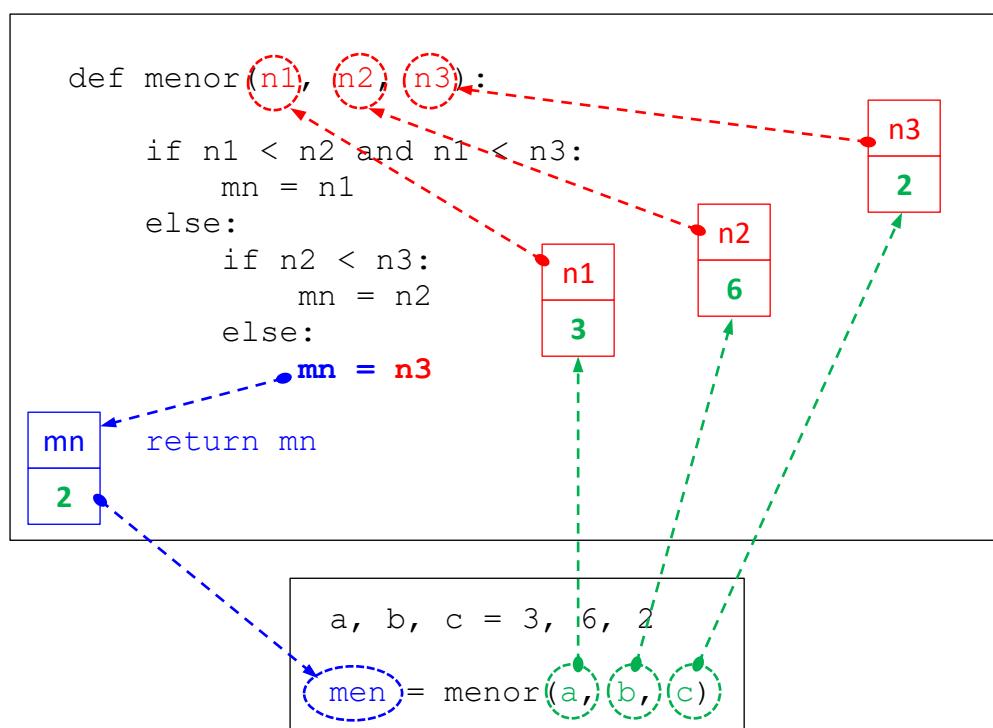
buscar el menor entre otras variables o valores constantes, como se ve en las invocaciones que siguen:

```
x, y, z = 30, 21, 75
m = menor(x, y, z)
# valor final de m: 21

p = menor(12, 100, 52)
# valor final de p: 12

t = 4
v = menor(5, t, 3)
# valor final de v: 4
```

Figura 9: Paso de parámetros a una función - Mecanismo de retorno en una función.



#### 4.] Paso de parámetros a una función (análisis detallado).

Las variables que se escriben entre los paréntesis de una función al invocarla, se designan con el nombre genérico de *parámetros*, y como pudo verse, un parámetro es una variable cuyo valor se *envía* a una función para que esta lo procese. En los ejemplos del final de la sección anterior, la función *menor()* fue invocada tres veces: en la primera invocación, se enviaron como parámetros los valores de las variables *x*, *y*, *z* para determinar el menor entre ellas. En la segunda invocación, los parámetros fueron los valores 12, 100 y 52. Y en la tercera invocación, se enviaron los valores 5, *t* y 3 (siendo *t* una variable previamente asignada).

La función *menor()* de la sección anterior fue desarrollada como parte de un programa propuesto en esta Ficha y es un ejemplo de cómo un programador puede plantear sus propias funciones. Pero note que el lenguaje Python provee una innumerable cantidad de

funciones ya definidas y listas para usar, todas basadas en los mismos principios de paso de parámetros y retorno de valores que hemos explicado. Algunos ejemplos conocidos de funciones provistas por Python (o *funciones nativas*) son *pow()*, *len()*, *int()*, *float()*, *print()*, *input()*, *randint()*, y muchas otras.

Como se dijo, una función representa un *proceso único*, pero este proceso puede ser aplicado sobre *variables diferentes*. Lo único que debe hacer un programador para que el proceso se aplique sobre unas u otras variables, es llamar a la función tantas veces como se requiera, y *enviar cada variable o valor como parámetro en cada llamada*. Como vemos, el *uso de parámetros* permite, finalmente, *que una misma función pueda ser usada en forma generalizada sobre variables diferentes que entran como datos*.

Para comprender la forma en que debe declararse la cabecera de una función, recordemos que cuando una función toma un parámetro, dicha función *hace caso omiso* del nombre del parámetro que recibe y se queda con una copia del *valor* del mismo para procesarlo. Pero para que la función pueda tomar esa copia, debe *asignar* dicho valor en *alguna otra variable que esté dentro de la función*, pues de otro modo lo perdería y no podría procesarlo. En otras palabras: la función debe declarar en alguna parte, una variable por cada parámetro que recibe, y en cada una de esas variables se guardará el valor de cada parámetro. Estas variables de uso interno se definen en la cabecera de la función, cuando se declara la misma. Volvamos a ver la función *menor()* de la sección anterior y una invocación a la misma para calcular el menor entre las variables *a*, *b* y *c*:

```
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn

# invocación a la función...
a, b, c, = 3, 6, 2
men = menor(a, b, c)
```

En la cabecera de la función se declaran las variables *n1*, *n2* y *n3*. Observemos que el objetivo de la función es determinar el menor entre *n1*, *n2* y *n3* que se consideran como los *datos* que la función recibe para el proceso que lleva a cabo.

Las variables que se declaran en la cabecera de una función también se llaman *parámetros*. Para evitar confusiones en las designaciones, a los parámetros definidos en la cabecera (como *n1*, *n2* y *n3* en el ejemplo) se los llama *parámetros formales*; y a los parámetros que se escriben entre paréntesis cuando se *invoca* a la función (como *a*, *b* y *c* en el ejemplo) se los llama *parámetros actuales*.

Cuando se hace la invocación *menor(a, b, c)* se están enviando como *parámetros actuales* las variables *a*, *b* y *c*. La función toma el valor del *primer parámetro actual* (la variable *a* en este caso) y lo *asigna automáticamente* en el *primer parámetro formal* de la función (o sea, la variable *n1* en este caso). El valor del *segundo parámetro actual* se asigna en forma también automática en el *segundo parámetro formal*, con lo cual el valor de *b* se asigna en la

variable *n2*. Y lo mismo ocurre con las variables *c* y *n3*. Si hubiera más parámetros, el programa seguiría el mismo mecanismo automático con los que queden. Cuando decimos que cada parámetro actual *se asigna en forma automática* en su correspondiente parámetro formal, queremos decir que el programador *no debe* escribir ninguna instrucción de asignación para efectuar esas acciones: *el programa las hace en forma implícita* [1].

Una vez que cada *parámetro actual* quedó asignado en su *parámetro formal* correspondiente, la función comienza a ejecutar su bloque de acciones. Pero como los *parámetros formales* tienen los mismos valores que los *parámetros actuales*, el programador de la función puede ahora *simplemente ignorar* los identificadores de los parámetros actuales, y limitarse a usar los identificadores de los formales.

De esta manera, *no importa como se llamen las variables en el momento de llamar a la función*. La función reemplaza esos nombres por los nombres de sus propios parámetros, y *se evita con esto tener que escribir el mismo código varias veces, cambiando el nombre de las variables*. Tanto da que la invocación haya sido del tipo *menor(a, b, c)* o *menor(c, d, e)*: no habrá problema alguno. Cuando en la función se use la variable *n1*, su valor será el correspondiente a la variable *a* o a la variable *c* (o a la que sea que se haya enviado en primer lugar). En forma similar, cuando la función use el valor *n2*, estará usando en realidad el valor de *b* o el valor de *d*, o cualquiera que sea la que se haya enviado como segundo parámetro actual.

Para finalizar esta breve introducción al concepto de parámetro, digamos que el uso de parámetros en una función se basa en las siguientes reglas generales:

1. Si en la cabecera de una función están declarados *n parámetros formales*, entonces al invocar a esa función deben enviarse *n parámetros actuales*.
2. El tipo de valor contenido en los *parámetros actuales* debería coincidir con el tipo de valor que el programador de la función esperaba para cada uno de los *parámetros formales*, tomados en orden de aparición de izquierda a derecha, para evitar problemas en tiempo de ejecución al intentar procesar valores de tipos incorrectos. Por ejemplo, supongamos la misma función *menor()* que ya hemos analizado:

```
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn
```

Está claro que en esta función se espera que los valores de los tres *parámetros formales* sean del mismo tipo (no necesariamente numérico). Tal como está planteada, esta función podría servir tanto para retornar el menor entre tres números, como para retornar el menor entre tres cadenas de caracteres:

```
m1 = menor(3, 5, 2)
# valor final de m1: 2

m2 = menor('mesa', 'silla', 'casa')
# valor final de m2: 'casa'
```

Pero la función fallaría si al invocarla ocurriese que uno de los parámetros es un número y los otros son cadenas (o cualquier combinación similar con parámetros de tipos diferentes), ya que en general no se pueden comparar valores de tipos diferentes. Si se la invoca en la forma siguiente:

```
m1 = menor('sol', 3, 'casa')
```

la función (y el programa) se interrumpiría en forma abrupta, lanzando un error como el que se muestra:

```
Traceback (most recent call last):
  File "C:/Ejemplo/prueba3.py", line 37, in <module>
    test()
  File "C:/Ejemplo/prueba3.py", line 26, in test
    m1 = menor('sol', 3, 'casa')
  File "C:/Ejemplo/prueba3.py", line 5, in mayor
    if n1 > n2:
TypeError: unorderable types: str() > int()
```

3. Al declarar la función debe indicarse la lista de *parámetros formales* de la misma, simplemente escribiendo sus nombres separados por comas. Los nombres que se designen para estos *parámetros formales* pueden ser cualesquiera (no importa si coinciden eventualmente o no con los nombres o identificadores de los *parámetros actuales*).
4. En el bloque de acciones de la función, deben usarse los *parámetros formales* y no los *parámetros actuales*. El nombre o identificador de los *actuales* no tiene importancia alguna dentro de la función, ya que sus valores *fueron copiados* en los *formales*. De hecho, como se dijo, los *parámetros actuales* podrían tener el mismo nombre que los *parámetros formales*, sin que esto afecte la forma de trabajo del programa ni de la función. No obstante, debe entenderse que incluso en este caso las variables actuales son diferentes de las formales (aunque se llamen igual) porque están ubicadas en *lugares distintos de la memoria*. Por ejemplo, en el siguiente esquema se define una función llamada *mayor()* que toma dos parámetros *n1* y *n2* y retorna el valor mayor, que al ser calculado se almacena temporalmente en la variable *m*:

```
def mayor(n1, n2):
    if n1 > n2:
        m = n1
    else:
        m = n2
    return m

def test():
    n1 = int(input('Primer valor: '))
    n2 = int(input('Segundo valor: '))

    m1 = mayor(n1, n2)

    print('Mayor:', m1)
```

Como se ve, en este caso los parámetros actuales tienen los mismos identificadores que los formales. Pero esto no es obligatorio, ni afecta en nada al funcionamiento del modelo: podrían haberse designado con nombres diferentes y el efecto sería el mismo: las variables *actuales* (se llamen como se llamen) están ubicadas en un lugar de la memoria (en este caso, el espacio de nombres de la función *test()*) y las formales están ubicadas en otro (en este caso, el espacio de nombres de la función *mayor()*). Los identificadores pueden coincidir (o no) pero las variables son variables diferentes e independientes las unas de las otras.

5. Si dentro del bloque de acciones de la función se altera el valor de un *parámetro formal*, este cambio **no afectará** al valor del *parámetro actual* asociado con él. Esto es simple de ver: al invocar a la función, se crean los *parámetros formales* como variables separadas, distintas e independientes, y automáticamente se copian en ellas los valores de los *parámetros actuales* respectivos (este esquema se conoce como *parametrización por valor* o *parametrización por copia*). Pero luego la función trabaja exclusivamente con los *parámetros formales*, que son variables diferentes que contienen *copias* de los *actuales*. Está claro entonces: si el programador cambia el valor de un *parámetro formal* (por asignación o carga por teclado, por ejemplo) *estará modificando la copia (y no la original)* por lo que las variables *actuales* permanecerán con sus valores. Suponga el siguiente caso, sólo a modo de ejemplo técnico:

```
def mayor(n1, n2):
    if n1 > n2:
        m = n1
    else:
        m = n2

n1 = n2 = 0
return m

def test():
    a = int(input('a: '))
    b = int(input('b: '))

    m1 = mayor(a, b)

    print('a:', a, 'b:', b, 'Mayor:', m1)

# script principal...
test()
```

Aquí la función *mayor()* asigna el mayor entre *n1* y *n2* en la variable interna *m*, y luego (antes de terminar con el *return* final) asigna un 0 tanto en *n1* como en *n2*. La función *test()* invoca a la función *mayor()*, le pasa las variables *a* y *b* como *parámetros actuales* y finalmente muestra los valores de *a*, *b* y el mayor obtenido. Sin importar si la función modificó los valores de *n1* y *n2*, lo que se mostrará en pantalla serán los *valores originales de las variables a y b*, que no sufrirán cambio alguno: *son variables diferentes, cuyos valores fueron copiados en n1 y n2*.

El uso de parámetros en una función es una práctica recomendable en toda situación, y no sólo en funciones que se invocan varias veces sobre diferentes variables en un mismo programa. Parametrizando una función el programador se prepara para la eventualidad de tener que modificar su programa, y en esa modificación tener que invocar varias veces a la misma función para operar sobre variables distintas. Por otra parte, una función parametrizada es más fácil de controlar en cuanto a posibles errores lógicos, ya que las variables que usa están definidas dentro de ella y sólo existen y pueden usarse dentro de ella. Y finalmente, una función parametrizada puede volver a utilizarse en forma natural en el mismo programa o en otros, sin tener que hacer modificaciones especiales, haciendo así más cómodo, rápido y eficiente el trabajo del programador.

## 5.] Mecanismo de retorno de valores en una función en Python (análisis detallado).

Hemos visto que normalmente una función suele definirse de forma que calcule uno o más resultados, y que en Python la declaración de una función puede hacerse de forma que la misma *retorne uno o más valores como resultado de su invocación*, mediante la instrucción *return*.

Pero una función también se puede declarar de forma que *no retorne valor alguno* sino que simplemente desarrolle una acción y termine. Por ejemplo, consideremos las siguientes funciones ya conocidas por nosotros, de la librería estándar de Python:

```
y = pow (x, n)           print(cadena)
```

La primera (*pow(x, n)*) permite elevar el valor *x* a la potencia de *n*, siendo *x* y *n* dos números que se toman como parámetro, y la segunda (*print(cadena)*) permite mostrar en la consola estándar lo que sea que contenga la variable o expresión *cadena*. Estas dos funciones que el lenguaje Python provee en librerías estándar, ilustran las dos clases de funciones que en general existen: las *funciones con retorno de valor* y las *funciones sin retorno de valor*:

- *Funciones con retorno de valor*: son aquellas que *devuelven un valor* como resultado de la acción que realizan, de forma tal que ese valor puede volver a usarse en alguna otra operación. La función *pow()* es de esta clase y puede usarse, por ejemplo, en cualquiera de las formas siguientes:

Ejemplo de uso	Explicación
<i>y = pow(x, n)</i>	El valor devuelto se asigna en una variable <i>y</i> .
<i>print(pow(x, n))</i>	El valor devuelto se muestra directamente en consola.
<i>y = 2 * pow(x, n)</i>	El valor devuelto se multiplica por dos y el resultado se asigna en <i>y</i> .
<i>y = pow(pow(x, n), 3)</i>	Calcula $x^n$ , eleva el resultado al cubo, y asigna el resultado final en <i>y</i> .

- *Funciones sin retorno de valor*: son aquellas que realizan alguna acción *pero no se espera que retornen valor alguno* como resultado de la misma. Un ejemplo es la función *print()* que permite visualizaciones en la consola estándar. Notar que una función sin retorno de valor típicamente no es utilizada en las formas que se indicaron para las funciones con retorno de valor, justamente debido a la ausencia de ese valor. Para invocarlas, simplemente se escribe su nombre y se pasan los parámetros que sean necesarios, si es que la función los pide:

```
print('Hola mundo...')
```

Las funciones *pow()* y *print()* como muchas otras, son funciones que ya vienen provistas por el lenguaje Python para usar en un programa. Pero además, como vimos, en Python y en cualquier lenguaje un programador puede desarrollar sus propias funciones de forma que se comporten en forma análoga a las funciones provistas por el lenguaje. Nos proponemos ahora mostrar la forma en que pueden definirse en Python funciones de los dos tipos que hemos expuesto.

### ✓ Implementación y uso de funciones que retornan valores en Python.

Las funciones *menor()* y *mayor()* que hemos analizado en la sección anterior son ejemplos claro de funciones que retornan algún valor. La idea general es que en el bloque de acciones de la función se desarrolle el proceso que debe llevar a cabo, y en algún momento se almacene en una o más variables el resultado o resultados de ese proceso. Al finalizar el

cálculo se deberá incluir una instrucción `return`, que es la que fuerza a la función a retornar el valor que se le indique.

Muchos problemas matemáticos pueden resolverse mediante algoritmos programados como funciones que retornan valores. A modo de ejemplo, tomemos el ya conocido caso del cálculo del *factorial* de un número (que se presentó en la *Ficha 7, problema 18*). En el programa que sigue se incluye una función `factorial()` que justamente hace ese trabajo:

```
__author__ = 'Catedra de AED'

def factorial(n):
    f = 1
    for i in range(2, n + 1):
        f *= i

    return f

# script principal...
num = int(input('Ingrese un numero (>=0 por favor): '))
fn = factorial(num)
print('Factorial de', num, '=', fn)
```

En este ejemplo, dentro de la función `factorial()` el parámetro *n* contiene el número al cual le será calculado el factorial, y ese factorial se guarda *transitoriamente* en la variable *f*. Como se dijo, las *funciones con retorno de valor* deben incluir en alguna parte de su bloque de acciones la instrucción `return`, indicando en la misma el valor que la función debe retornar. Ese valor volverá al punto donde fue llamada la función, que en nuestro caso es la línea dentro del script principal que indica `fn = factorial(num)`. El valor retornado se guarda en la variable *fn*.

Observar que la instrucción `return` es *cancelativa*: al ejecutar esta instrucción, la ejecución de la función que la contiene se *da por finalizada*, aun cuando no se haya llegado al final de la misma. Esto significa que debe tenerse cuidado en el uso de la instrucción `return` porque cualquier instrucción que sea colocada debajo de ella en la misma rama lógica en una función *no será ejecutada* (pero Python no informará error alguno). Por ejemplo, la siguiente función está mal planteada porque el `print()` del final nunca será ejecutado al estar ubicado debajo de `return` en el mismo bloque:

```
def f1():
    x = int(input('x: '))
    f = 4 * x
    return f
    print(f)      # incorrecto: este print nunca será ejecutado...
```

Por otra parte, notemos que en Python una función puede retornar dos o más valores si fuese necesario, en forma de tupla [3] [2]. Esto constituye una característica muy propia y especial de Python, ya que en la mayoría de los lenguajes la instrucción `return` no puede usarse para devolver más de un elemento a la vez. El siguiente ejemplo muestra un programa completo que contiene una función `min_max()` que toma tres parámetros *a*, *b* y *c* y *calcula y retorna tanto el menor como el mayor* de esos tres números:

```
__author__ = 'Catedra de AED'
```

```

def min_max(a, b, c):
    # determinar el menor mn
    if a < b and a < c:
        mn = a
    elif b < c:
        mn = b
    else:
        mn = c

    # determinar el mayor my
    if a > b and a > c:
        my = a
    elif b > c:
        my = b
    else:
        my = c

    # retornar los resultados...
    return mn, my

def test():
    print('Funciones que retornan multiples valores...')
    a = int(input('a: '))
    b = int(input('b: '))
    c = int(input('c: '))

    # procesos: invocar a la función...
    men, may = min_max(a, b, c)

    # visualización de resultados...
    print('El menor es:', men)
    print('El mayor es:', may)

    # script principal...
    # ... solo invoca a la función test()
    test()

```

Además de la función *min\_max()*, el programa incluye una segunda función *test()* que hace la carga por teclado de los datos, invoca a la función *min\_max()* y muestra los resultados al final. Esta es una forma típica de plantear un programa completo: en lugar de un script principal largo y complejo, se usa una función específicamente dedicada a "arrancar el programa". Sigue siendo necesario que el programa incluya un script principal, pero si existe esta función de arranque el script principal se limita sólo a invocar a esa función (veremos con más detalle esta y otras estrategias de planteo de programas en fichas posteriores.)

Como la función *min\_max()* está retornando una tupla, para invocarla y asignar los dos valores que devuelve se pueden emplear dos variables, como se ve en este segmento de la función *test()* que hace la llamada a *min\_max()* en el programa:

```

men, may = min_max(a, b, c)

print('El menor es:', men)
print('El mayor es:', may)

```

En este caso, el primer valor de la tupla retornada por *min\_max()* (o sea, el número menor) será asignado en la variable *men*, y el segundo (el mayor) en la variable *may*. Obviamente,

también se puede invocar a la función asignando el resultado en una única variable, que en este caso quedará definida ella misma como una tupla, con el valor menor en la posición 0 de la tupla, y el mayor en la posición 1 :

```
res = min_max(a, b, c)

print('El menor es:', res[0])
print('El mayor es:', res[1])
```

La función obviamente también puede asignar la tupla en una variable y retornarla (la función sigue retornando una tupla, y puede ser invocada exactamente en las mismas formas que vimos en los dos ejemplos anteriores) [3] [2]:

```
def min_max(a, b, c):

    # determinar el menor mn
    if a < b and a < c:
        mn = a
    elif b < c:
        mn = b
    else:
        mn = c

    # determinar el mayor my
    if a > b and a > c:
        my = a
    elif b > c:
        my = b
    else:
        my = c

    # retornar ambos...
    r = mn, my
    return r
```

Un detalle interesante al respecto de la división en subproblemas y la programación estructurada: en Python, una función puede contener la definición de una o más funciones *dentro de ella* (y estas "subfunciones" se conocen como *funciones locales*). Si el proceso que la función principal lleva a cabo puede descomponerse en subprocessos, no hay inconveniente (aunque tampoco obligación) en plantear cada subprocesso como una función local. Por ejemplo, la función *min\_max()* claramente está llevando a cabo dos procesos diferentes: el cálculo del menor y el cálculo del mayor. Podemos dejarla tal como está, pero también sería válido hacer un planteo como el siguiente:

```
def min_max(a, b, c):

    def menor():
        # determinar el menor mn
        if a < b and a < c:
            mn = a
        elif b < c:
            mn = b
        else:
            mn = c
        return mn

    def mayor():
        # determinar el mayor my
```

```

        if a > b and a > c:
            my = a
        elif b > c:
            my = b
        else:
            my = c
        return my

# invocar a las funciones locales...
r1 = menor()
r2 = mayor()

# retornar los resultados...
return r1, r2

```

Observe que esta versión de la función `min_max()` es básicamente la misma que oportunamente se desarrolló, y lo único que cambió es la forma en que están implementados internamente los procesos de cálculo del menor y del mayor: ahora tenemos dos funciones locales a la función `min_max()`: la primera se llama `menor()` y su trabajo es calcular y retornar el menor, y la segunda se llama `mayor()`, y está encargada de calcular y retornar el mayor.

Lo que debemos saber respecto de una función `f()` definida como local a otra `g()`, es que `f()` sólo existe y puede usarse dentro del ámbito del bloque de la función contenedora `g()`. Cualquier intento de usar `f()` fuera de `g()` provocará un error de intérprete: las funciones `menor()` y `mayor()` de nuestro ejemplo sólo pueden usarse dentro del bloque de acciones de la función `min_max()`.

#### ✓ Implementación y uso de funciones sin retorno de valor en Python.

Como ya se indicó, se trata de funciones que realizan una acción al ser invocadas, pero no se espera que retorne ningún valor al terminar. La función `print()` que ya conocemos, es de este tipo: muestra una salida por pantalla pero no se espera que retorne valor alguno al terminar.

Lo que esencialmente caracteriza a una función de esta clase es *la posible ausencia de la instrucción return en su bloque de acciones*. Simplemente, la función hace su trabajo y finaliza, pero no necesita entregar hacia el exterior un valor en forma explícita con la instrucción `return`.

Un ejemplo concreto es la misma función `test()` que sirve como función de entrada del programa para el cálculo del menor y el mayor que mostramos en la sección anterior:

```

def test():
    # título general y carga de datos...
    print('Funciones que retornan multiples valores...')
    a = int(input('a: '))
    b = int(input('b: '))
    c = int(input('c: '))

    # procesos: invocar a la función...
    men, may = min_max(a, b, c)

    # visualización de resultados...
    print('El menor es:', men)
    print('El mayor es:', may)

```

La función *test()* lleva a cabo el trabajo de cargar los datos, invocar a la función que calcula los resultados, y finalmente mostrar en la consola de salida esos resultados. Pero al concluir con todo, simplemente termina y *no cuenta con una instrucción return al final*.

Típicamente, si una función está planteada como sin retorno de valor, entonces lo común es que para invocarla *simplemente se escriba su nombre y se le pasen los parámetros que pudiera pedir*, sin más, como se hizo con la función *test()* en el *script principal* del programa anterior:

```
# script principal...
test()
```

Recordando que la instrucción *return* tiene efecto cancelativo, se puede hacer que una función incluya un *return* sólo para forzar su terminación (por ejemplo, si se presenta algún error). En casos así, *no es necesario* indicar el valor devuelto al usar *return*, pero como *return sigue siendo cancelativa*, debe tenerse aquí también el cuidado de no escribir instrucciones que queden por debajo de ella en la misma línea de ejecución. Por ejemplo, supongamos que se quiere plantear una función que tome dos números en dos parámetros *n1* y *n2* y muestre el cociente entre ellos. Como el cociente sólo está definido si el denominador es diferente de cero, el programador podría querer que la función calcule y muestre el cociente *n1 / n2* sólo si *n2* es diferente de cero, y *terminando sin hacer nada en caso contrario*. Una forma de hacerlo sería la siguiente [3]:

```
def dividir(n1, n2):
    if n2 == 0:
        return
    else:
        c = n1 / n2
        print('Cociente:', c)
```

Como se puede ver, la función *dividir()* chequea si *n2* es cero, y en ese caso usa un *return* para provocar la terminación de la función, sin acompañar a ese *return* con un valor para que sea devuelto. En la práctica, se dice que esta función no retorna valor (aunque esté usando un *return*), y la forma normal de invocarla es la que ya se indicó: escribir simplemente su nombre y su lista de parámetros (si los tuviese) entre paréntesis:

```
# invocación de la función...
dividir(n1, n2)
```

Note que si en la función anterior la condición fuese verdadera, se ejecutaría el *return* y luego la función terminaría. Lo que sea que se hubiese escrito debajo de *return* en el mismo bloque, o fuera de la condición, no llegaría a ejecutarse nunca. Por ese motivo, es común que los programadores más experimentados simplifiquen el código fuente de la función en la forma siguiente:

```
def dividir(n1, n2):
    if n2 == 0:
        return

    c = n1 / n2
    print('Cociente:', c)
```

Si observa con atención, esta función es totalmente equivalente a la original (que incluía un *else*): si la condición es cierta, la función termina (igual que la original). Por lo tanto, si la

condición es falsa, y sólo si es falsa, la función seguirá adelante y calculará y mostrará la división (igual que la función original) que ahora está fuera de la condición al eliminar el *else* (que claramente, ya no es necesario).

No obstante todo lo que hemos analizado hasta aquí, tenga en cuenta un detalle importante: en general, *cualquier* función en Python *siempre puede invocarse como si retornase un valor*, aun cuando la misma no incluya un *return* explícito en su bloque de acciones o si alguna de sus ramas lógicas no lo incluyese, o si se activa un *return* sin asociarle un valor. **En casos así la función retorna implícitamente el valor *None*** y el programador puede invocarla asumiendo ese retorno [3]. A modo de ejemplo, considere el siguiente programa sencillo:

```
__author__ = 'Cátedra de AED'

def titulos():
    print('Prueba de funciones...')
    print('... retorno implícito de None')

def test():
    x = titulos()
    print('x:', x)

# script principal...
test()
```

En el programa del ejemplo, la función *titulos()* es una función *sin retorno de valor*: no incluye un *return* explícito en su bloque de acciones. Sin embargo, en la función *test()* se invoca a *titulos()* como si esta retornase un valor, **asignando su potencial valor devuelto en la variable x**. Como la función *titulos()* en realidad no incluye un *return*, el valor que se asignará en *x* será finalmente *None*... pero el programa funcionará y no lanzará error. La ejecución del programa provocará la siguiente salida en consola:

```
Prueba de funciones...
... retorno implícito de None
x: None
```

Como se ve, Python asume que si una función *f()* es invocada en un contexto en el cual se esperaría que retorne un valor, asumirá que el valor devuelto es *None* si *f()* en realidad no tuviese previsto un valor a devolver. En ese sentido, la función *titulos()* del programa anterior podría dejarse tal como está, pero también **podría incluir explícitamente el retorno de *None*, y sería en todo equivalente a la versión original**:

```
__author__ = 'Cátedra de AED'

def titulos():
    print('Prueba de funciones...')
    print('... retorno explícito de None')
    return None

def test():
    x = titulos()
    print('x:', x)
```

```
# script principal...
test()
```

Para terminar de reforzar lo anterior, considere el siguiente programa:

```
__author__ = 'Cátedra de AED'

def procesar():
    a = int(input('a: '))
    b = int(input('b: '))

    if b == 0:
        return

    c = a / b
    return c

def test():
    x = procesar()
    print('x:', x)

# script principal
test()
```

La función *procesar()* del ejemplo, retornará el cociente entre *a* y *b* si *b* es diferente de cero y por lo tanto, hasta aquí, es una *función con retorno de valor*. Pero si *b* es cero, la función termina su ejecución con un *return* que no devuelve valor alguno, y por lo tanto, si pasa por esta rama, la función se comporta como una *función sin retorno de valor*. ¿Cuál es entonces el valor que quedará alojado en la variable *x* al invocar a *procesar()* en el bloque de acciones de la función *test()*? Respuesta: si *b* es diferente de cero, *x* quedará valiendo el cociente entre *a* y *b*; pero si *b* es cero, *x* quedará valiendo *None*.

Las siguientes son formas de plantear la misma función *procesar()* del ejemplo anterior, que son *totalmente equivalentes a la anterior* (dejamos el análisis de cada una para el estudiante):

```
# versión 2...
def procesar():
    a = int(input('a: '))
    b = int(input('b: '))

    if b == 0:
        return None

    c = a / b
    return c

# versión 3...
def procesar():
    a = int(input('a: '))
    b = int(input('b: '))

    if b != 0:
        c = a / b
        return c
```

```

# versión 4...
def procesar():
    a = int(input('a: '))
    b = int(input('b: '))

    if b != 0:
        c = a / b
        return c
    else:
        return None

# versión 5...
def procesar():
    a = int(input('a: '))
    b = int(input('b: '))

    if b != 0:
        c = a / b
        return c

    return None

# versión 6...
def procesar():
    a = int(input('a: '))
    b = int(input('b: '))

    if b != 0:
        c = a / b
        return c

    return

```

Finalmente, digamos que la instrucción `pass` (que ya vimos al estudiar las instrucciones condicionales y los ciclos) puede usarse también para dejar vacío el bloque de acciones de una función (en casos en que el programador aún no tenga definido qué debe hacer esa función), como se ve en los siguientes ejemplos:

```

# funcion para calcular el factorial de un numero
def factorial(n):
    pass      # pendiente para desarrollar...

# funcion para determinar el mayor entre dos numeros...
def mayor(a, b):
    pass      # pendiente para desarrollar...

```

Para cerrar esta introducción general a los conceptos de parametrización y retorno de valores, proponemos un nuevo problema a modo de caso de análisis:

**Problema 27.)** *Se tienen los datos de tres postulantes a un empleo, a los que se les realizó un test para conocer el nivel de formación previa de cada uno. Por cada postulante, se tienen entonces los siguientes datos: nombre del postulante, cantidad total de preguntas que se le realizaron y cantidad de preguntas que contestó correctamente. Se pide confeccionar un programa que lea los datos de los tres postulantes, informe el nivel de formación previa de cada uno según los criterios de aprobación que se indican más abajo, e indique finalmente el nombre del postulante que ganó el puesto. Los*

*criterios de aprobación son los siguientes, en función del porcentaje de respuestas correctas sobre el total de preguntas realizadas a cada postulante:*

- *Nivel Superior:* Porcentaje  $\geq 90\%$
- *Nivel Medio:*  $75\% \leq \text{Porcentaje} < 90\%$
- *Nivel Regular:*  $50\% \leq \text{Porcentaje} < 75\%$
- *Fuera de Nivel:*  $\text{Porcentaje} < 50\%$

**Discusión y solución:** Se trata de un problema típico de gestión de datos del personal de una empresa. El programa completo podría ser el siguiente:

```

__author__ = 'Catedra de AED'

def porcentaje(tp, pbc):
    return pbc * 100 / tp

def nivel(p):
    if p >= 90:
        return 'Superior'

    if p >= 75:
        return 'Medio'

    if p >= 50:
        return 'Regular'

    return 'Fuera de Nivel'

def test():
    # título general y ciclo de carga de datos...
    print('Selección de nuevo personal para una empresa...')
    for i in range(1, 4):
        # carga de datos de UN postulante...
        nom = input('Nombre postulante ' + str(i) + ': ')
        tp = int(input('Total de preguntas: '))
        cpbr = int(input('Total de preguntas bien respondidas: '))

        # procesos...
        pr = porcentaje(tp, cpbr)

        # procesos y visualización de resultados...
        print('Nombre', i, ':', nom, '- Nivel:', nivel(pr), '- Porc.:', pr, '%')

        # determinación del aspirante con mayor porcentaje...
        if i == 1:
            pmay, nmay = pr, nom
        elif porc > pmay:
            pmay, nmay = pr, nom

    # visualización del ganador del puesto...
    if pmay > 50:
        print('Ganador:', nmay, '- con porcentaje de:', pmay, '%')
    else:
        print('No hay ganador: todos tienen porcentaje menor al 50%')

```

```
# script principal...
# ... sólo invocar a test()...
test()
```

En el programa mostrado, la función **test()** se usa como función de entrada o arranque del programa, y es la única que se invoca en el script principal. No tiene parámetros ni retorna valores, y su tarea es tomar desde el teclado todos los datos de todos los postulantes, para luego invocar a las funciones que realizan el resto del trabajo y mostrar los resultados. La carga de datos se hace mediante un ciclo *for*, ajustado para realizar tres iteraciones (una por cada aspirante a cargar):

```
def test():

    # título general y ciclo de carga de datos...
    print('Selección de nuevo personal para una empresa...')
    for i in range(1, 4):
        # carga de datos de UN postulante...
        nom = input('Nombre postulante ' + str(i) + ': ')
        tp = int(input('Total de preguntas: '))
        cpbr = int(input('Total de preguntas bien respondidas: '))

        # procesos...
        pr = porcentaje(tp, cpbr)

        # procesos y visualización de resultados...
        print('Nombre', i, ':', nom, '- Nivel:', nivel(pr), '- Porc.:', pr, '%')

        # determinación del aspirante con mayor porcentaje...
        if i == 1:
            pmay, nmay = pr, nom
        elif pr > pmay:
            pmay, nmay = pr, nom

    # visualización del ganador del puesto...
    if pmay > 50:
        print('Ganador:', nmay, '- con porcentaje de:', pmay, '%')
    else:
        print('No hay ganador: todos tienen porcentaje menor al 50%')

# script principal...
# ... sólo invocar a test()...
test()
```

En cada vuelta del ciclo, se cargan los datos de *un* aspirante en las variables *nom*, *tp* y *cpbr*. Está claro que lo primero que debe hacerse una vez cargados los datos de cada aspirante, es calcular su porcentaje de respuestas correctas. Esto se hace con la función **porcentaje()**, la cual toma como parámetro la cantidad total de preguntas que se le hicieron a un postulante (*tp*) y la cantidad total de preguntas que ese postulante respondió en forma correcta (*pbc*), y calcula y retorna el porcentaje que *pbc* representa en *tp*:

```
def porcentaje(tp, pbc):
    return pbc * 100 / tp
```

Como la función está parametrizada, puede ser usada enviándole distintos valores de distintos postulantes como se hace en la función *test()*:

```
# procesos...
pr = porcentaje(tp, cpbr)
```

La función `nivel()` toma un único parámetro  $p$ , conteniendo el porcentaje de preguntas bien contestadas por un postulante cualquiera, y retorna una cadena con el nivel de conocimientos previos que mostró ese postulante, de acuerdo a la tabla que se indicó en el enunciado:

```
def nivel(p):
    if p >= 90:
        return 'Superior'

    if p >= 75:
        return 'Medio'

    if p >= 50:
        return 'Regular'

    return 'Fuera de Nivel'
```

Otra vez: como la función está parametrizada, se usa enviándole distintos porcentajes de distintos postulantes como se hace en la función `test()` directamente al mostrar los resultados:

```
# procesos y visualización de resultados...
print('Nombre', i, ':', nom, '- Nivel:', nivel(pr), '- Porc.:', pr, '%')
```

Al final del bloque de acciones del ciclo, se incluye un proceso para detectar cuál es el mayor porcentaje que haya sido obtenido por los postulantes, y el nombre del postulante que obtuvo ese porcentaje mayor. El proceso que se aplica es que se estudió en la *Ficha 07*, concretamente la *variante 1* (consistente en preguntar en cada vuelta del ciclo si el dato que se procesa es el primero o no). Cada vez que se detecta un porcentaje mayor al que ya se tenía en la variable `pmay`, se cambia el valor de `pmay` por ese nuevo porcentaje, y al mismo tiempo se almacena en la variable `nmay` el nombre del aspirante con ese porcentaje:

```
# determinación del aspirante con mayor porcentaje...
if i == 1:
    pmay, nmay = pr, nom
elif porc > pmay:
    pmay, nmay = pr, nom
```

Inmediatamente luego de finalizar el ciclo, se usa un `if` para chequear si el porcentaje mayor que quedó guardado en `pmay` es superior a 50. Si ese fue el caso, se muestra junto con el nombre contenido en `nmay` como el ganador del puesto. Pero si `pmay` quedó valiendo menos que 50, se muestra un mensaje como "Ninguno... todos están fuera de nivel" ya que en ese caso **todos** los porcentajes eran menores que 50:

```
# visualización del ganador del puesto...
if pmay > 50:
    print('Ganador:', nmay, '- con porcentaje de:', pmay, '%')
else:
    print('No hay ganador: todos tienen porcentaje menor al 50%)
```

## 6.] Variables locales y variables globales.

En Python, cualquier variable que se inicializa dentro del bloque de una función es asumida como una *variable local* a ese bloque (y por lo tanto, *local a la función*). En general, una *variable local* a una función es aquella que sólo puede usarse dentro de esa función, y no existe (o no es reconocida) fuera de ella. Se dice por esto, que en Python toda función define un *espacio de nombres* (o *namespace*): un bloque de código en el cual las variables que se definen pertenecen a ese espacio y no son visibles ni utilizables desde fuera de él [3].

Consideremos el programa para el cálculo del factorial que se presentó anteriormente en esta misma Ficha:

```
__author__ = 'Catedra de AED'

def factorial(n):
    f = 1
    for i in range(2, n + 1):
        f *= i

    return f

# script principal...
num = int(input('Ingrese un numero (>=0 por favor): '))
fn = factorial(num)
print('Factorial de', num, '=', fn)
```

La función *factorial()* de este programa recibe un parámetro formal *n* con el número al cual debe calculársele el factorial, y dentro de su bloque de acciones usa dos variables adicionales *f* e *i*. Como ambas son inicializadas (o sea, definidas) dentro de la función, ambas son entonces *variables locales a la función*: ninguna de las dos puede usarse fuera del bloque de *factorial()*, y cualquier intento de hacerlo produce un error de intérprete por uso de variable no definida, como ocurriría si se plantea el programa de forma que se pida mostrar el valor de *f* en el script principal:

```
__author__ = 'Catedra de AED'

def factorial(n):
    f = 1
    for i in range(2, n + 1):
        f *= i

    return f

# script principal...
num = int(input('Ingrese un numero (>=0 por favor): '))
fn = factorial(num)
print('Factorial de', num, '=', fn)
print('Valor de f:', f)
```

Si se intenta ejecutar el programa anterior, se producirá el siguiente mensaje de error luego de interrumpirse la ejecución:

```
Ingrese un numero (>=0 por favor): 3
Traceback (most recent call last):
  File "C:/Ejemplo/factorial.py", line 16, in <module>
    print('Valor de f:', f)
NameError: name 'f' is not defined
```

Como la asignación de la variable *f* se está haciendo dentro de la función, Python asume que esa variable está siendo definida *en ese momento* y que le pertenece sólo a esa función, por lo que no permitirá que sea utilizable desde fuera de ella.

Por otra parte, observe que si una variable se define en el *script principal* (es decir, fuera de cualquier función y por lo tanto sin quedar encerrada en el bloque de ninguna) entonces *puede ser usada en cualquier lugar del programa, ya sea en el propio script principal y/o en toda función de ese programa*. Si el programa anterior se modifica para que la función *factorial()* muestre en pantalla el valor de la variable *num* (definida en el script principal), efectivamente funcionará:

```
__author__ = 'Catedra de AED'

def factorial(n):
    f = 1
    for i in range(2, n + 1):
        f *= i
    print('Número:', num)

    return f

# script principal...
num = int(input('Ingrese un numero (>=0 por favor): '))
fn = factorial(num)
print('Factorial de', num, '=', fn)
```

La salida de este programa será algo como:

```
Ingrese un numero (>=0 por favor): 6
Número: 6
Factorial de 6 = 720
```

En general, la región de un programa donde una variable es reconocida y utilizable, se llama el **ámbito** de esa variable. Las variables *num* y *fn* definidas en el script principal del programa anterior, no están encerradas en ningún bloque de función, y por lo tanto, pueden ser usadas en cualquier parte del programa, incluidas las funciones que ese programa tuviese. Se dice que esas variables son de **ámbito global** en el programa (o que son **variables globales**).

Como vimos, las variables que se asignan (o sea, se definen) dentro del bloque de una función (como *f* e *i* en la función *factorial()*) se consideran *variables nuevas*, propias de esa función, y se conocen como variables de **ámbito local** a esa función (o **variables locales** a la función). Si una variable es local a un ámbito, sólo puede usarse dentro de ese ámbito.

Los *parámetros formales* de una función, son también *variables locales* a esa función, aunque con una pequeña diferencia o ventaja a su favor: los *parámetros formales son variables locales que se inicializan en forma automática al ser invocada la función*, asignando en ellos los valores que se hayan enviado como parámetros actuales. En cambio, una variable local común debe ser inicializada en forma explícita dentro del bloque de la función con una asignación.

En general, si una variable se define como local a una función es porque el programador no tiene intención de hacer que esa variable sea visible desde fuera de la función. La variable se está usando como una variable auxiliar interna dentro del proceso que la función encierra, y no necesariamente como un dato o un resultado importante que deba ser tomado desde o enviado hacia el exterior.

Si el programador necesita compartir con otras funciones el valor de una variable local la forma esencial de hacerlo es ya conocida: si la variable debe tomar datos desde el exterior, definirla como *parámetro formal en la función*, y si debe devolverse al exterior el valor de la variable, usar el *mecanismo de retorno* (instrucción *return*) para lograrlo. Toda otra variable que se use en la función pero no sea ni dato ni resultado debería dejarse como local, inaccesible desde el exterior. En el programa del cálculo del factorial, la función *factorial()* toma como parámetro a *n* (dato) y retorna el valor final de *f* (resultado). La variable *i* que se usa para controlar el *for* no es dato ni resultado sino una simple variable auxiliar de control y por eso se mantiene estrictamente como local.

En Python existe una forma adicional de hacer que una variable definida dentro de una función sea visible desde fuera de la misma y consiste en usar la palabra reservada *global* para avisarle al intérprete que las variables enumeradas con ella deben ser tomadas como *globales* y *no como locales* [3]. Considere la siguiente variante del programa del factorial:

```
__author__ = 'Catedra de AED'

def factorial(n):
    global f, i
    f = 1
    for i in range(2, n + 1):
        f *= i

    return f

# script principal...
num = int(input('Ingrese un numero (>=0 por favor): '))
fn = factorial(num)
print('Factorial de', num, '=', fn)

print('Valor final de f:', f)
print('Valor final de i:', i)
```

En esta versión del programa, el bloque de acciones del función *factorial()* comienza con una declaración *global*, listando en la misma línea y separadas por comas a todas las variables que el programador desea informar como de *ámbito global*. La declaración *global f, i* hace que el intérprete no encierre a esas variables en el *ámbito local* de la función, sino que las hace *escapar hacia el ámbito global*. De esta forma, la primera asignación que se haga en estas dos variables dentro de esta función será tomada como una *definición global de variable*, y podrán ser compartidas y utilizadas en otras funciones o en el script principal: de hecho, las dos instrucciones *print()* que figuran al final de ese script principal ejecutan sin problemas y muestran los valores de las variables *f* e *i* tal como quedaron asignados dentro de la función.

Un hecho que debe notar y tratar en forma cuidadosa, es que distintas funciones podrían tener variables locales designadas con el mismo identificador, o con el mismo nombre que alguna variable en el script principal y todavía así serían consideradas como variables diferentes.

A modo de ejemplo, suponga el siguiente programa, en el cual se cargan por teclado dos variables *a* y *b*. La función *ordenar()* toma esas variables como parámetro y las ordena, asignando el menor en la variable *men* y el mayor en *may*. A su vez la función

*calcular\_areas()* toma como parámetros a *men* y *may*, y calcula el área de un círculo suponiendo que *men* es el radio, y el área de un cuadrado suponiendo que *may* es su lado.

En este ejemplo (y sólo por ser un ejemplo...) la función *ordenar()* no retorna los valores de *men* y *may*, sino que los deja asignados en esas variables:

```

def ordenar(a, b):
    if a > b:
        men = b
        may = a
    else:
        men = a
        may = b

def calcular_areas(men, may):
    global acuad, acirc
    acuad = may ** 2
    acirc = 3.1415 * men ** 2

# script principal...
# asignación de variables para supuesto uso global...
men, may = 0, 0

print('Cálculo de las áreas de un cuadrado y un círculo...')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))

ordenar(a, b)
calcular_areas(men, may)

print('Área del cuadrado:', acuad)
print('Área del círculo:', acirc)

```

Como el script principal definió las variables *men* y *may* y las asignó con el valor cero a cada una, entonces ambas son de *ámbito global* y por lo tanto se podría esperar que fuesen utilizables en cualquier función. Sin embargo, la función *ordenar()* no tiene una declaración *global men, may*. Esto implica que las asignaciones que se están haciendo en el bloque de esta función sobre las variables *men* y *may*, están llevando al intérprete a considerarlas como *variables nuevas y locales a ordenar()*.

La situación es que entonces, *existen dos pares de variables* llamadas *men* y *may*: en el script principal y en la función *calcular\_areas()* son visibles y utilizables las variables *men* y *may* que se definieron en el mismo script principal (que sólo por eso son globales y el valor de ambas es cero). Pero en la función *ordenar()* en este momento hay *otras dos* variables también llamadas *men* y *may*, diferentes de las anteriores, y de *ámbito local*. El valor de estas nuevas variables no quedará definido hasta que se invoque a la función y se asigne los valores de *a* o *b* en ellas. Para dejarlo claro: estas dos variables *no son las mismas* que se usan en el script principal y en la función *calcular\_areas()*: están ubicadas en diferentes lugares de la memoria, aunque lleven el mismo identificador.

Por lo tanto, si el programa se ejecuta así, el script principal **definirá dos variables de alcance global men y may con valor cero**. Luego de cargar los valores de *a* y *b*, **invocará a la función ordenar()**, la cual **definirá dos nuevas variables men y may, sin tocar ni alterar en absoluto a**

las dos que existen en el ámbito global. Cuando `ordenar()` asigna valores en `men` y `may`, lo está haciendo en las *variables locales*, mientras que las dos globales continúan valiendo cero.

Al terminar de ejecutarse la función `ordenar()`, sus dos variables locales desaparecen (y con ellas cualquier valor que hubiesen contenido) pero las dos globales siguen existiendo (y sus valores siguen siendo cero...). Cuando luego se invoque a la función `calcular_areas()`, hará los cálculos usando las *variables globales* (que son las únicas a las que tendría acceso) y como ambas valen cero, las dos áreas serán cero a su vez. El programa terminará siempre mostrando dos áreas iguales a cero, sin importar lo que se haya cargado en las variables `a` y `b` al inicio del mismo.

Como se ve, el sólo hecho de definir variables en el script principal no garantiza que las mismas serán utilizables en forma compartida entre todas las funciones que implemente el programador. Si cualquier función asigna un valor en una variable cuyo identificador ya existía en un ámbito global, entonces estará creando una variable local nueva, con el mismo nombre que la global, y la local siempre tendrá preferencia de uso sobre la global. Se dice en estos casos, que la *variable local* está *ocultando a la global* (y por lo tanto, impidiendo su uso en el ámbito de la función). La manera de evitar esto y darle preferencia a la global por sobre la local, es usar la declaración *global* ya citada [3].

En general, veremos que en la medida de lo posible el uso de variables globales debería tratar de evitarse en un programa, ya que suele provocar confusión justamente debido a este tipo de conflictos con las variables locales, entre otros problemas. La forma general y correcta de hacer que una función tome datos desde o envíe resultados hacia el exterior es mediante el uso de parámetros y el mecanismo de retorno, sin tener que recurrir a variables globales.

El uso de variables globales puede a veces parecer simple y directo, pero al mismo tiempo abre la puerta a potenciales problemas: como las variables globales son de uso compartido, cualquier función puede cambiar sus valores y estos cambios pueden afectar la lógica de funcionamiento en el resto de las funciones.

Otra vez: si una función necesita ciertos datos, los mismos deberían serle enviados a modo de *parámetros* (sin tener que recurrir a variables globales) y si la función necesita hacer públicos ciertos resultados, entonces puede hacerlo mediante el *mecanismo de retorno* (usando `return` para devolver uno o más resultados) sin tener que recurrir a variables globales.

Nada de lo anterior implica que el uso de variables globales esté prohibido: en determinadas circunstancias (y siempre a criterio del programador) la inclusión de una o más variables globales realmente ayuda a simplificar la estructura de ciertos programas complejos, que deberían usar listas interminables de parámetros en sus funciones para poder compartir variables. En cambio, si esas variables son globales, están disponibles sin tener que parametrizarlas y algunas funciones simplifican sus declaraciones.

No es obligatorio que una función tome parámetros o retorne valores. En algunos casos eso no será necesario. Por ejemplo, la función `test()` del programa para el problema 26 en esta misma Ficha, no recibe parámetro alguno ni utiliza `return` para devolver ningún resultado. En ese caso puntual, esa función está tomando sus datos desde el teclado de consola en forma directa (y no desde parámetros) y está publicando sus resultados en forma directa en la

consola de salida. El objetivo de la función `test()` en el ejemplo citado es justamente ese: convertirse en la función encargada de implementar la interfaz de usuario (IU), por lo que entonces es natural que recurra al teclado y a la pantalla para tomar sus datos y sacar sus resultados.

## 7.] Tratamiento de números primos.

Dedicaremos esta sección al análisis de problemas básicos referidos a los *números primos*. Recuerde que un entero positivo es *primo* si es divisible solamente por si mismo y por 1. En contrapartida, se suele designar como *número compuesto* a un número que no sea *primo*. Entre los problemas esenciales que tienen a los *números primos* como foco citamos al menos tres, que estudiaremos con detalle:

1. Determinar si un número  $n$  entero y positivo  $n$  es primo o no.
2. Dado un número entero positivo  $n$  (primo o no) encontrar el primer número primo que sea mayor que  $n$  (el primo siguiente a  $n$ ).
3. Dado un número entero positivo  $n$ , encontrar la *factorización* de  $n$  (o sea, la *descomposición de  $n$  en sus factores primos*).

Comencemos con el primero:

**Problema 28.)** *Determinar si un número entero y positivo  $n$  es primo o no. Por ejemplo, el número  $n = 28$  no es primo (ya que además ser divisible por 1 y por 28, es divisible por 2 y por 7). Pero  $n = 29$  es primo, ya que sólo es divisible por 1 y por el propio 29.*

**Discusión y solución:** Obviamente, el algoritmo básico (y clásico) para determinar si un número  $n$  es primo, es el *algoritmo de divisiones sucesivas*, que consiste en dividir a  $n$  por cada uno de los números en el intervalo  $[2..n-1]$  y comprobar si alguno de todos ellos divide a  $n$  en forma exacta. Si alguno lo hace, retornar `False` (pues  $n$  no es primo en ese caso) y si ninguno lo hace retornar `True`. Un esquema muy elemental en pseudocódigo podría ser el siguiente:

```
is_prime(n):
    1.) Para i entre 2 y n-1:
        1.1.) Si i divide a n en forma exacta:
            1.1.1.) Retorne False
    2.) Retorne True
```

Aún siendo tan básico y directo, el algoritmo anterior realiza demasiadas divisiones en el peor caso y puede mejorarse un poco. Se puede probar que si ningún número en el intervalo  $[2, n/2]$  divide a  $n$  en forma exacta, entonces ya ningún otro lo hará, por lo cual se pueden ahorrar la mitad de las divisiones. La demostración intuitiva surge de considerar que  $n/2$  es el mayor número en el intervalo  $[2..n-1]$  que puede dividir a  $n$  en forma exacta, ya que cualquier número mayor a  $n/2$  sólo puede restarse una vez de  $n$ , dejando un resto mayor a 0... Con esta simple idea el algoritmo puede reformularse así:

```
is_prime(n):
    1.) Para i entre 2 y n//2:
        1.1.) Si i divide a n en forma exacta:
            1.1.1.) Retorne False
    2.) Retorne True
```

Con otro pequeño esfuerzo matemático, podemos lograr otra mejora en la cantidad de divisiones a realizar: se puede probar también que si ningún número en el intervalo  $[2, \sqrt{n}]$

divide a  $n$  en forma exacta, entonces ya ningún otro lo hará, reduciendo mucho más la cantidad de divisiones a realizar. En efecto, si  $n$  es un número compuesto (no primo) entonces  $n$  tiene que ser el producto de dos o más números que son divisores exactos de  $n$ . Pero esto implica que al menos uno de esos factores debe ser menor que  $\sqrt{n}$ , ya que si todos fuesen mayores o iguales el producto entre ellos sería mayor a  $n$ . Con esta idea, el algoritmo puede quedar así:

```
is_prime(n):
    1.) Para i entre 2 y  $\sqrt{n}$ :
        1.1.) Si i divide a n en forma exacta:
            1.1.1.) Retorne False
    2.) Retorne True
```

Se pueden lograr algunas mejoras más si se consideran los siguientes hechos [4]:

1. El único número primo *par* es el 2.
2. Por lo tanto, todo otro número primo es *impar*.
3. Si un número  $x$  es *impar*, no puede ser dividido en forma exacta por 2 ni por ningún otro número par (ya que de lo contrario  $x$  sería par)

Con todo lo anterior se puede plantear un programa que se ejecutará en forma aceptablemente veloz, pero sólo si  $n$  es un número razonablemente pequeño y/o tiene divisores exactos también pequeños. Pero si  $n$  es muy grande (por ejemplo, si  $n$  tiene 25 o más dígitos:  $n \geq 10^{25}$ ) entonces este algoritmo será penosamente lento pues deberá ejecutar *miles de millones de divisiones* en el peor caso (si  $n$  es efectivamente primo o si su divisor más pequeño se aproxima a  $\sqrt{n}$ ). Por ejemplo, si se quiere comprobar la *primodalidad* de un número  $n$  de 35 dígitos ( $n \geq 10^{35}$ ) el algoritmo que sugerimos hará en el peor caso alrededor de  $\sqrt{10^{35}} \approx 316227766016837933$  divisiones (más de 300 mil millones de millones). Si nuestra computadora ejecutase unas mil millones de divisiones por segundo (o sea,  $10^9$  divisiones por segundo), entonces le tomará  $(316227766016837933 / 10^9)$  segundos el cálculo total en el peor caso, que equivale a alrededor de 316227766 (mas de 300 millones) de segundos. Si el alumno se toma el trabajo de convertir esa cantidad de segundos a años, comprobará que a nuestra computadora le llevará alrededor de 10 años terminar el proceso para un peor caso... y se pone peor a medida que aumenta el número de dígitos de  $n$ .

Se han planteado a lo largo de los siglos distintas ideas y algoritmos para acelerar el proceso, pero esas estrategias por ahora están fuera de nuestro campo de discusión. Será suficiente con programar tan bien como se pueda el *algoritmo de las divisiones sucesivas*. La función que sigue, toma a  $n$  como parámetro, y retorna *True* si  $n$  es primo, o *False* si no lo es. El algoritmo aplicado es la combinación de todos los elementos que hemos discutido en los párrafos anteriores (dejamos el análisis fino del código fuente para el estudiante):

```
def is_prime(n):
    # numeros negativos no admitidos...
    if n < 0:
        return None

    # por definicion, el 1 no es primo...
    if n == 1:
        return False

    # si n = 2, es primo y el unico primo par...
    if n == 2:
```

```

    return True

# si n no es 2 pero n es par, no es primo...
if n % 2 == 0:
    return False

# si llegamos aca, n es impar... por lo tanto no hace falta
# probar si es divisible por numeros pares...
# ... y alcanza con probar divisores hasta el valor pow(n, 0.5)...
raiz = int(pow(n, 0.5))
for div in range(3, raiz + 1, 2):
    if n % div == 0:
        return False

return True

```

Y a partir de esto, se puede plantear una función relativamente sencilla para el segundo problema esencial, cuyo enunciado formalizamos:

**Problema 29.)** *Dado un número entero y positivo  $n$  (que puede ser primo o no), determine el valor del primer número primo que sea mayor a  $n$ . Por ejemplo, dado  $n = 24$  (que no es primo), entonces el primer número primo mayor a 24 es 29. Y dado  $n = 29$  (que es primo) entonces el primer número primo mayor a 29 es el 31.*

**Discusión y solución:** La idea es simple y directa: la función *next\_prime(n)* usa un ciclo *while* que comienza chequeando el primer número impar  $p$  que sea mayor a  $n$ , para saber si  $p$  es primo. Si lo es, se retorna  $p$ . Y si no lo es, el ciclo suma 2 a  $p$  para obtener el siguiente impar, y hace una nueva repetición para volver a chequear su primalidad. Obviamente, para saber si  $p$  es primo se usa la misma función *is\_prime(n)* que ya hemos presentado en el problema anterior. La función *next\_prime(n)* se muestra a continuación (de nuevo, se dejan los detalles finos de código fuente para el análisis del estudiante):

```

def next_prime(n):
    # si n es menor a 2, el siguiente primo es 2...
    # ... no nos preocupa si n es negativo... buscamos primos naturales...
    if n < 2:
        return 2

    # Si n es par (puede ser n = 2, pero no nos afecta) entonces el
    # SIGUIENTE posible primo p no es 2... y es impar...
    p = n + 1
    if p % 2 == 0:
        p += 1

    # ahora p es impar... comenzar con el propio p
    # y buscar el siguiente impar que sea primo...
    while not is_prime(p):
        p += 2

    # ... y retornarlo
    return p

```

Y el último de los problemas esenciales que veremos, se enuncia del siguiente modo:

**Problema 30.)** *Dado un número entero y positivo  $n$ , encontrar y mostrar su **factorización** (también conocida como su **descomposición en factores primos**). Por ejemplo, si  $n = 28$ , su factorización es de*

*la forma  $28 = 2 * 2 * 7 = 2^2 * 7$ . Y si  $n = 13$  (que es primo) su factorización sólo incluye al propio 13 ya que  $13 = 1 * 13 = 13$ .*

**Discusión y solución:** Un problema importantísimo en aritmética y en ciencias de la computación es el de la *descomposición de un número entero en sus factores primos (o factorización de ese número)*. La factorización de un número entero positivo  $n$  es el conjunto de números  $\{p_1, p_2, \dots, p_k\}$  también enteros y positivos **pero todos primos**, tal que el producto de todos ellos es igual al número  $n$  original. El *Teorema Fundamental de la Aritmética* garantiza que esa descomposición existe y además es *única* para cualquier número  $n$  (y la primera demostración práctica de este teorema se debe a *Euclides*... quién si no...)

Piense qué tan serio es este problema si el valor de  $n$  es grande, muy grande o enorme... (por ejemplo: es fácil encontrar que la factorización de 28 es igual a  $2 * 2 * 7 = 2^2 * 7$ , pero no es tan simple ni tan rápido encontrar la factorización de un número que esté en el orden de  $10^{100}$  o  $10^{200}$ ) Obviamente, el algoritmo puede plantearse recurriendo a las soluciones que ya se han planteado para determinar si un número es primo, y para obtener el siguiente número primo mayor que otro número dado.

El tema de la *descomposición de un número en sus factores primos* es una cuestión para no tomar a la ligera. Intuitivamente, ya hemos visto que el algoritmo básico de divisiones sucesivas para determinar si un número es primo resulta en general muy ineficiente si  $n$  es muy grande. Y para descomponer un número en sus factores primos, se debe probar a dividir ese número por cada primo menor que él, con lo cual el programador debe encontrar esos primos y luego dividir. Si detectar un solo primo puede llevar tiempo, detectar varios llevará mucho más...

Hasta ahora, no se conocen algoritmos que realicen esa descomposición en forma eficiente (oportunamente hablaremos de qué significaría que un algoritmo sea eficiente... pero digamos que un algoritmo eficiente debería poder resolver el problema velozmente, sea cual sea el número  $n$ ). Ya hemos indicado que si el número  $n$  es relativamente pequeño entonces encontrar una solución es trivial. Pero para ciertos números muy grandes, los algoritmos conocidos demoran *tiempos asombrosos*.

El algoritmo que aplicaremos, se basa inicialmente en la idea de explorar sistemáticamente todos los números primos menores o iguales que  $n//2$ , de forma que cada vez que encontremos que uno de ellos divide a  $n$  en forma exacta, se muestre en consola de salida. Una primera aproximación al ciclo general, podría verse así:

```

primo = 2
while primo <= n//2:
    # si primo es divisor de n, mostrarlo...
    if n % primo == 0:
        print(primo)
    # ...en todos los casos, tomar el siguiente primo y seguir...
    primo = next_prime(primo)

```

Sabemos que todos los divisores exactos de  $n$  (distintos de 1 y  $n$ ), estarán todos en el intervalo  $[2, n//2]$ , por lo que el ciclo explora sólo ese intervalo. El primer número primo en el intervalo de búsqueda es claramente el 2, por lo que *primo* comienza valiendo 2 antes de iniciar el ciclo. En el bloque de acciones del ciclo se pregunta si  $n$  es divisible por *primo*, y en caso de serlo se muestra el valor de *primo*, ya que se habrá encontrado de esta forma un

divisor *primo* y exacto para  $n$ . Finalmente, antes de terminar el bloque del ciclo, se invoca a nuestra ya conocida función *next\_prime(n)* para obtener el siguiente número *primo* mayor a  $n$ , y se regresa al ciclo para controlar si este es todavía menor o igual que  $n//2$ .

Si bien parece sencillo y a primera vista correcto, el hecho es que el algoritmo básico anterior está planteado en forma ingenua y sólo funciona correctamente si cada primo que forma parte de la factorización de  $n$ , aparece en ella una sola vez: Por ejemplo, si  $n = 14$ , el esquema funciona y detecta correctamente que la factorización es  $(2, 7)$ , o si  $n = 15$ , detecta que se descompone en  $(3, 5)$ . Pero si el número es  $n = 28$ , cuya factorización ya citada es  $(2, 2, 7)$ , el sencillo algoritmo anterior sólo muestra  $(2, 7)$ .

El problema es que en nuestro planteo propuesto, se comprueba si *primo* es divisor exacto de  $n$  y en caso afirmativo se muestra el valor de *primo*, pero solo una vez... sin considerar que ese mismo *primo* podría dividir a  $n$  más de una vez en forma exacta. En nuestro caso, si  $n = 28$ , el proceso correcto sería dividir por 2, obteniendo un *cociente parcial* de 14 y un resto de 0. Como el resto es cero, se mostraría el 2 una vez, pero luego se intentaría dividir por 2 al *cociente parcial* anterior (que era 14), obteniendo otro *cociente parcial* de 7 y otra vez un resto de 0. El 2 volvería mostrarse, y otra vez debería comprobarse si el *último cociente parcial* (7) es divisible por 2. Como en este caso 7 no es divisible por 2, recién ahora podemos dejar *primo* = 2 de lado, y buscar el siguiente primo para repetir el proceso:

```
primo = 2
while primo <= n//2:
    # si primo es divisor de n, mostrarlo...
    # ... pero tantas veces como la division sea posible...
    cociente_parcial = n
    while cociente_parcial > 1 and cociente_parcial % primo == 0:
        print(primo)
        cociente_parcial //= primo

    # ...en todos los casos, tomar el siguiente primo y seguir...
    primo = next_prime(primo)
```

En el esquema revisado anterior, se agregó el mecanismo para controlar si un mismo *primo* divide más de una vez a  $n$ , mostrando ese *primo* tantas veces como sea necesario. Y nos queda sólo un pequeño control adicional: el ciclo más exterior, está tomando uno a uno todos los primos en el intervalo  $[2, n//2]$ , y el proceso total sólo se detendrá cuando los haya controlado exactamente a todos. Sin embargo, notemos que lo que realmente necesitamos es encontrar los primos que multiplicados entre sí den como resultado al propio  $n$ , y todos esos primos podrían encontrarse bastante antes de llegar a  $n//2$ . Sabemos por el *Teorema Fundamental de la Aritmética*, que la factorización de  $n$  existe **y es única**, por lo que una vez que hallemos **un** conjunto de primos que al multiplicarse dan como resultado a  $n$ , podremos detener el proceso sin necesidad de seguir comprobando otros primos.

Por lo tanto, podemos agregar una variable *producto* con valor inicial 1, *acumular en ella en forma multiplicativa* todos los primos que se hayan encontrado y mostrado, y comprobar si el valor de *producto* llegó a  $n$  para cortar el proceso en ese momento:

```
primo, producto = 2, 1
while primo <= n//2:
    # si primo es divisor de n, mostrarlo...
    # ... pero tantas veces como la division sea posible...
    cociente_parcial = n
    while cociente_parcial > 1 and cociente_parcial % primo == 0:
        print(primo)

        producto *= primo
```

```

        producto *= primo
        cociente_parcial //= primo
        if producto == n:
            return

    # ...en todos los casos, tomar el siguiente primo y seguir...
    primo = next_prime(primo)

```

Dentro del segundo ciclo `while` se ha agregado el control de la variable `producto` de acuerdo a lo dicho. El detalle a observar, es que dentro de ese ciclo se incluyó una instrucción condicional para chequear si `producto == n`, y en caso de ser cierto se interrumpe todo el proceso con una invocación a `return`. Si este bloque estuviese contenido dentro de una función (como efectivamente será el caso), entonces ese `return` dará por terminada la propia función, y no sólo al ciclo que contenía al `return`.

La función `factorization(n)` que mostramos a continuación, aplica estas ideas y completa nuestro análisis:

```

def factorization(n):
    # eliminamos casos no previstos...
    if n < 0:
        print('Se esperaba un numero positivo...')
        return

    # si n es primo, o es el 1, mostrarlo y terminar...
    if n == 1 or is_prime(n):
        print(n, end=' ')
        return

    # n no es primo...
    # ...probar a dividirlo por cada primo menor que n//2...
    primo, producto = 2, 1
    while primo <= n//2:
        # si primo es divisor de n, mostrarlo...
        # ...pero tantas veces como la division sea posible...
        cociente_parcial = n
        while cociente_parcial > 1 and cociente_parcial % primo == 0:
            print(primo, end=' ')
            producto *= primo
            cociente_parcial //= primo
        if producto == n:
            return

    # ...en todos los casos, tomar el siguiente primo y seguir...
    primo = next_prime(primo)

```

Un pequeño detalle respecto de la función `print()`: note que en la función `factorization()`, al invocar a `print()` se ha usado un parámetro adicional llamado `end`, asignado con un espacio en blanco:

```
print(primo, end=' ')
```

Ese parámetro forma parte de un esquema de parametrización especial en Python designado como **parametrización por palabra clave**, que veremos en detalle oportunamente. Por ahora, es suficiente con saber que en la función `print()` el parámetro `end` es de uso opcional, pero si se usa se puede asignar con el carácter que el programador quiera usar para indicar el final de la línea mostrada por `print()`. Por defecto, el valor de `end` es un carácter "`\n`" (salto de línea), lo cual explica por qué las líneas mostradas con `print()`

aparecen siempre a renglón seguido. Si se asigna en `end` un espacio en blanco, entonces `print()` cambiará el salto de línea por el blanco, y las salidas mostradas aparecerán en la misma línea de la consola.

Debido a que la *factorización* de un número es un proceso que hasta hoy no tiene algoritmos eficientes para resolverlo, e incluso una supercomputadora podría estar meses o años para descomponer un número muy grande (por ejemplo, un número de 200 dígitos) es que muchos *sistemas de encriptación* modernos se basan en hacer algún tipo de *factorización* en su núcleo algorítmico. El *algoritmo RSA de clave pública*<sup>3</sup>, por ejemplo, que es la base de los sistemas de encriptación usados hoy en la web para proteger todo tipo de transacciones críticas, se basa en la *factorización de números de más de 100 dígitos* generados como producto de otros dos números primos aleatorios muy muy grandes y en la seguridad de que ningún algoritmo conocido logrará descomponerlo velozmente, incluso en computadoras de última generación.

A modo de ejemplo de análisis, recordemos la función `eval(expr)` (provista por Python) que hemos presentado en la *Ficha 8*. Como vimos, esta función toma como parámetro una *cadena de caracteres que expresa una instrucción* válida en Python, analiza esa expresión, y si efectivamente es válida la ejecuta y retorna el resultado de esa expresión. Ejemplo:

```
x = eval('3 + 4**2')
print('x:', x)
```

La salida en consola del script anterior será algo como `x: 19`.

Así, podemos plantear una función `total_time(expr)` que tome como parámetro una cadena que represente una expresión cualquiera, y *retorne el tiempo de ejecución de la misma*, aplicando los recursos de medición de tiempos que también vimos en la *Ficha 8*:

```
def total_time(expr):
    t1 = time.perf_counter()
    r = eval(expr)
    t2 = time.perf_counter()
    return t2 - t1
```

La función `total_time(expr)` que mostramos, mide (en la forma ya explicada) el tiempo que demora la ejecución de la expresión dada por `expr` y lo retorna. Podemos usar entonces esta función para medir tiempos de cualesquiera procesos, sin más que pasarle una cadena de caracteres que exprese al proceso.

Aprovechando lo anterior, en el siguiente script usamos la función `total_time(expr)` de forma de medir el tiempo de ejecución de la función `factorization(n)` para los tres números que se ven en el código fuente:

```
print('Factorizacion de 1234578:', end=' ')
t1 = total_time('factorization(1234578)')
print('\nDemora en segundos:', t1)

print('\nFactorizacion de 12345785:', end=' ')
t2 = total_time('factorization(12345785)')
print('\nDemora en segundos:', t2)
```

---

<sup>3</sup> La sigla *RSA* es abreviatura de los apellidos de los creadores del algoritmo: *Rivest, Shamir y Adleman*. Una breve biografía de *Adi Shamir* está disponible en nuestra Galería de Personas Célebres, en el aula virtual del curso.

```
print('\nFactorizacion de 12374578:', end=' ')
t1 = total_time('factorization(12374578)')
print('\nDemora en segundos:', t1)
```

Ninguno de los tres números es demasiado grande, y sin embargo los tiempos de ejecución son crecientes y muy considerables...:

```
Factorizacion de 1234578: 2 3 205763
Demora en segundos: 1.078471563392499
```

```
Factorizacion de 12345785: 5 2469157
Demora en segundos: 21.502046654107453
```

```
Factorizacion de 12374578: 2 6187289
Demora en segundos: 72.79415064572953
```

El primero de los tres números tiene siete dígitos, y su factorización insumió poco más de un segundo. El segundo número tiene ocho dígitos pero uno de sus factores primos es relativamente grande (por lo que la función *factorization()* tarda más en encontrarlo) y la demora fue aquí de más de 22 segundos... Y el tercer número tiene también ocho dígitos, pero uno de sus factores primos es aún mayor (casi tres veces) que en el caso anterior, provocando una considerable demora en hallarlo para la función *factorization(n)*... que llega en este caso al sorprendente tiempo de casi 73 segundos (más de un minuto... para una computadora potente...) En general, el tiempo de ejecución será mayor mientras más dígitos tenga *n*, y mayores sean los factores primos que entran en la descomposición de *n*.

Por supuesto, nuestra función *factorization(n)* aún está planteada de manera algo tosca, y realiza demasiado trabajo de *fuerza bruta*. Se podrían introducir mejoras sustanciales, por ejemplo, si se guardase de alguna forma la lista de los *k* primeros números primos encontrados, de forma de no tener que volver a buscarlos cada vez que se pruebe con un *n* diferente. Esto lograría acelerar y reducir el tiempo de ejecución, aunque al costo de utilizar memoria extra para almacenar los primos ya conocidos.

Mostramos a continuación un programa completo con todas las funciones y scripts analizados hasta aquí:

```
import time

def total_time(expr):
    t1 = time.perf_counter()
    r = eval(expr)
    t2 = time.perf_counter()
    return t2 - t1

def is_prime(n):
    # numeros negativos no admitidos...
    if n < 0:
        return None

    # por definicion, el 1 no es primo...
    if n == 1:
        return False

    # si n = 2, es primo y el unico primo par...
    if n == 2:
        return True

    # si n > 2 y es par, no es primo...
    if n % 2 == 0:
        return False

    # si n > 2 y es impar, comprobamos divisibilidad...
    for i in range(3, int(n**0.5)+1, 2):
        if n % i == 0:
            return False

    return True
```

```

if n == 2:
    return True

# si no es n = 2 pero n es par, no es primo...
if n % 2 == 0:
    return False

# si llegamos aca, n es impar... por lo tanto no hace falta
# probar si es divisible por numeros pares...
# ... y alcanza con probar divisores hasta el valor pow(n, 0.5)...
raiz = int(pow(n, 0.5))
for div in range(3, raiz + 1, 2):
    if n % div == 0:
        return False

return True


def next_prime(n):
    # si n es menor a 2, el siguiente primo es 2...
    # ... no nos preocupa si n es negativo...
    # ... buscamos primos naturales...
    if n < 2:
        return 2

    # Si n es par (puede ser n = 2, pero no nos afecta)
    # entonces el SIGUIENTE posible primo p no es 2...
    # ...y es impar...
    p = n + 1
    if p % 2 == 0:
        p += 1

    # ahora p es impar... comenzar con el propio p
    # y buscar el siguiente impar que sea primo...
    while not is_prime(p):
        p += 2

    # ... y retornarlo
    return p


def factorization(n):
    # eliminamos casos no previstos...
    if n < 0:
        print('Se esperaba un numero positivo...')
        return

    # si n es primo, o es el 1, mostrarlo y terminar...
    if n == 1 or is_prime(n):
        print(n, end=' ')
        return

    # n no es primo...
    # ...probar a dividirlo por cada primo menor que n//2...
    primo, producto = 2, 1
    while primo <= n//2:
        # si primo es divisor de n, mostrarlo...
        # ... pero tantas veces como la division sea posible...
        cociente_parcial = n
        while cociente_parcial > 1 and cociente_parcial % primo == 0:
            print(primo, end=' ')
            cociente_parcial //= primo

```

```

        producto *= primo
        cociente_parcial /= primo
        if producto == n:
            return

        # ...en todos los casos, tomar el siguiente primo y seguir...
        primo = next_prime(primo)

def test():
    print('Pruebas con numeros primos...')

    n = int(input('Ingrese un entero positivo: '))
    print('Es primo?:', is_prime(n))
    print('Siguiente primo:', next_prime(n))
    print('Factorizacion:', end=' ')
    factorization(n)

    # mostramos la factorizacion, de tres numeros grandes...
    # ...pero midiendo el tiempo de ejecucion...
    print('Factorizacion de 1234578:', end=' ')
    t1 = total_time('factorization(1234578)')
    print('\nDemora en segundos:', t1)

    print('\nFactorizacion de 12345785:', end=' ')
    t2 = total_time('factorization(12345785)')
    print('\nDemora en segundos:', t2)

    print('\nFactorizacion de 12374578:', end=' ')
    t1 = total_time('factorization(12374578)')
    print('\nDemora en segundos:', t1)

# script principal...
test()

```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [3] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: <https://docs.python.org/3/>.
- [4] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

# Ficha 10

## Programación Modular

---

### 1.] Introducción.

Existen básicamente dos motivos por los cuales se utilizan funciones (subrutinas) en un programa en cualquier lenguaje de programación [1]:

- ✓ El primer motivo es lograr *ahorro de líneas de código*, desarrollando como función a todo bloque de instrucciones que en un mismo programa se use en muchas ocasiones. Así, si en un programa se debe realizar varias veces el ordenamiento de tres o cuatro variables, pero cambiando cada vez las variables que se ordenan, se define *una sola función* (con los *parámetros* que correspondan), y *no* se escribe dos o tres veces el código para el ordenamiento. Este es el motivo por el cual originalmente se definió el concepto de función en un lenguaje de programación.
- ✓ El segundo motivo es el de permitir que un programador pueda *modularizar adecuadamente sus programas*, dividiéndolos en subproblemas que le resulten más fáciles de manejar y controlar. En este sentido, el uso de funciones no se limita sólo a evitar posibles redundancias de código, sino que también apunta a la mejor estructuración de un programa. De hecho (y como vimos), se designa como *programación estructurada* al paradigma de programación que, entre otros principios, aplica la *programación modular*, que consiste en dividir un problema en subproblemas, y estos a su vez en otros, de forma que finalmente se plantee un programa en base a funciones que resuelvan los subproblemas.

Notemos, a modo de comentario, que de acuerdo con esto último un lenguaje de programación será tanto más apto para la programación estructurada mientras más fácilmente permita que un programador la lleve a cabo. En ese sentido, puede decirse que **todo** lenguaje facilita la programación estructurada, pues todo lenguaje permite definir funciones. Sin embargo, algunos lenguajes van más allá: en el lenguaje *Python* o en el lenguaje *Pascal*, por ejemplo, se pueden definir funciones y *también se puede definir una función dentro de otra*. Esto último no es válido en la mayoría de los lenguajes: en *C / C++* una función puede llamar a otra si es necesario (como en todos los lenguajes), *pero no puede contener la declaración de otra dentro de sí*. Los lenguajes *Python* y *Pascal*, entonces, son más aptos para aplicarse en situaciones de problemas que se dividen en otros, y estos a su vez en otros, y por lo tanto favorecen mejor la programación estructurada que el lenguaje *C / C++*. Por supuesto, como se dijo, en *C / C++* (como en cualquier lenguaje, incluido *Python*) se pueden definir tantas funciones como se quiera, y luego hacer que estas se invoquen entre sí.

Hemos indicado que un *paradigma de programación* es un conjunto de reglas, convenciones y prácticas para desenvolverse con éxito en la tarea de la programación de computadoras. Hemos dicho también que uno de esos paradigmas es el que se conoce como *programación estructurada*, dentro del cual nos mantendremos a lo largo de este curso. La programación estructurada se basa en una serie de principios y buenas prácticas que apuntan primero a que el programador pueda resolver un problema descomponiéndolo en partes más simples

(que hemos designado como *subproblemas*) y segundo a que el programador pueda plantear programas conceptualmente claros, más simples de comprender y eventualmente también más simples de corregir o modificar.

Dentro de la *programación estructurada*, la estrategia de dividir un problema en subproblemas más simples, y luego escribir subrutinas en un lenguaje de programación que se correspondan con la solución a esos subproblemas, se conoce como *programación modular*. Este nombre se debe a que en general, las subrutinas o conjuntos de subrutinas que se plantean para resolver a los subproblemas detectados, suelen designarse también como *módulos de programación*.

En este punto, debe quedar claro que la *programación modular* es **uno** de los principios básicos o estrategias de la *programación estructurada*, pero no es el único: el paradigma estructurado dispone de otras convenciones de trabajo (que iremos descubriendo y analizando a medida que avancemos en el curso), tales como, por ejemplo, evitar los llamados *saltos incondicionales* dentro un bloque de instrucciones (que en muchos lenguajes pueden hacerse con órdenes de la forma "go to").

A modo de ejemplo, analizaremos más abajo un problema relativamente simple del campo de la gestión administrativa y plantearemos para ese problema una *solución modular*: intentaremos descubrir los subproblemas más relevantes, dar un algoritmo que permita resolverlos, y finalmente escribir un programa en Python que se base en el planteo modular propuesto. Si durante el desarrollo aparece algún elemento que requiera la aplicación de nuevos elementos del lenguaje y/o de alguna variante interesante de temas o elementos ya conocidos, entonces se hará una explicación puntual dentro del análisis del mismo problema. Esperamos con esto seguir contribuyendo a que los estudiantes dispongan de un pequeño "catálogo" de problemas resueltos que siempre es útil cuando se enfrentan nuevos problemas, ya que muchas veces la solución a un nuevo problema suele inspirarse en soluciones conocidas para problemas ya analizados.

El programa está planteado en forma simple, utilizando sólo estructuras condicionales (sin ciclos) debido a que el conjunto de datos a procesar es pequeño y de tamaño fijo, y además para poder centrar la discusión en la forma de dividir en subproblemas y ajustarlos al uso de funciones. Retomaremos además aquí, el uso del esquema de pseudocódigo para aclarar algunas explicaciones. El enunciado es el siguiente:

**Problema 31.) Una empresa de turismo que vende viajes para egresados de colegios secundarios, ofrece a tres cursos distintos la siguiente promoción: El costo del viaje por persona es de \$ 1360, pero si el grupo excede de las 40 personas, la empresa realiza un descuento del 5% sobre el costo total del viaje para el curso. Desarrollar un programa, que cargando la cantidad de alumnos de cada uno de los tres cursos, permita determinar:**

1. *El curso más numeroso*
2. *El monto del viaje para cada curso*
3. *El porcentaje que representa el monto del viaje del curso más numeroso sobre el total de la ganancia de la empresa.*

**Discusión y solución:** En este problema la división en subproblemas resulta imprescindible: si no se hace un planteo bien modularizado y con subprocesos claramente separados, el resultado podría ser un programa con su lógica muy intrincada, con muchos bloques de código repetidos (y por lo tanto, redundante) [1].

En cuanto a los datos, este problema sólo pide la carga de tres valores (que llamaremos *c1*, *c2* y *c3*) que representan la cantidad de alumnos en tres cursos de una escuela secundaria. El primero de los resultados pedidos es indicar cuál de esos tres cursos es el más numeroso, lo cual puede hacerse con una subrutina que compare los valores de *c1*, *c2* y *c3* y almacene un descriptor de tipo cadena de caracteres en una variable de salida (que podemos llamar *may\_cur*). Si llamamos *mayor()* a la subrutina, un primer esbozo en pseudocódigo sería:

```
mayor(c1, c2, c3):
    si c1 > c2 y c2 > c3:
        may_cur = 'Primero'
    sino
        si c2 > c3:
            may_cur = 'Segundo'
        sino:
            may_cur = 'Tercero'

    return may_cur
```

El primer resultado pedido quedaría completamente cubierto con esta subrutina. Note que como ahora sabemos que la subrutina se llamará *mayor()*, entonces el pseudocódigo comienza con el nombre de la misma y no con la palabra *algoritmo* (como hicimos en fichas anteriores).

Cuando seguimos avanzando en la lectura de los resultados a cubrir, vemos que el tercero nos pide calcular el porcentaje que el *monto* del curso más numeroso representa en el *monto total* facturado. Aún no hemos calculados los montos para cada curso, pero por el momento podemos asumir que cuando lo hagamos los tendremos asignados en otras tres variables *m1*, *m2* y *m3*. El monto total será la suma de estos tres valores, y para el porcentaje pedido tenemos que saber cuál de los tres corresponde al curso con más alumnos. La subrutina *mayor()* fue la encargada de determinar el curso más numeroso, pero en el momento en que la planteamos no tuvimos en cuenta la necesidad de conocer los montos. Para no tener que *volver a preguntar cuál de los cursos es el más numeroso*, podemos volver atrás a la subrutina *mayor()*, asumir que para ese momento ya estarán calculados los montos, y replantearla para que no sólo se guarde un descriptor del mayor, sino también el monto de ese curso en una segunda variable de salida (que llamaremos *may*):

```
mayor(c1, m1, c2, m2, c3, m3):
    si c1 > c2 y c2 > c3:
        may_cur = 'Primero'
        may = m1
    sino:
        si c2 > c3:
            may_cur = 'Segundo'
            may = m2
        sino:
            may_cur = 'Tercero'
            may = m3

    return may_cur, may
```

Con esto, la subrutina *mayor()* resuelve el primer objetivo (determinar el curso mayor) y ayuda a resolver el tercero (el porcentaje del monto del mayor sobre el total). Este último resultado quedará asignado en la variable *porc*, que puede calcularse como se ve en el

pseudocódigo de la subrutina que llamaremos *porcentaje()* (note que antes de realizar el cálculo, se verifica el valor de *mtot* para evitar un error de división por cero):

```
porcentaje(m1, m2, m3, may) :
    mtot = m1 + m2 + m3
    si mtot != 0:
        porc = may * 100 / mtot
    sino:
        porc = 0

    return porc
```

Con esto, tenemos resueltos el primero y el tercero de los requerimientos del problema. Sólo nos queda (por fin) analizar cómo hacer el cálculo del monto que debe pagar cada curso, e informar esos montos en consola de salida. En principio, cada alumno paga 1360 pesos, por lo cual el monto inicial (*m1*, *m2* y *m3*) de cada curso podría calcularse como se ve en nuestro primer planteo de la subrutina *montos()*:

```
montos(c1, c2, c3):
    m1 = c1 * 1360
    m2 = c2 * 1360
    m3 = c3 * 1360

    return m1, m2, m3
```

Sin embargo, el enunciado aclara que si el número de alumnos en un curso es mayor a 40, entonces la empresa hará un descuento del 5% sobre el total inicial a pagar. Por lo tanto, la subrutina *montos()* podría modificarse de la siguiente forma:

```
montos(c1, c2, c3):
    sea m1 = c1 * 1360
    sea m2 = c2 * 1360
    sea m3 = c3 * 1360

    si c1 > 40:
        m1 = m1 - m1*5/100
    si c2 > 40:
        m2 = m2 - m2*5/100
    si c1 > 40:
        m3 = m3 - m3*5/100

    retornar m1, m2, m3
```

Como sabemos, una instrucción de la forma

$$m1 = m1 - m1*5/100$$

actúa como un acumulador para el valor de la expresión  $m1*5/100$ , pero restando en lugar de sumar: El miembro derecho de la asignación se ejecuta primero, tomando los valores que en ese momento tengan las variables que aparecen. En este caso, la única es *m1* que al momento de ejecutar la instrucción contiene el *monto inicial* a pagar por el curso. En el miembro derecho se calcula el 5% de ese monto, y el mismo es restado del valor actual de *m1*, con lo cual el resultado es el monto inicial menos el 5% de ese mismo monto. Al asignar ese resultado en la propia variable *m1*, el valor de la misma se actualiza: deja de valer el monto inicial y pasa a valer el monto descontado. Lo mismo vale para las otras dos expresiones con *m2* y *m3*.

Con esto la lógica general del algoritmo para resolver el problema queda cerrada. El código fuente completo en Python se muestra a continuación, aplicando todas las ideas discutidas y

recordando que el script principal sólo incluirá una invocación a una función de arranque general (que llamaremos `test()`):

```
__author__ = 'Cátedra de AED'

def mayor(c1, m1, c2, m2, c3, m3):
    if c1 > c2 and c1 > c3:
        may = m1
        may_cur = 'Primero'
    else:
        if c2 > c3:
            may = m2
            may_cur = 'Segundo'
        else:
            may = m3
            may_cur = 'Tercero'

    return may_cur, may

def montos(c1, c2, c3):
    m1 = c1 * 1360
    m2 = c2 * 1360
    m3 = c3 * 1360

    if c1 > 40:
        m1 = m1 - m1/100*5

    if c2 > 40:
        m2 = m2 - m2/100*5

    if c3 > 40:
        m3 = m3 - m3/100*5

    return m1, m2, m3

def porcentaje(m1, m2, m3, may):
    mtot = m1 + m2 + m3
    if mtot != 0:
        porc = may / mtot * 100
    else:
        porc = 0

    return porc

def test():
    # título general y carga de datos...
    print('Cálculo de los montos de un viaje de estudios...')
    c1 = int(input('Ingrese la cantidad de alumnos del primer curso: '))
    c2 = int(input('Ingrese la cantidad de alumnos del segundo curso: '))
    c3 = int(input('Ingrese la cantidad de alumnos del tercer curso: '))

    # procesos... invocar a las funciones en el orden correcto...
    m1, m2, m3 = montos(c1, c2, c3)
    may_cur, may = mayor(c1, m1, c2, m2, c3, m3)
    porc = porcentaje(m1, m2, m3, may)
```

```

# visualización de resultados
print('El curso mas numeroso es el', may_cur)
print('El monto del viaje del primer curso es:', m1)
print('El monto del viaje del segundo curso es:', m2)
print('El monto del viaje del tercer curso es:', m3)
print('El porcentaje del monto del mas numeroso en el total es:', porc)

# script principal: sólo invocar a test()...
test()

```

La idea de definir una función que contenga todo el montaje inicial completo del programa (como nuestra función *test()*) para después simplemente invocar a esa función como única acción dentro del script principal es la que sugerimos ahora y aplicaremos de aquí en más en todos nuestros modelos, ejemplos y desarrollos. Como se puede observar, esto permite que el programa quede completamente modularizado, sin bloques de código "sueltos" que no podrían ser invocados desde otras funciones si el programador tuviese necesidad de hacerlo.

En el desarrollo de la solución del problema anterior, no hemos desplegado diagramas de flujo pero sí pseudocódigos: recuerde que ambas son herramientas auxiliares que los programadores usan en función de sus necesidades, y en este ejercicio el análisis de la lógica directamente desde el código fuente era simple.

## 2.] Funciones generalizadas (o reutilizables).

En fichas anteriores se presentaron y se analizaron problemas de todo tipo (del campo de la matemática, la gestión administrativa, los juegos<sup>1</sup>, la física, y otros) pero la idea de la *modularización* es siempre la misma: consiste en plantear un programa a través de *módulos* (en nuestro caso, *funciones*) que representen soluciones a cada uno de los subproblemas que se hayan detectado. A su vez, el uso de parámetros y retorno de valores permite que una función pueda ser interpretada y usada como un *proceso general*, independiente de elementos externos (como variables globales, por ejemplo): los datos que la función necesita son tomados desde sus parámetros formales, y los resultados que ella genera son devueltos al exterior mediante el mecanismo de retorno [1].

A medida que van ganando experiencia los programadores descubren que muchos procesos que fueron pensados originalmente como funciones específicas de un programa en particular, vuelven a ser útiles en otros programas para resolver otros problemas. Ese es el caso, por ejemplo, de la función para calcular el *factorial* de un número (presentada en la *Ficha 7*), que se vuelve a usar en diversos problemas estudiados en otras fichas, o de las funciones que hemos presentado en la *Ficha 9* para *tratamiento de números primos*.

Esta situación es extremadamente común (y además deseable...) y es una de las múltiples formas en que se produce lo que se conoce como *reutilización de código* (o *reuso de código*): una

---

<sup>1</sup> Hemos citado en la *Ficha 7* a la que quizás fue la primera película cuyo argumento se basaba en algún juego de computadoras (*WarGames* [o *Juegos de Guerra*] de 1983). Y a partir de entonces, aparecieron muchas más. En algunas, el argumento gira alrededor de personajes que usan juegos de computadoras y/o son expertos en ellos (como *Ender's Game* [o *El Juego de Ender*] de 2013). Pero también aparecen cada vez más películas que no tratan sobre juegos de computadoras, pero están inspiradas en algún juego muy conocido: *Prince of Persia: The Sands of Time* (o *El Príncipe de Persia: Las Arenas del Tiempo*) de 2010; *Battleship* (o *Batalla Naval*) de 2012, *Transformers* (de 2007) y las tres secuelas posteriores de esta; y muchas, pero muchas más...

misma pieza (o módulo) de software ya diseñada para un sistema, proyecto o programa, se vuelve a utilizar en forma directa y por simple mecanismo de invocación, sin cambios ni ajustes, en otro sistema, proyecto o programa. Note que en ese sentido, todas las funciones predefinidas que un lenguaje de programación provee en sus librerías estándar son reutilizables: el programador puede usarlas una y otra vez en cuanto programa lo requiera, sólo invocándolas (y eventualmente, agregando alguna instrucción adicional como *import*, en Python).

¿Cuáles son los elementos que hacen que una función sea *reutilizable*? En general la reutilización de una función depende de dos factores: por un lado, mientras menos dependiente de elementos *declarados en forma externa* sea una función, más reutilizable será. Y en forma similar, por otra parte, mientras menos dependa una función de *entradas y salidas realizadas a través del teclado y la pantalla*, más reutilizable será también. Y la forma esencial de lograr esa independencia, es recurrir al empleo de parámetros para pasar datos a la función, y al mecanismo de retorno para obtener los resultados que ella entregue.

En este contexto, una función que se ha planteado de forma de ser reutilizable se dice también una *función generalizada*, ya que puede aplicarse en forma general y sin importar las características específicas del programa que la utilice: sólo se requiere saber qué parámetros enviarle, y qué resultados devolverá. Todo lo demás, es local a la función y administrado por su bloque de acciones.

Tomemos por ejemplo nuestra ya conocida función para calcular el *factorial* de un número *n* y analicemos las siguientes variantes para ella:

Figura 1: Variantes para el planteo de la función *factorial()*.

**variante a.)**

```
def factorial():
    global n, f
    f = 1
    for i in range(2, n+1):
        f *= i
```



**variante b.)**

```
def factorial():
    n = int(input('Ingrese n: '))
    f = 1
    for i in range(2, n+1):
        f *= i
    print('Su factorial es:', f)
```



**variante c.)**

```
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
```



Generalizada - Reutilizable  
¡Buena idea!



En la figura anterior, la *variante a.)* de la función *factorial()* está planteada de forma de usar *variables globales*, lo cual marca que *depende de elementos que están definidos fuera de ella* (como la variable *n*) o que son accesibles desde fuera de ella (como la variable *f*). El problema con esta *dependencia externa*, es que para poder usar esta función en cualquier programa, el mismo debe tener definidas las dos variables *n* y *f* en algún lugar *externo a la función*, pues de otro modo esta no tendrá disponibles los datos que necesita (la variable *n* en este caso) y/o no tendrá forma de comunicar sus resultados (que en este caso dejó almacenados en la variable *f*). Conclusión: la función así planteada *no es reutilizable* en forma directa (o tiene menos probabilidad de ser reusable). No puede simplemente ser importada a un programa cualquiera e invocarse sin hacer cambios y agregados en el código fuente del programa y/o la propia función.

La *variante b.)* que se muestra en la misma figura no usa variables globales ni depende de elemento alguno definido fuera de la función, por lo que podría parecer una buena solución. Sin embargo, el hecho es que esta segunda versión toma sus datos (el valor de *n*) desde el teclado de la consola estándar, y envía sus resultados (la variable *f*) directamente a la consola de salida estándar. El problema con esta *dependencia de la interfaz de usuario* es más sutil: para invocar a esta función, necesaria y forzosamente se debe aceptar que la misma cargue por teclado el valor de *n* y muestre el valor de *f* por pantalla. Pero si en el programa en el cual se desea reusar la función el valor de *n* estuviese ya definido y asignado como resultado de un cálculo previo, por ejemplo, entonces la carga por teclado exigida por nuestra nueva versión sería completamente inútil y nos obligaría a cambiar el código fuente, para eliminar esa carga... Y algo parecido ocurriría con la visualización del valor de *f* al final de la función. Además, existe otro inconveniente: la función está asumiendo que la interfaz de usuario estará basada en el uso del teclado y la pantalla de la consola estándar, pero si la idea fuese usar esa función en un programa pensado para una interfaz visual (una IGU con ventanas y objetos gráficos controlados por el mouse, por ejemplo), entonces la función no tendría cabida y debería ser completamente rediseñada. El *problema de la dependencia de una función o módulo de software respecto de la interfaz de usuario empleada*, también se conoce como *acoplamiento entre procesos e interfaz de usuario*. Conclusión: otra vez, la función así planteada no es reutilizable en forma directa y/o tiene menos posibilidades de ser reutilizada.

Finalmente, la versión que se muestra en la *variante c* de la Figura 1 usa *parámetros* para captar sus datos desde el exterior (la *variable formal n* en este caso), con lo cual no importa cuáles sean las *variables actuales* que se envíen al invocarla ni dónde hayan sido definidas, ya que la función *tomará copias* de sus valores y los asignará en sus *parámetros formales* (que a los efectos, son variables locales de la función). En forma similar, la función devuelve con el *mecanismo de retorno* los resultados que haya generado y que tiene almacenados en variables locales propias de la función (la variable *f* en este caso). *No hay dependencia externa*, ya que todas las variables que usa la función son locales a ella, y si es necesario compartir valores con el exterior se hace mediante sus parámetros y sus retornos. Por otra parte, la función *tampoco presenta acoplamiento entre procesos e interfaz* de usuario: sus datos son tomados de sus *parámetros formales* (y no desde el teclado u otro dispositivo) y sus resultados salen al exterior mediante *return* (y no directamente a la pantalla). Conclusión: *la función está diseñada en forma genérica*, sin dependencias, y puede ser reutilizada en forma amplia, sin restricciones, en distintos programas y situaciones.

Obviamente, no siempre el programador necesitará diseñar funciones o módulos genéricos y reusables. En algún momento durante el desarrollo de un programa casi seguro deberá incluir una o mas funciones que interactúen con el usuario mediante la interfaz (en algún momento el usuario deberá introducir los datos y ver los resultados en pantalla...) pero la idea es tratar de delimitar esa responsabilidad a unas pocas funciones especialmente dispuestas para ello. Eso es lo que típicamente hemos hecho hasta ahora con nuestra clásica función *test()* (o como sea que la llame el programador) para *lanzar el programa*: hemos tratado de incluir en esa función (o en alguna auxiliar de ella) todas las tareas de carga por teclado y visualización por pantalla.

Así, puede notarse que en un programa con estructura relativamente compleja habrá un *primer grupo de funciones cuyo objetivo será la administración de la interfaz de usuario*, y un *segundo grupo de funciones cuyo objetivo será resolver los problemas específicos del enunciado o requerimiento planteado*. Las primeras, necesariamente tendrán algún tipo de acoplamiento entre procesos e interfaz. Pero las segundas, deberían intentar ser diseñadas en forma genérica y al menos estas serían reusables cuando fuese necesario.

Note que una ventaja de esta estrategia, es que si el programa está orientado a una interfaz de consola estándar, y luego se decide hacer un *upgrade* o mejora pasándolo a una interfaz gráfica con ventanas y objetos visuales de alto nivel, entonces el programador sólo deberá cambiar las funciones del *primer grupo*, pero no debería tener que hacer ningún cambio en las del *segundo grupo*.

Proponemos el análisis y resolución del siguiente problema, especialmente pensado para resaltar los conceptos que hemos expuesto hasta aquí:

**Problema 32.) Desarrollar un programa controlado por menú de opciones, que incluya opciones para realizar las siguientes tareas:**

1. *Cargar un valor entero n por teclado, validando que sea mayor que 0, y mostrar todos los números impares y múltiplos de 3 que haya en el intervalo [1, n] (ambos incluidos).*
2. *Cargar dos valores enteros a y b por teclado, validando que  $1 < a < b$ , y determinar si existe algún número primo en el intervalo  $[a, b]$ . Si existe alguno, muestre el primero que encuentre. Si no, informe con un mensaje.*
3. *Cargar por teclado una secuencia de números uno a uno, cortando el proceso cuando el número cargado sea el 0. Determinar si todos los números entraron ordenados de menor a mayor.*

**Discusión y solución:** El planteo de la estructura del *menú de opciones* fue analizado en la *Ficha 8*, por lo que remitimos al estudiante a ese contexto por si necesita revisarlo. El menú estará controlado por la siguiente función *menu()*, que no presenta dificultad alguna, y será esa función la que se invoque desde el script principal para comenzar el programa:

```
def menu():
    # titulo general...
    print('Menú de opciones y funciones generalizadas')

    op = 1
    while op != 4:
        # visualizacion de las opciones...
        print('1. Impares multiplos de 3')
        print('2. Primos en un intervalo')
        print('3. Secuencia ordenada')
```

```

print('4. Salir')
op = int(input('Ingrese el numero de la opcion elegida: '))

# chequeo de la opcion elegida...
if op == 1:
    opcion1()
elif op == 2:
    opcion2()
elif op == 3:
    opcion3()

# script principal...
menu()

```

Claramente esta función es una de las que tendrá como objetivo la gestión de la interfaz de usuario, y es de esperar entonces que realice operaciones de visualización por pantalla y carga por teclado.

Nos concentraremos ahora en la forma de resolver cada tarea pedida, de ser posible con funciones generalizadas. El primer requerimiento es mostrar todos los impares múltiplos de 3 en el intervalo  $[1, n]$ , cargando  $n$  por teclado. La función *opcion1()* será invocada si el usuario elige la opción 1 del menú:

```

def validar_mayor_que(inf):
    n = inf - 1
    while n <= inf:
        n = int(input('Valor (mayor que ' + str(inf) + ' por favor...): '))
        if n <= inf:
            print('Error... se pidió > ' + str(inf), '... cargue de nuevo...')
    return n

def opcion1():
    n = validar_mayor_que(0)
    res = odd_multiples(n)
    print('Intervalo analizado: [ 1', ', ', n, ', ]')
    print('Impares multiplos de 3 en ese intervalo:', res)

```

La carga por teclado del valor  $n$  se hace a través de la función auxiliar *validar\_mayor\_que(inf)*, la cual toma como parámetro un número  $inf$ , y carga un valor por teclado controlando que ese valor sea mayor que  $inf$ . Como el valor  $inf$  viene parametrizado, la misma función puede usarse en cuanta situación requiera cargar por teclado un número controlando que sea mayor que otro dado.

Luego, la función *opcion1()* invoca a la función *odd\_multiples(n)*, la cual puede plantearse, en principio, en forma simple y directa en forma similar a la que sigue para encontrar los números impares múltiplos de 3 (*considere que la función que sigue aún no es la versión definitiva*):

```

# versión preliminar... será modificada!!!
def odd_multiples(n):
    print('Impares y multiplos de 3 en [ 1', ', ', n, ', ]:')
    for i in range(1, n+1):
        if i % 2 == 1 and i % 3 == 0:
            print(i)

```

En esta primera versión, la función toma como parámetro el valor  $n$  que indica el límite derecho del intervalo a analizar, y con un ciclo *for* explora sistemáticamente todos los números  $i$  en  $[1, n]$ , chequeando uno por uno si efectivamente  $i$  es impar ( $i \% 2 == 1$ ) y a la

vez múltiplo de (o divisible por) 3 ( $i \% 3 == 0$ ). En caso de serlo, el valor  $i$  se muestra en consola de salida.

La función cumple con lo pedido pero hay varios problemas con ella: el primero es que tiene dependencia con la interfaz de usuario (acoplamiento entre interfaz y procesos). No siempre será sencillo eliminar ese problema, sobre todo en las primeras semanas de un curso introductorio cuando aún no se cuenta con todas las herramientas que permitirían hacerlo en forma simple, pero en este caso tenemos una vía: en lugar de mostrar cada número dentro de la función, podemos ir armando una *tupla* que contenga a esos números, y retornar esa tupla al final del ciclo. La visualización de la tupla (y de paso, también la visualización del mensaje inicial de la función) se deja para la función que invoque a *odd\_multiples(n)* (en nuestro caso, la función *opcion1()* del ciclo del menú):

```
def odd_multiples(n):
    r = ()
    for i in range(1, n+1):
        if i % 2 == 1 and i % 3 == 0:
            r = r + i,
    return r
```

En esta segunda versión no hay acoplamiento de procesos e interfaz (ni dependencia externa). El proceso comienza definiendo una tupla vacía  $r$ , en la cual se irán agregando los números a medida que se encuentren. El ciclo *for* itera sobre el *range(1, n+1)* y cada valor  $i$  que sea impar y divisible por 3 se añade a la tupla  $r$  mediante la instrucción

$r = r + i,$

que es equivalente a:

$r = r + (i,)$

y también a:

$r += i,$     o bien a:     $r += (i,)$

Cualquiera de las cuatro expresiones hace que  $r$  pase a referir a una nueva tupla, que contiene lo mismo que ya contenía la original, pero ahora con el valor de  $i$  añadido al final de la tupla. Observe que el operador  $+$  permite *unir tuplas para formar una nueva* (del mismo modo que permite unir cadenas de caracteres), pero cada uno de los operandos debe ser a su vez una tupla [2] [3]. En nuestro caso, la expresión:

$r = r + i$

sin la coma a la derecha de la variable  $i$  estaría intentando unir la tupla  $r$  con el número entero  $i$ , y se produciría un error de intérprete:

```
Traceback (most recent call last):
  File "C:/Ejemplo/prueba8.py", line 191, in <module>
    menu()
  File "C:/ prueba8.py", line 14, in odd_multiples
    r = r + i
TypeError: can only concatenate tuple (not "int") to tuple
```

Al finalizar el ciclo *for*, la función *odd\_multiples(n)* retorna la tupla completa, y con esto se ha eliminado cualquier acoplamiento procesos/interfaz. Sin embargo, habíamos sugerido que la función original tenía varios problemas, y el hecho es que aún subsiste uno: así como está planteada, la función *hace demasiado trabajo extra*, y si el valor de  $n$  fuese realmente grande (recuerde el problema de la factorización de números enteros...) la demora en el tiempo de

ejecución podría comenzar a notarse. Como dijimos, posiblemente en este problema sencillo esa demora sería insignificante incluso para valores grandes de  $n$ , pero lo importante es que el programador se vaya formando de manera de saber identificar situaciones en las que podría ahorrar tiempo, y las resuelva tan eficientemente como se pueda en preparación a futuros problemas más exigentes.

En este caso, el ciclo *for* está tomando todo valor posible dentro del intervalo  $[1, n]$ , aún sabiendo de antemano que sólo nos interesan los impares. Eso podría ajustarse en forma simple, haciendo que el *range* generado sólo contenga a los impares del intervalo citado:

```
def odd_multiples(n):
    r = ()
    for i in range(1, n+1, 2):
        if i % 3 == 0:
            r = r + i,
    return r
```

En la forma que acabamos de mostrar, el rango generado es de la forma  $(1, 3, 5, 7, \dots)$  y contiene solamente los números impares pedidos. El ciclo, por lo tanto, reduce la cantidad de repeticiones de  $n$  a  $n/2$  (ya que tiene ahora la mitad de los números originales). Y como ahora sabemos que cada valor  $i$  tomado por el ciclo es efectivamente impar, ya no es necesario preguntar si  $i$  es impar en cada vuelta... sólo se chequea si  $i$  es múltiplo de 3.

Podría parecer que eso es todo. Pero todavía nos quedan algunas posibilidades de reducción de tiempo y/o código fuente. Por lo pronto, si sólo nos interesan los impares múltiplos de 3, no hay razón para que el *range* comience desde 1: *podría comenzar directamente desde el propio 3* y seguir desde allí:

```
for i in range(3, n+1, 2):
```

Y otra vez... nos interesan sólo los impares *múltiplos de 3*... Nos preguntamos si habrá forma de predecir dónde estarán esos números dentro de  $[1, n]$ , sabiendo que partimos desde el 3 (que es el primero que cumple la condición) y sólo iteramos sobre los impares. Un rápido análisis del rango iterado nos muestra lo siguiente:

$(3, 5, 7, 9, 11, 13, 15, 17, 19, 21, \dots)$

Empezando desde el 3, el siguiente impar múltiplo de 3 es el 9 (o sea,  $3 + 6$ ). Y el que sigue es 15 (que es  $9 + 6$ ). El próximo es 21 (que es  $15 + 6$ )... y todo parece apuntar a que la regla es partir desde el 3 y sumar 6 en forma progresiva. El siguiente ciclo *for*:

```
for i in range(3, n+1, 6):
```

hace exactamente eso, generando el range  $(3, 9, 15, 21, 27, \dots)$  que sólo contiene impares múltiplos de 3. Por lo tanto, el ciclo hará aún menos repeticiones si recorre ese rango (aproximadamente un tercio de la mitad de  $n$ ) y ni siquiera tendrá que seguir preguntando si  $i$  es múltiplo de 3:

```
def odd_multiples(n):
    r = ()
    for i in range(3, n+1, 6):
        r += i,
    return r
```

La regla de *comenzar desde 3 y sumar 6 en forma progresiva* puede formalizarse: todo número par es de la forma  $2k$  con  $k$  entero (el 2 es necesariamente un factor de su

descomposición prima). Por lo tanto, un número impar es de la forma  $2k + 1$  (si un número de esta forma se divide por 2, necesariamente deja un resto de 1). Y todo impar  $i$  que a la vez sea múltiplo de 3, será entonces de la forma  $i = 3(2k + 1)$  (claramente, si un número de esta forma se divide por 3 deja un resto de 0) lo que equivale a  $i = 6k + 3$ . Con  $k = 0, 1, 2, \dots$  la sucesión que se obtiene es de la forma  $(3, 9, 15, 21, \dots)$  que es la que estábamos buscando.

Por razones de claridad, la anterior será nuestra versión final para la función `odd_multiples(n)`. Sin embargo, note que la tupla `r` finalmente generada y retornada, puede ser directamente creada a partir del `range` que está siendo iterado por el ciclo, usando la función `tuple()` que citamos más arriba como una de las formas de asignar una tupla [2]:

```
r = tuple(range(3, n+1, 6))
```

En este caso, la función `tuple()` toma cada uno de los valores del `range(3, n+1, 6)` y los agrega a la tupla en el mismo orden en que figuran en el `range()`. Por lo tanto, la función `odd_multiples(n)` en realidad ni siquiera necesitaría el `for` para iterar el `range()`: bastaría con retornar directamente la tupla generada:

```
def odd_multiples(n):
    r = tuple(range(3, n+1, 6))
    return r
```

En rigor, el tiempo que llevaría ejecutar esta función estaría en el mismo orden aproximado que el que llevaría ejecutar la versión anterior, con el `for` iterando el `range()` y armando la tupla un número a la vez, ya que la función `tuple()` hace internamente ese mismo trabajo.

La segunda tarea pedida en el enunciado general del problema era determinar si en el intervalo  $[a, b]$  existe al menos un número primo, y mostrar en ese caso el primero que se encuentre. Está claro que para resolver este problema, podemos *reutilizar* las funciones `is_prime(n)` y `next_prime(n)` que ya hemos analizado y presentado en la *Ficha 9*, y esa reutilización será perfectamente posible ya que ambas funciones se plantearon originalmente en forma genérica (sin dependencias externas y sin acoplamiento procesos/interfaz, gracias al uso correcto de parámetros y retornos).

La función `opcion2()` será la que se invoque cuando el usuario elija la opción 2 del menú, y será la encargada de leer desde el teclado los valores  $a$  y  $b$ , e invocar luego a la función que diseñemos para buscar el primer primo en  $[a, b]$ :

```
def opcion2():
    a = validar_mayor_que(1)
    b = validar_mayor_que(a)
    primo = search_prime(a, b)
    print('Intervalo analizado: [', a, ',', b, ']')
    if primo is not None:
        print('Hay al menos un numero primo en ese intervalo:', primo)
    else:
        print('No hay numeros primos en ese intervalo')
```

La función `opcion2()` carga el valor de  $a$ , usando la misma función `validar_mayor_que()` que se usó en la carga de  $n$  para el problema anterior, pero ahora controlando que  $a$  sea mayor a 1. De forma similar, se invoca otra vez a `validar_mayor_que()` para cargar  $b$ , controlando ahora que  $b$  sea mayor que  $a$ ...

La función `search_prime(a, b)` será la encargada de cumplir el requerimiento de buscar el primer primo en  $[a, b]$ , que en realidad es muy sencillo si se cuenta con la función `next_prime(n)` ya programada (cosa que hicimos en la *Ficha 9*):

```
def search_prime(a, b):
    p = next_prime(a-1)
    if p <= b:
        return p

    return None
```

Como el propio valor  $a$  podría ser primo, invocamos a *next\_prime()* pasando como parámetro el valor  $a-1$ . Con esto, la función retornará el primer número primo que encuentre que sea *mayor que  $a-1$*  (y si  $a$  es primo, será justamente  $a$  el número retornado). Sea cual fuese el primo  $p$  retornado, queremos saber si está en el intervalo  $[a, b]$ . Ya sabemos que  $p \geq a$  por lo dicho anteriormente. Por lo tanto, sólo nos queda comprobar si a la vez  $p$  es menor o igual que  $b$  (cosa que se hace con la instrucción condicional dentro de la función). Si efectivamente es  $p \leq b$ , se retorna  $p$  y se cumple así con el requerimiento: el valor  $p$  es el primer primo dentro de  $[a, b]$  (podría haber otros primos mayores que  $p$  dentro del mismo intervalo, pero sólo se pidió el primero). Pero si  $p$  fuese mayor que  $b$ , entonces ese primer primo encontrado estaría fuera de  $[a, b]$  y la función en ese caso retorna *None*.

La función *opcion2()* invoca a *search\_prime(a, b)* asignando el valor retornado en la variable *primo*:

```
primo = search_prime(a, b)
```

y luego **controla el valor almacenado en esa variable**, para saber si existía o no un primo en el intervalo [3] [2]:

```
if primo is not None:
    print('Hay al menos un numero primo en ese intervalo:', primo)
else:
    print('No hay numeros primos en ese intervalo')
```

El valor *None* puede ser usado a modo de *flag* o *bandera* (como aquí) para indicar un resultado exitoso o fallido, pero para chequear si una variable vale *None* o vale un valor distinto de *None*, debe usarse el operador *is* o la combinación *is not*:

```
if x is None:
    # x vale None en esta rama...

if t is not None:
    # t no vale None en esta rama...
```

Finalmente, el tercer requerimiento del enunciado del problema era cargar una secuencia de números, cortando cuando llegue el 0, y simplemente determinar si esa secuencia entró ordenada de menor a mayor o no. La función *opcion3()* se activará cuando el usuario elija la opción 3 del menú, invocará a su vez a la función *check\_order()* y mostrará un mensaje avisando el resultado del proceso:

```
def opcion3():
    ok = check_order()
    if ok:
        print('La secuencia cargada estaba ordenada de menor a mayor')
    else:
        print('La secuencia cargada NO estaba ordenada de menor a mayor')
```

La función *check\_order()* usa un ciclo de carga por doble lectura para ingresar la secuencia a razón de un número por vuelta en la variable *num* [1]. Se usa una segunda variable *anterior* para almacenar en ella el valor de la vuelta anterior del ciclo. Si en cualquier vuelta de ese ciclo, el número cargado en esa misma vuelta en *num* fuese menor que el que se había cargado en la vuelta anterior (y que tenemos en la citada variable *anterior*), entonces

sabremos que la secuencia cargada no está ordenada (al menos dos de sus números estaban en orden invertido). La función usa un *flag* llamado *ok* para indicar en todo momento lo que se sabe de la secuencia: el valor *True* se usa para indicar que hasta ese momento la secuencia está ordenada, y el valor *False* se usa para indicar que se ha detectado al menos un par de valores invertidos:

```
def check_order():
    # asumimos que la secuencia está ordenada...
    # ... ya que todavía no hemos cargado nada...
    # ... y la secuencia vacía de hecho está ordenada...
    ok = True

    # cargamos el primer numero...
    num = int(input('Cargue el primer valor (con 0 corta): '))

    # tomamos el primero como el "anterior" para la primera vuelta...
    anterior = num

    # ciclo de carga...
    while num != 0:

        # si el que tenemos es menor que el anterior,
        # entonces la secuencia está desordenada...
        # ... cambiar el flag ok a false
        if num < anterior:
            ok = False

        # almacenar el actual (num) como "anterior"
        # para la vuelta que sigue...
        anterior = num

        # cargamos el siguiente numero para continuar el ciclo...
        num = int(input('Cargue el siguiente valor (con 0 corta): '))

    # retornar el flag con el estado final...
    # True: estaba ordenada - False: estaba desordenada
    return ok
```

Note un detalle: la función *check\_order()* que hemos sugerido aquí está planteada con un evidente acoplamiento entre procesos e interfaz de usuario. En principio, la idea era que ese acoplamiento se permitiese sólo en aquellas funciones que estaban destinadas a la interfaz con el usuario (como la función *menu()*, y las funciones *opcion1()*, *opcion2()*, *opcion3()* y *validar\_mayor\_que()*), pero se eliminase en las funciones cuyo objetivo fuese el de resolver el problema o requerimiento central (como las funciones *odd\_multiples()* y *search\_prime()* o sus auxiliares *is\_prime()* y *next\_prime()*).

El hecho es que esa idea es conceptualmente correcta, y el programador debería hacer el mayor de sus esfuerzos por respetarla; pero en este caso el proceso está profundamente mezclado con la carga desde el teclado y hacer la separación implica la aplicación de herramientas del lenguaje que aún no hemos presentado en el curso (por ejemplo, el uso generalizado de estructuras de datos mutables y de tamaño adaptativo, como los *arrays dinámicos* (llamados *listas* en Python)). Por este motivo, aceptaremos como versión final en este análisis para la función *check\_order()* a la que acabamos de discutir, y oportunamente se estudiarán técnicas que permitirán desacoplar los procesos y la interfaz de usuario también en casos como este.

El programa completo en su versión final, sería entonces el siguiente:

```

__author__ = 'Catedra de AED'

# funciones "grupo 2": genéricas, para resolver cada problema...
def is_prime(n):
    # numeros negativos no admitidos...
    if n < 0:
        return None

    # por definicion, el 1 no es primo...
    if n == 1:
        return False

    # si n = 2, es primo y el unico primo par...
    if n == 2:
        return True

    # si no es n = 2 pero n es par, no es primo...
    if n % 2 == 0:
        return False

    # si llegamos aca, n es impar... por lo tanto no hace falta
    # probar si es divisible por numeros pares...
    # ... y alcanza con probar divisores hasta el valor pow(n, 0.5)...
    raiz = int(pow(n, 0.5))
    for div in range(3, raiz + 1, 2):
        if n % div == 0:
            return False

    return True

def next_prime(n):
    # si n es menor a 2, el siguiente primo es 2...
    # ... no nos preocupa si n es negativo...
    # ... buscamos primos naturales...
    if n < 2:
        return 2

    # Si n es par (puede ser n = 2, pero no nos afecta)
    # entonces el SIGUIENTE posible primo p no es 2...
    # ...y es impar...
    p = n + 1
    if p % 2 == 0:
        p += 1

    # ahora p es impar... comenzar con el propio p
    # y buscar el siguiente impar que sea primo...
    while not is_prime(p):
        p += 2

    # ... y retornarlo
    return p

def odd_multiples(n):
    # asignar en r una tupla vacia...
    r = ()

    # iterar el rango de valores impares entre 3 y n...
    # ...y tomar solo los multiplos de 3...
    # ...que comienzan en 3 y siguen uno cada 6...
    for i in range(3, n+1, 6):
        # agregar i a la tupla r...
        r += i,

    # retornar la tupla con los números encontrados
    return r

```

```

def search_prime(a, b):
    # tomar el siguiente primo "p" desde a-1...
    # que puede ser el propio "a"...
    # ... u otro mayor que "a"...
    p = next_prime(a-1)

    # ... si p es menor o igual que b, retornar p...
    # ... ya que entonces a <= p <= b...
    if p <= b:
        return p

    # ... y si p > b, pues retornar None...
    return None

def check_order():
    # asumimos que la secuencia esta ordenada...
    # ... ya que todavia no hemos cargado nada...
    # ... y la secuencia vacia de hecho esta ordenada...
    ok = True

    # cargamos el primer numero...
    num = int(input('Cargue el primer valor (con 0 corta): '))

    # tomamos el primero como el "anterior" para la primera vuelta...
    anterior = num

    # ciclo de carga...
    while num != 0:

        # si el que tenemos es menor que el anterior,
        # entonces la secuencia esta desordenada...
        # ... cambiar el flag ok a false
        if num < anterior:
            ok = False

        # almacenar el actual (num) como "anterior"
        # para la vuelta que sigue...
        anterior = num

        # cargamos el siguiente numero para continuar el ciclo...
        num = int(input('Cargue el siguiente valor (con 0 corta): '))

    # retornar el flag con el estado final...
    # True: estaba ordenada - False: estaba desordenada
    return ok

# funciones "grupo 1": gestionan la interfaz de usuario...
def validar_mayor_que(inf):
    n = inf - 1
    while n <= inf:
        n = int(input('Valor (>' + str(inf) + ' por favor...): '))
        if n <= inf:
            print('Error... se pidió >' + str(inf), '... cargue de nuevo...')
    return n

def opcion1():
    n = validar_mayor_que(0)
    res = odd_multiples(n)
    print('Intervalo analizado: [ 1, , , n, ]')
    print('Impares multiplos de 3 en ese intervalo:', res)

def opcion2():

```

```

a = validar_mayor_que(1)
b = validar_mayor_que(a)
primo = search_prime(a, b)
print('Intervalo analizado: [', a, ',', b, ']')
if primo is not None:
    print('Hay al menos un numero primo en ese intervalo:', primo)
else:
    print('No hay numeros primos en ese intervalo')

def opcion3():
    ok = check_order()
    if ok:
        print('La secuencia cargada estaba ordenada de menor a mayor')
    else:
        print('La secuencia cargada NO estaba ordenada de menor a mayor')

def menu():
    # titulo general...
    print('Menu de opciones y funciones generalizadas')

    op = 1
    while op != 4:
        # visualizacion de las opciones...
        print('1. Impares multiplos de 3')
        print('2. Primos en un intervalo')
        print('3. Secuencia ordenada')
        print('4. Salir')
        op = int(input('Ingrese el numero de la opcion elegida: '))

        # chequeo de la opcion elegida...
        if op == 1:
            opcion1()
        elif op == 2:
            opcion2()
        elif op == 3:
            opcion3()

# script principal...
menu()

```

### 3.] Elementos de cálculo combinatorio.

El *cálculo combinatorio* (o también *análisis combinatorio*) es una rama de las matemáticas (y en particular de la matemática discreta) cuyo objeto general es el estudio de las posibilidades en que pueden combinarse los  $n$  elementos de un conjunto, agrupándolos de maneras diferentes (ya sea en grupos de  $m$  elementos con  $m < n$ , o en grupos de exactamente  $n$  elementos, etc.) En algunos problemas importará enumerar o caracterizar cada una de las formas posibles de combinación, y en otros será suficiente sólo con indicar cuántas maneras diferentes existen para hacer esas combinaciones [4].

Un elemento básico del cálculo combinatorio es el llamado *Principio Fundamental de Conteo* (o *Principio Básico de Conteo*) que expresa *cuántas resultados posibles existen de combinar k eventos*: Si un evento cualquiera puede presentarse en  $e_1$  formas diferentes, y luego otro evento puede presentarse en  $e_2$  formas distintas, y otros eventos sucesivos pueden aparecer en  $e_3, e_4, \dots, e_k$  formas diferentes, entonces la cantidad total de formas en que pueden combinarse los  $k$  eventos presentados es igual a:

$$t = e_1 * e_2 * e_3 * \dots * e_k$$

Un ejemplo aclara la idea: suponga que en un comercio de venta de prendas de vestir se tiene que preparar un maniquí en vidriera, y para hacerlo deben colocarle al menos un pantalón, un par de zapatos y una camisa. Si el empleado encargado de esa tarea tiene siete diferentes modelos de pantalones ( $e1 = 7$ ), cuatro diferentes modelos de zapatos ( $e2 = 4$ ) y ocho modelos distintos de camisas ( $e3 = 8$ ), entonces ese empleado tendrá que elegir entre una cantidad total  $t$  de variantes igual a:

$$\begin{aligned} t &= e1 * e2 * e3 \\ t &= 7 * 4 * 8 \\ t &= 224 \text{ resultados posibles} \end{aligned}$$

Otra necesidad común del análisis combinatorio consiste en determinar en cuántas formas diferentes pueden arreglarse o combinarse los  $n$  elementos de un conjunto, tomados todos a la vez, pero de forma que cada uno aparezca una sola vez en cada arreglo (no valen repeticiones), y de forma que el orden importe: dos variaciones con los mismos elementos pero en diferente orden, se consideran distintas. Esto se conoce como el [número de permutaciones de  \$n\$  elementos, tomados de a  \$n\$](#) , y podemos denotarlo como  $p(n, n)$ .

A modo de ejemplo, suponga que se desea saber cuántos posibles números diferentes pueden formarse con los dígitos 1, 2, 3 y 4, sin repetir ninguno de ellos. Evidentemente, cada arreglo o permutación de estos cuatro números tendrá un total de cuatro dígitos. Para el primer dígito de la izquierda, se puede elegir a cualquiera de los cuatro originales, por lo cual tenemos 4 posibles elecciones de partida.

Pero para el segundo dígito, como no podemos repetir ninguno, sólo podremos elegir alguno que no sea igual al que se eligió como primer dígito, quedando entonces 3 posibilidades adicionales para cada una de las 4 iniciales. Usando el mismo argumento, el tercer dígito sólo tendrá 2 alternativas y el cuarto dispondrá de sólo una. Se deduce que la cantidad de permutaciones  $p(4, 4)$  (cuatro dígitos diferentes tomados de a cuatro, sin repetir dígitos) será igual a  $4 * 3 * 2 * 1 = 4! = 24$  permutaciones.

Esto puede verse con mucha claridad usando un diagrama tipo árbol para exponer las permutaciones [\[5\]](#) (ver *Figura 2*). Entonces es relativamente simple probar que:

$$p(n, n) = n!$$

Efectivamente: si en cada arreglo o permutación puede aparecer sólo una vez cada uno de los  $n$  elementos originales y el orden importa, entonces la posición de partida puede ocuparse de  $n$  formas distintas. La segunda posición sólo puede ocuparse de  $n-1$  formas diferentes con el resto de los números, y la progresión será:

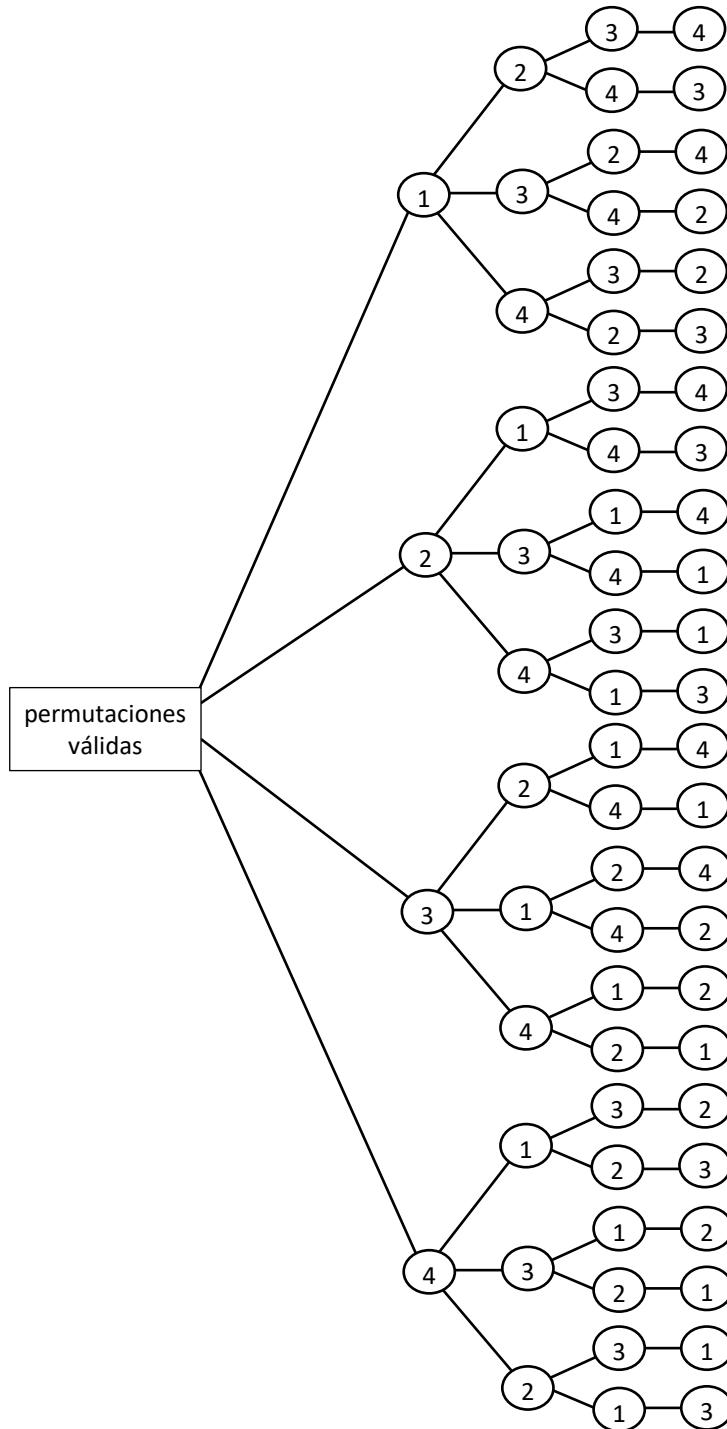
$$\begin{aligned} p(n, n) &= n * (n-1) * (n-2) * \dots * 3 * 2 * 1 \\ p(n, n) &= n! \end{aligned}$$

En general, la idea de *permutación* implica que *el orden en que se presentan los elementos es relevante*. Así, cada una de las 24 formas de combinar 4 dígitos diferentes es una permutación diferente de las otras (aunque todas tienen los mismos números) ya que en todas los números aparecen en distinto orden.

En muchas situaciones el problema se restringe un poco: se tienen efectivamente  $n$  elementos en un conjunto, y se quiere determinar la cantidad de permutaciones diferentes que pueden obtenerse [pero tomando los elementos en grupos de  \$m\$  \(con  \$m < n\$ \), y sin permitir repeticiones](#). Como se trata de permutaciones, el *orden es relevante*: se toman

como diferentes dos variaciones que tengan los mismos elementos pero en distinto orden. Usaremos la notación  $p(n, m)$  (con  $m \leq n$ ) para referirnos a permutaciones en general.

**Figura 2: Diagrama tipo árbol – Permutaciones de los números 1, 2, 3 y 4 tomados de a 4.**



Por ejemplo, suponga que en una universidad se debe seleccionar a los tres mejores estudiantes para distinguirlos como el abanderado y los dos escoltas, y las autoridades deben decidir entre siete estudiantes que fueron propuestos por distintos méritos. Se quiere saber de cuántas formas posibles podría quedar formado el trío de alumnos distinguidos. En este caso, el número de alumnos del conjunto es  $n = 7$ , y se desea agruparlos de a  $m = 3$ , pero de forma que, otra vez, será importante el orden en que queden. Así una terna

formada por "Juan", "Luis" y "Pedro" será tomada como diferente de otra como "Pedro", "Luis" y "Juan", ya que en ambas las distinciones corresponderán a alumnos diferentes.

Se trata de calcular  $p(n, m) = p(7, 3)$  = cantidad de permutaciones existentes, tomando 7 elementos en grupos de 3 distintos, sin repetirlos, y asumiendo como diferentes a aquellas en que los mismos elementos aparezcan en distinto orden. Podemos ver que cada terna posible de jugadores tendrá 7 posibilidades diferentes en cuanto al primer lugar (el abanderado), y el segundo lugar (primer escolta) será cubierto por alguno de los otros 6, quedando hasta aquí un total de  $7 * 6$  permutaciones. El tercer lugar (segundo escolta) sólo puede ser cubierto por alguno de los otros 5, quedando un total de  $7 * 6 * 5$ . Pero como sólo se pide permutarlos de tres en tres, el proceso finaliza allí... y el total será de 210 posibles ternas [4].

Está claro que en el cálculo de  $p(7, 3)$  no puede haber más de  $m = 3$  factores: la cantidad de posibilidades para el primer lugar ( $n = 7$ ) multiplicado por  $n-1 (= 6)$  y luego por  $n-2 (= 5)$ :

$$p(7, 3) = n * (n-1) * (n-2) = 7 * 6 * 5 = 210$$

con lo que la multiplicación debe detenerse al llegar al 5. Puede verse entonces que el último factor es de la forma:  $n - m + 1 = 7 - 3 + 1 = 5$ . Por lo tanto, la cantidad de permutaciones  $p(n, m)$  (con  $m < n$ ) es:

$$p(n, m) = n * (n-1) * (n-2) * \dots * (n-m+1)$$

y puede probarse por inducción matemática que lo anterior equivale a su vez a [5]:

$$p(n, m) = \frac{n!}{(n-m)!}$$

Es sencillo comprobar que en el caso particular en que  $m = n$ , la fórmula anterior lleva a:

$$p(n, m) = p(n, n) = n! / (n - n)! = n! / 0! = n!$$

lo cual ya sabíamos y era esperable: si  $m = n$ , entonces se está pidiendo lo que ya habíamos calculado: la cantidad de permutaciones tomando  $n$  elementos en grupos de  $n$ .

Hasta aquí hemos hablado de permutaciones *sin repetición* de elementos (también llamadas *permutaciones sin reposición*: una vez que se usa un elemento, no se repone al conjunto para volver a usarlo). **Pero en algunos casos se requiere saber cuántas permutaciones podrían formarse permitiendo que los elementos del conjunto se repitan**. Es el típico caso (por ejemplo) de determinar la cantidad de variantes que pueden formarse con los dígitos de una *cerradura de combinación* (que en realidad, debería entonces llamarse *cerradura de permutación*) de un candado o de una caja de seguridad.

Suponga que un candado tiene un mecanismo de seguridad de  $m = 4$  posiciones, y se quiere determinar cuántas permutaciones posibles existen para ese mecanismo. Cada posición se ocupa con un número entre 0 y 9 (o sea, el conjunto desde el cual se seleccionan los números tiene  $n = 10$  elementos) y deben permutarse tomados de a  $m = 4$ . La primera posición de la cerradura puede ocuparse con cualquiera de los  $n$  dígitos. Y como todos los números siguen siendo válidos para ocupar la segunda posición, entonces esta puede volver a ocuparse con cualquiera de los  $n$  dígitos teniendo hasta aquí,  $n * n = 10 * 10$  variantes. Y como los  $n$  dígitos vuelven a ser válidos para la tercera y la cuarta posición, resulta que el

total de permutaciones es  $n * n * n * n = 10 * 10 * 10 * 10 = 10^4 = 10000$  permutaciones posibles.

Puede verse entonces que si el conjunto de partida tiene  $n$  elementos, y se quiere permutarlos tomados de a  $m$ , pero permitiendo repeticiones (esto es, *permutaciones con reposición*) entonces el total  $pr(n, m)$  surge de la sencilla fórmula siguiente:

$$pr(n, m) = n^m \quad (m \leq n)$$

Las permutaciones tienen en cuenta el orden final de cada variación de elementos. Pero en muchos casos, *ese orden no es relevante y dos variaciones de los mismos elementos se deben tomar como iguales a los efectos del conteo*. Si ese es el caso, se habla de *combinaciones* en lugar de *permutaciones*.

Por ejemplo, suponga que en un evento deportivo para reunir fondos para una obra caritativa se tienen  $n = 6$  jugadores profesionales y se desea que en cada uno de los dos equipos que participarán del evento haya siempre  $m = 3$  jugadores profesionales. ¿En cuántas formas diferentes estos jugadores podrían repartirse en estas condiciones?

Puede verse que ahora no es importante el orden: un agrupamiento de tres jugadores como "Diego", "Lionel" y "Gabriel" será exactamente el mismo que otro de la forma "Gabriel", "Diego" y "Lionel", y no deben contarse como dos diferentes. Tampoco es válido en este caso contemplar repeticiones: no es posible que un mismo jugador aparezca más de una vez en un grupo. Por lo tanto, hablamos de cuántas *combinaciones sin repetición (o sin reposición)* pueden formarse tomando de a 3 a los 6 jugadores.

La forma general de calcular ese número puede pensarse así: se calculan las permutaciones de 6 tomados de a 3, como si el orden fuese relevante. En este caso, sería  $6*5*4 = 120$ . Pero luego debemos pensar que en esas 120 permutaciones hay muchas que son variantes ordenadas de los mismos nombres, y deben ser eliminadas del cálculo. ¿Cuántas son las que sobran? Es simple: Con  $m = 3$  nombres se pueden formar  $3*2*1 = 3! = m! = 6$  permutaciones ordenadas, que en realidad son iguales entre sí si se las toma como combinaciones. Por lo tanto, para eliminar del cálculo general a todas las permutaciones repetidas, se divide por 6 al total de 120 anterior, quedando 20 formas de agrupar a estos jugadores.

Entonces, la fórmula general para calcular las *combinaciones* de  $n$  elementos tomados de a  $m$  (con  $m \leq n$ ) (denotada aquí como  $c(n, m)$ ) es [5]:

$$c(n, m) = \frac{n!}{m! * (n - m)!}$$

Otra vez, note que si  $n = m$  entonces la fórmula anterior se reduce a  $n! / n! = 1$  lo cual de nuevo era esperable: hay una sola forma de combinar  $n$  elementos en grupos de a  $n$  sin que sea relevante el orden: tomarlos a todos juntos.

Finalmente, nos preguntamos qué pasaría si teniendo un conjunto de  $n$  elementos, quisiéramos saber el total de combinaciones tomados de a  $m$ , pero admitiendo repeticiones (o sea, *combinaciones de n elementos en grupos de m, con reposición*, que denotaremos como  $cr(n, m)$ ).

Por ejemplo, suponga que se tienen  $n = 5$  recipientes, cada uno conteniendo un tipo diferente de fruta (por caso, un recipiente de naranjas, otro de manzanas, y los otros durazno, peras y ciruelas) Suponga que se nos indica que podemos tomar hasta  $m = 3$  frutas,

seleccionándolas como queramos. Está claro que una selección de la forma (naranja, manzana, durazno) será igual a otra de la forma (manzana, durazno, naranja) y por lo tanto, el orden en que se combinen no es relevante. Y como ahora sería válido elegir dos o tres futas del mismo recipiente, entonces una combinación de la forma (naranja, naranja, manzana) también sería válida. Si se pide determinar cuántas variantes de combinación existen en estas condiciones, se tiene un caso de *combinaciones de  $n$  elementos tomados en grupos de  $a$   $m$ , con reposición* ( $cr(n, m)$ ).

Para deducir la fórmula a utilizar, supongamos que cada vez que se toma una fruta de un recipiente esa operación se marca como "*Tomar*" (y simbolizamos como  $T$ ), y cada vez que se salta a un recipiente desde otro (sea que tome una fruta o no del anterior) marcamos ese hecho como "*Saltar*" (simbolizado como  $S$ ). Entonces, las siguientes combinaciones podrían representarse como sigue (suponiendo que los recipientes están en orden indicado más arriba):

- ✓ (naranja, naranja, pera)  $\Rightarrow (T\ T) S\ S\ S (T) S$

Al comenzar, estamos en el recipiente de naranjas y tomamos 2 ( $T\ T$ ). Luego, la primera  $S$  nos saca del recipiente de naranjas y nos lleva al de manzanas. Como no queremos manzanas, no tomamos ninguna y la segunda  $S$  nos pone en el recipiente de duraznos. Tampoco queremos duraznos, y la tercera  $S$  nos deja en el de peras, donde tomamos una ( $T$ ). Finalmente, (para salir del puesto de venta!), saltamos otra vez y la última  $S$  nos lleva al recipiente de ciruelas (y ya no volvemos saltar... la  $S$  significa "*salte desde un recipiente a otro*")

- ✓ (manzana, pera, ciruela)  $\Rightarrow S (T) S\ S (T) S (T)$

Al comenzar estamos en el de naranjas y sin tomar ninguna saltamos (la primera  $S$ ) al de manzanas. Allí tomamos una (la primera ( $T$ )) y pasamos al de duraznos (la segunda  $S$ ). No tomamos ninguno y pasamos al de peras (la tercera  $S$ ) donde tomamos una (la segunda ( $T$ )). De allí seguimos al de ciruelas (la cuarta  $S$ ) y tomamos una (la última ( $T$ )).

Con este sistema de codificación, podemos notar que para  $n = 5$  y  $m = 3$ , en el cálculo de combinaciones con reposición siempre tenemos  $m + (n - 1) = 3 + (5 - 1) = 7$  posiciones *en total* (marcadas como  $S$  o como  $T$ ) y de esas 7, siempre hay  $m = 3$  posiciones marcadas como  $T$ . Por lo tanto, el problema puede plantearse como *calcular la cantidad de formas en que pueden combinarse  $m + (n - 1)$  tomados de  $a$   $m$  (cantidad de posiciones marcadas como  $T$ ), sin importar si hay reposición o no.*

Es decir, podemos calcular la cantidad  $cr(n, m)$  ( $n$  elementos combinados de a  $m$ , con reposición) como si se tratase de  $c(m+(n-1), m)$  (o sea:  $m+(n-1)$  elementos, combinados de a  $m$ , sin reposición) y la fórmula sería:

$$cr(n, m) = c(m + (n - 1), m) = \frac{(m + (n - 1))!}{m! * (n - 1)!}$$

Y en el ejemplo de  $n = 5$  recipientes y  $m = 3$  frutas en total, el cálculo quedaría:

$$cr(5, 3) = \frac{(3 + (5 - 1))!}{3! * (5 - 1)!} = \frac{7!}{3! * 4!} = \frac{5040}{6 * 24} = 35$$

Tenemos 35 formas de seleccionar 3 frutas de 5 recipientes diferentes, con opción a repetir la fruta que queramos.

Luego de este análisis completo, podemos poner todo en un tabla a modo de resumen de fórmulas y situaciones:

**Figura 3: Tabla de fórmulas y situaciones generales del cálculo combinatorio.**

Tipo	¿Importa el orden?	¿Admite repeticiones?	Situación	Fórmula
1. Permutación	sí	no	$n$ elementos permutados de a $n$ , sin reposición.	$p(n, n) = n!$
2. Permutación	sí	no	$n$ elementos permutados de a $m$ ( $m \leq n$ ), sin reposición.	$p(n, m) = \frac{n!}{(n - m)!}$
3. Permutación	sí	sí	$n$ elementos permutados de a $m$ (con $m \leq n$ ), con reposición.	$pr(n, m) = n^m$
4. Combinación	no	no	$n$ elementos combinados de a $m$ (con $m \leq n$ ), sin reposición.	$c(n, m) = \frac{n!}{m! (n - m)!}$
5. Combinación	no	sí	$n$ elementos combinados de a $m$ (con $m \leq n$ ), con reposición.	$cr(n, m) = \frac{(m + (n - 1))!}{m! (n - 1)!}$
6. Conteo de resultados	Total de resultados posibles de combinar un evento con $e_1$ variantes, con otro de $e_2$ variantes, y otro con $e_3$ , y así hasta uno al final con $e_k$ variantes.			$t = e_1 * e_2 * e_3 * \dots * e_k$

Como conclusión, podemos ver que el uso de factoriales resulta ser mucho más natural de lo que se habría esperado, ya que se requiere como operación fundamental en muchos problemas de conteo. Y podemos trabajarla en la resolución de un problema de aplicación, para cerrar esta Ficha:

**Problema 33.) Desarrollar un programa controlado por menú de opciones, que incluya opciones para realizar las siguientes tareas:**

1. *Cargar por teclado la cantidad  $n$  de colores que pueden tener los automóviles a la venta en una concesionaria, además de la cantidad  $m$  de modelos que existen de una marca en particular, y finalmente la cantidad  $t$  de opciones que se ofrecen a los clientes en cuanto a tipos de neumáticos a aplicar en cada vehículo. Calcule la cantidad total de combinaciones que tiene un cliente cuando deba elegir un automóvil de esa marca.*
2. *Para el diseño de un juego de tablero basado en combinar  $n$  sílabas, se desea saber cuántas formas diferentes existen de combinar esas  $n$  sílabas, sin repetirlas. Cargar  $n$  por teclado y mostrar esa cantidad.*
3. *Cargar por teclado la cantidad  $n$  de agentes de seguridad disponibles en una compañía de servicios de vigilancia. La compañía desea que en todo momento, sus agentes se distribuyan en grupos de  $m$  agentes. Calcule la cantidad de grupos diferentes que pueden formarse.*
4. *Cargar por teclado la cantidad  $n$  de corredores que participan de una carrera de Fórmula 1. Cargue también por teclado la cantidad de  $m$ , con  $m < n$ , de posiciones finales que tendrán puntos por llegar en esas posiciones. Calcular y mostrar la cantidad de posibles formas en que pueden cubrirse las  $m$  posiciones puntuables, con los  $n$  corredores.*

5. *Cargar por teclado la cantidad m de letras que conforman la clave de identificación asignada a los empleados de una empresa. Sabiendo que las posibles letras diferentes del alfabeto español son n = 27, calcular la cantidad de formas que podría tener una clave.*
6. *Cargar por teclado la cantidad n de colores que pueden tener las fichas contenidas en una bolsa, y la cantidad m de fichas que se deben extraer de la bolsa para un ejercicio de probabilidades. Calcular la cantidad de variantes que puede tener cada conjunto posible de m fichas en cuanto a sus posibles colores.*

**Discusión y solución:** Se trata de un programa para aplicar en forma general las fórmulas presentadas en esta sección.

Para el punto 1 se aplica en forma directa el cálculo del *total de resultados*  $r = n * m * t$  que se explicó más arriba (fórmula 6 de la tabla de la Figura 3). La función que sigue hace el trabajo:

```
def total_resultados(n, m, t):
    return n * m * t
```

En el punto 2 se necesita saber la cantidad de *permutaciones* posibles que pueden formarse con  $n$  sílabas diferentes tomadas de a  $n$  (esto es, sin repetir sílabas y considerando como diferentes a dos variantes con las mismas sílabas pero en otro orden). Sabemos que en este caso el total de permutaciones es  $p(n, n) = n!$  (fórmula 1 de la Figura 3) y eso es lo que aplica la siguiente función *permutaciones\_sin\_reposición()*, asumiendo que la función *factorial()* está definida y disponible:

```
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

def permutaciones_sin_reposición(n, m):
    fn = factorial(n)
    fd = factorial(n - m)
    return fn // fd
```

Note que la función *permutaciones\_sin\_reposición()* toma dos parámetros  $n$  y  $m$  que pueden ser diferentes o también pueden ser iguales. Si  $n = m$ , la función calculará el valor  $n!$  y lo retornará. Y si fuese  $n > m$ , la función calculará las permutaciones que correspondan y también retornará ese valor. La misma función puede usarse para casos en que  $n = m$  o también en casos en que  $n > m$  (es decir, aplica tanto para la fórmula 1 como para la fórmula 2 de la Figura 3)

En el punto 3, se pide la cantidad de *combinaciones* en que pueden agruparse  $n$  agentes, tomados de a  $m$ : ahora los grupos con exactamente los mismos agentes son iguales, ya que el orden en que llegan al grupo no es relevante. Y las combinaciones no pueden incluir más de una vez al mismo agente, por lo que se trata de combinaciones sin reposición. La fórmula a aplicar es entonces la fórmula 4 de la Figura 3. La siguiente función hace el cálculo pedido:

```
def combinaciones_sin_reposición(n, m):
    fn = factorial(n)
    fm = factorial(m)
    fr = factorial(n - m)
    return fn // (fm * fr)
```

El punto 4 es una situación de cálculo de *permutaciones de  $n$  tomados de a  $m$ , sin reposición*: hay  $n$  pilotos de carreras y  $m$  puestos que otorgan puntaje. Como no es lo mismo llegar primero que segundo o tercero (el puntaje obtenido será diferente), entonces el orden es importante y eso nos define que se trata de permutaciones. Como además, ningún piloto puede llegar en más de una posición (o es primero o es segundo, pero no puede ser primero y segundo al mismo tiempo), entonces sabemos que las *permutaciones son sin repetición* (o *sin reposición*). Por lo tanto, aplica la *fórmula 2* de la *Figura 3*, que ya teníamos programada en la función *permutaciones\_sin\_reposición()*:

```
def permutaciones_sin_reposición(n, m):
    fn = factorial(n)
    fd = factorial(n - m)
    return fn // fd
```

El requerimiento 5 del problema es calcular cuántas formas distintas podría tener una clave formada por  $n$  letras, sabiendo que cada letra puede ser una de 27 posibles. Está claro que dos claves con las mismas letras pero en orden diferente, valen como distintas, y por lo tanto tenemos un problema de *permutaciones: el orden importa*. El conjunto de letras está formado por 27 letras posibles ( $n = 27$ ), y cada clave debe formarse combinando  $m$  letras, por lo que tenemos que permutar  $n$  elementos tomados de a  $m$ . Además, cada letra podría repetirse en una clave, por lo que finalmente tenemos un problema de permutaciones de  $n$  tomados de a  $m$ , con reposición, y aplica la *fórmula 3* de la *Figura 3*. La siguiente función calcula ese valor:

```
def permutaciones_con_reposición(n, m):
    return pow(n, m)
```

Finalmente, el punto 6 supone fichas de  $n$  colores guardadas en una bolsa, y se pide calcular en cuántas formas podrían salir combinados esos colores tomando aleatoriamente  $m$  fichas de la bolsa en un momento dado. Como dos arreglos de fichas de los mismos colores pero en diferente orden serían iguales, entonces estamos ante un problema de cálculo de *combinaciones (el orden no es relevante)*. Y como podrían aparecer fichas del mismo color cada vez que se extraigan  $m$  fichas, entonces las combinaciones *admiten reposición*. Se trata de combinaciones de  $n$  colores, tomados de a  $m$  por vez, *con reposición*. Aplica la *fórmula 5* de la *Figura 3*, y la función que sigue hace ese trabajo:

```
def combinaciones_con_reposición(n, m):
    fn = factorial(m + (n - 1))
    fm = factorial(m)
    fr = factorial(n - 1)
    return fn // (fm * fr)
```

El programa completo se muestra a continuación:

```
__author__ = 'Catedra de AED'

# funciones "grupo 2": genéricas... resuelven problemas específicos...
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
```

```

def permutaciones_con_reposición(n, m):
    return pow(n, m)

def permutaciones_sin_reposición(n, m):
    fn = factorial(n)
    fd = factorial(n - m)
    return fn // fd

def combinaciones_con_reposición(n, m):
    fn = factorial(m + (n - 1))
    fm = factorial(m)
    fr = factorial(n - 1)
    return fn // (fm * fr)

def combinaciones_sin_reposición(n, m):
    fn = factorial(n)
    fm = factorial(m)
    fr = factorial(n - m)
    return fn // (fm * fr)

def total_resultados(n, m, t):
    return n * m * t

# funciones "grupo 1": gestionan la interfaz de usuario...
def validar_mayor_que(inf):
    n = inf - 1
    while n < inf:
        n = int(input('Valor (mayor que ' + str(inf) + ' por favor...): '))
        if n < inf:
            print('Error... se pidió >' + str(inf), '... cargue de nuevo...')
    return n

def validar_menor_que(sup):
    n = sup + 1
    while n > sup:
        n = int(input('Valor (menor que ' + str(sup) + ' por favor...): '))
        if n > sup:
            print('Error... se pidió <' + str(sup), '... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor (entre ' + str(inf) + ' y ' + str(sup)+ ': ''))
        if n < inf or n > sup:
            print('Se pidió entre', inf, 'y', sup, '... cargue de nuevo...')
    return n

def opción1():
    print('Cantidad de colores...')
    n = validar_mayor_que(0)

    print('Cantidad de modelos...')
    m = validar_mayor_que(0)

    print('Cantidad de tipos de neumáticos...')
    t = validar_mayor_que(0)

    res = total_resultados(n, m, t)
    print('Cantidad total de combinaciones:', res)

```

```

def opcion2():
    print('Cantidad de sílabas diferentes...')
    n = validar_mayor_que(0)
    tp = permutaciones_sin_reposición(n, n)
    print('Permutaciones de', n, 'sílabas tomadas de a', n, ':', tp)

def opcion3():
    print('Cantidad de agentes...')
    n = validar_mayor_que(0)

    print('Cantidad de agentes por grupo...')
    m = validar_intervalo(0, n)

    tp = combinaciones_sin_reposición(n, m)
    print('Combinaciones de', n, 'agentes tomados en grupos de a', m, ':', tp)

def opcion4():
    print('Cantidad de pilotos...')
    n = validar_mayor_que(0)

    print('Cantidad de puestos con puntaje...')
    m = validar_intervalo(0, n)

    tp = permutaciones_sin_reposición(n, m)
    print('Permutaciones de', n, 'pilotos tomados de a', m, ':', tp)

def opcion5():
    n = 27
    print('La cantidad de letras a combinar es', n)

    print('Cantidad de posiciones en cada clave...')
    k = validar_mayor_que(0)

    tp = permutaciones_con_reposición(n, k)
    print('Permutaciones de', n, 'letras tomadas de a', k, 'con repeticiones:', tp)

def opcion6():
    print('Cantidad de colores...')
    n = validar_mayor_que(0)

    print('Cantidad de fichas a extraer...')
    m = validar_mayor_que(0)

    tp = combinaciones_con_reposición(n, m)
    print('Combinaciones de', n, 'colores tomados de a', m, 'con reposicion:', tp)

def menu():
    # título general...
    print('Menú de opciones - Cálculo combinatorio')

    op = 1
    while op != 7:
        # visualización de las opciones...
        print('1. Total de ofertas de vehículos')
        print('2. Permutaciones de sílabas')
        print('3. Combinaciones de agentes de seguridad')
        print('4. Permutaciones de pilotos de carreras')
        print('5. Permutaciones de letras en una clave')
        print('6. Combinaciones de colores')

        print('7. Salir')

```

```

op = int(input('Ingrese el numero de la opcion elegida: '))

# chequeo de la opcion elegida...
if op == 1:
    opcion1()
elif op == 2:
    opcion2()
elif op == 3:
    opcion3()
elif op == 4:
    opcion4()
elif op == 5:
    opcion5()
elif op == 6:
    opcion6()

# script principal...
menu()

```

El programa incluye tres funciones de carga con validación, que seguramente resultarán útiles al estudiante:

- **validar\_mayor\_que(*inf*):** carga y retorna un número entero, validando que ese número sea mayor al valor *inf* tomado como parámetro.
- **validar\_menor\_que(*sup*):** carga y retorna un número entero, validando que ese número sea menor al valor *sup* tomado como parámetro.
- **validar\_intervalo(*inf, sup*):** carga y retorna un número entero, validando que ese número sea mayor al valor *inf* tomado como parámetro y al mismo tiempo menor que el valor *sup* también tomado como parámetro.

En este caso la segunda no está siendo utilizada en el programa, pero se deja disponible de todos modos por su utilidad práctica.

Las funciones *opcion1()*, *opcion2()* y *opcion3()* son similares a las que ya hemos presentado en otros programas con menú de opciones, y son las encargadas de manejar la interfaz de usuario antes y después de invocar a las funciones genéricas *total\_resultados()*, *combinaciones()* y *permutaciones()*. Dejamos su análisis para el alumno.

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [4] S. Lipschutz, Matemáticas para Computación, México: McGraw-Hill / Interamericana, 1993.
- [5] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.

# Ficha 11

## Módulos y Paquetes

### 1.] Tratamiento especial de parámetros en Python.

En la *Ficha 9* hemos analizado la forma utilizar parámetros en una función en Python, y hemos visto luego que la parametrización es uno de los mecanismos que favorecen el desarrollo de funciones genéricas y reutilizables. Pero los conceptos y mecanismos que vimos en la Ficha 9 no agotan el tema: una función en Python puede plantearse de forma de aceptar y manejar un *número variable de parámetros*, en forma similar a lo que permiten hacer otros lenguajes como C y C++, o a lo que ya hace Python con algunas de sus funciones predefinidas como *max()* y *min()*:

```
m1 = max(3, 5, 6)
m2 = max(2, 5, 6, 7, 2, 1)

m3 = min(5, 6, 3, 8)
m4 = min(4, 9, 2, 8, 1, 7, 0)
```

Estas dos funciones de Python pueden ser invocadas enviándoles diferentes cantidades de parámetros actuales cada vez, y ambas procesarán esos parámetros y retornarán el valor mayor o el valor menor, respectivamente.

Puede verse que de forma, una función resulta aun más genérica y reusable, ya que puede ser aplicada en muchos contextos que requieran ese servicio, sin importar la cantidad de parámetros que deban pasarse a modo de datos. No es necesario contar con funciones diferentes que hagan la misma tarea sobre un número distinto de parámetros: *la misma y única función acepta la cantidad de parámetros que se requiera procesar, y los procesa*.

Obviamente, el programador puede desarrollar sus propias funciones con número variable de parámetros. Existen básicamente *tres mecanismos* para lograr esto en Python, y esos tres mecanismos pueden a su vez combinarse. Veamos brevemente cada uno de ellos [\[1\]](#):

a.) **Parámetros con valores por defecto:** Es común y muy útil asignar a los *parámetros formales* de una función un *valor por defecto*, de forma de poder invocar a la función enviando menos parámetros actuales. Se dice que un *parámetro formal tiene un valor por defecto*, cuando el mismo es asignado en forma explícita con un valor en el momento en que se define en la cabecera de la función. Veamos un ejemplo:

Supongamos que se quiere definir una función que tome como parámetros a dos números *n1* y *n2*, y se desea que la función retorne una tupla con esos mismos números, pero ordenados. Típicamente, el ordenamiento esperado es de menor a mayor pero en ocasiones podría requerirse que sea de mayor a menor.

En lugar de definir dos funciones (una para cada tipo de ordenamiento) se puede desarrollar sólo una, que tome tres parámetros: los dos números a ordenar y un flag o bandera de tipo

booleano para indicar si queremos un ordenamiento ascendente o descendente, como se ve en el siguiente modelo: En la función *ordenar()* los parámetros formales *n1* y *n2* son los números que se deben comparar. Estos dos parámetros no tienen valor por defecto, **por lo que al invocar a la función deben obligatoriamente ser enviados los parámetros actuales correspondientes.** El tercer parámetro (*ascendent*) es un valor booleano que indica si la función debe retornar los valores en orden ascendente (*ascendent = True*) o descendente (*ascendent = False*). **El detalle es que este tercer parámetro tiene por defecto el valor True,** por lo cual si el programador desea resultados en orden ascendente, **puede invocar a la función enviando sólo los dos números a comparar y la función asumirá que el valor del tercer parámetro es True:**

```
__author__ = 'Catedra de AED'

def ordenar(n1, n2, ascendent=True):
    # se asume ascendent = True...
    first, second = n2, n1
    if n1 < n2:
        first, second = n1, n2

    # ... pero si ascendent = False, invertir los valores...
    if not ascendent:
        first, second = second, first

    return first, second

def test():
    a = int(input('Ingrese el primer valor: '))
    b = int(input('Ingrese el segundo valor: '))

    # orden ascendente...
    men, may = ordenar(a, b)

    print('Menor:', men)
    print('Mayor:', may)

    c = int(input('Ingrese el primer valor: '))
    d = int(input('Ingrese el segundo valor: '))

    # orden descendente...
    may, men = ordenar(c, d, False)

    print('Menor:', men)
    print('Mayor:', may)

# script principal...
test()
```

Como se ve, **si se quiere que el ordenamiento sea descendente, la función *ordenar()* debe invocarse con tres parámetros actuales**, como el caso de *ordenar(c, d, False)*. El tercer parámetro actual puede obviarse al invocar a la función (como en *ordenar(a, b)*) y su valor se asumirá *True*, pero también puede enviarse si el programador así lo desea: una invocación de la forma *ordenar(a, b)* es equivalente a otra de la forma *ordenar(a, b, True)*: en ambas, el tercer parámetro formal se tomará como igual a *True*, y la función ordenará de menor a mayor.

Los parámetros formales que no tienen valor por defecto en la función, se designan como *parámetros posicionales*; y note lo que ya hemos indicado: si una función tiene *parámetros posicionales*, entonces al invocar a la función **es obligatorio** enviar los parámetros actuales que correspondan a esos posicionales, *pues de lo contrario se lanzará un error de intérprete*. Además, al declarar una función, los *parámetros posicionales* deben definirse siempre **antes** que los parámetros con valor default. La siguiente cabecera para la función *ordenar()* es incorrecta, ya que el parámetro formal posicional *n2* se está definiendo después de un parámetro con valor default:

```
# incorrecto...
def ordenar(n1, ascendent=True, n2):
```

En nuestro ejemplo original, la función *ordenar()* original tiene *dos parámetros formales posicionales* (*n1* y *n2*) y un tercero con *valor por defecto* (*ascendent*). Por lo tanto, al invocarla es siempre obligatorio enviarle al menos dos valores actuales para los posicionales *n1* y *n2*:

```
may, men = ordenar(a, b, False)      # correcto...
men, may = ordenar(c, d)              # correcto...
men, may = ordenar(c, d, True)        # correcto...

may, men = ordenar()                  # incorrecto...
men, may = ordenar(a)                # incorrecto...

men, may = ordenar(a, False)          # atención aquí...
```

Los tres ejemplos *resaltados en color verde en el modelo anterior son correctos. Los dos resaltados en rojo producirán un error de intérprete*: en el primero de ellos, los dos parámetros posicionales *n1* y *n2* se quedan sin valor actual, y en el segundo, el primer parámetro *n1* se asigna con el valor actual de *a*, pero el segundo se queda sin asignar.

El *ejemplo resaltado en color violeta* no produce un error de intérprete, ya que efectivamente se han enviado dos parámetros actuales al invocar a la función... Sin embargo, hay un problema: los parámetros actuales que se envían son asignados a los parámetros formales posicionales en orden de aparición, de acuerdo a su posición en la cabecera de la función (de allí el nombre de *posicionales*) por lo que el valor de *a* será asignado en *n1* y el valor *False* será asignado en *n2*. En la función, el parámetro *ascendent* tomará entonces su valor por defecto (*True*), y la función procesará datos incorrectos (comparará un *int* con un *boolean* y asumirá un ordenamiento ascendente cuando el programador lo quería descendente)...

**b.) Parámetros con palabra clave:** Si una función tiene parámetros asignados con valores por defecto, podría haber problemas de ambigüedad para invocarla correctamente, ya que los valores de los parámetros actuales se toman en orden de aparición para ser asignados en los formales [1] [2]. Considere el siguiente ejemplo simple:

```
__author__ = 'Catedra de AED'

def datos(nombre, pais='Argentina', sexo='Varon', trabaja=True, estado='Soltero'):
    print('Datos recibidos: ')
    print('Nombre:', nombre)
    print('Pais:', pais)
    print('Sexo:', sexo)
```

```

print('Tiene trabajo?:', trabaja)
print('Estado civil:', estado)

def test():

    # error (interprete): falta el parametro 1 (obligatorio)...
    # datos()

    # Ok...
    datos('Pedro')

    # Ok...
    datos('Luisa', 'Uruguay', 'Mujer')

    # error (ambiguedad): toma pais = "Mujer", sexo = "Casada"...
    # datos('Maria', 'Mujer', 'Casada')

test()

```

La función `datos()` del modelo, tiene cinco parámetros: el primero (`nombre`) es posicional y los otros cuatro (`pais`, `sexo`, `trabaja`, `estado`) tienen valores por defecto (o por `default`). En forma correcta, el parámetro posicional `nombre` está definido *antes* que los cuatro parámetros con valor `default`. Como la función tiene al menos un parámetro posicional, entonces obligatoriamente debe enviarse al menos un parámetro actual al invocarla. *La siguiente invocación produciría un error de intérprete* (y por eso está *comentarizada* en el ejemplo anterior):

```

# error (interprete): falta el parametro 1 (obligatorio)...
# datos()

```

La siguiente invocación es *correcta*: el parámetro posicional `nombre` queda asignado con el valor 'Pedro', y los otros cuatro parámetros toman sus valores `default`:

```

# Ok...
datos('Pedro')

```

La salida en pantalla producida al invocar a la función en estas condiciones, sería la siguiente:

```

Datos recibidos:
Nombre: Pedro
Pais: Argentina
Sexo: Varon
Tiene trabajo?: True
Estado civil: Soltero

```

Como los cuatro últimos parámetros formales tienen valores `default`, podemos ignorar algunos de ellos al invocar a la función. La siguiente invocación también es correcta:

```

# Ok...
datos('Luisa', 'Uruguay', 'Mujer')

```

El parámetro `nombre` se asignará con la cadena 'Luisa', y los dos parámetros que siguen (`pais` y `sexo`) serán asignados con los valores 'Uruguay' y 'Mujer' respectivamente. Los dos últimos parámetros (`trabaja` y `estado`) quedarán con sus valores `default`. La salida sería la siguiente:

```

Datos recibidos:
Nombre: Luisa
Pais: Uruguay

```

```

Sexo: Mujer
Tiene trabajo?: True
Estado civil: Soltero

```

Pero finalmente, si se ejecuta el ejemplo tal como está, *la última invocación a la función datos() asignará valores incorrectamente en los parámetros formales*: si queremos dejar el parámetro *pais* en su valor *default*, no podemos saltarla y luego seguir enviando valores explícitos porque eso provocaría ambigüedad. La salida producida por la última invocación sería la siguiente:

```

Datos recibidos:
Nombre: Maria
Pais: Mujer
Sexo: Casada
Tiene trabajo?: True
Estado civil: Soltero

```

Conclusión: los parámetros con valores *default* deben declararse después que los parámetros posicionales, y además no pueden saltarse en forma directa cuando la función es invocada. Todos los parámetros actuales que se envíen a la función, serán tomados y asignados a los parámetros formales *en estricto orden de aparición* de izquierda a derecha, pudiendo provocar ambigüedades si se intenta saltar un parámetro.

Para evitar este tipo de problemas y poder *seleccionar* qué parámetros se deben dejar con su valor *default* y qué parámetros se desea asignar en forma explícita, en Python *se puede seleccionar cada parámetro por su nombre (usando el mecanismo de selección de parámetro por palabra clave)* cuando se invoca a la función:

```

__author__ = 'Catedra de AED'

def datos(nombre, pais='Argentina', sexo='Varon', trabaja=True, estado='Soltero'):
    print('Datos recibidos: ')
    print('Nombre:', nombre)
    print('Pais:', pais)
    print('Sexo:', sexo)
    print('Tiene trabajo?:', trabaja)
    print('Estado civil:', estado)

def test():
    # ok...
    datos('Luigi', pais='Italia')

    # ok... el parámetro "nombre" tambien puede accederse asi...
    datos(nombre='Luigi', pais='Italia')

    # ok.... uno sin palabra clave, otro con palabra clave, y el resto default...
    datos('Camila', 'Argentina', sexo='Mujer')

    # ok.... el orden de palabras clave no importa...
    datos('Bruno', sexo='Varon', pais='Italia')

    # error: luego de una palabra clave, no puede seguir explícito...
    # datos('Mary', pais='Inglaterra', 'Mujer')

    # error: no se puede asignar dos veces el mismo parametro...
    # datos('Federico', pais='Argentina', pais='Italia')

    # error: no puede usar un parametro que no existe...

```

```
# datos('Conrado', colegio='Lasalle')

test()
```

Como puede verse, la idea es en principio simple: al **invocar** a la función, se escribe el nombre del parámetro formal cuyo valor se quiere asignar en forma explícita y se asigna a ese parámetro el valor que se requiere. Sin embargo, hay algunas reglas que deben respetarse, y que se deducen del modelo anterior:

- Cualquier parámetro puede seleccionarse usando la notación de palabra clave, **incluso los parámetros posicionales** (que no tienen un valor default asociado):

```
# ok... el parámetro "nombre" tambien puede accederse así...
datos(nombre='Luigi', pais='Italia')
```

- Una vez que se accedió a un parámetro por palabra clave, **los que siguen a él en la lista de parámetros formales deben** ser accedidos por palabra clave cuando se invoca a la función:

```
# error: luego de una palabra clave, no puede seguir explícito...
# datos('Mary', pais='Inglaterra', 'Mujer')
```

- No se puede asignar **más de un valor al mismo parámetro**:

```
# error: no se puede asignar dos veces el mismo parametro...
# datos('Federico', pais='Argentina', paiss='Italia')
```

- No se puede usar un **parámetro que no existe**:

```
# error: no puede usar un parametro que no existe...
# datos('Conrado', colegio='Lasalle')
```

- Mientras se respeten las reglas anteriores, **no hay problema en cambiar el orden de acceso a los parámetros usando sus palabras clave**:

```
# ok... el orden no importa... si se respeta todo lo demás...
datos('Bruno', sexo='Varon', pais='Italia')
```

Un detalle interesante, es que obviamente Python usa masivamente tanto el mecanismo de parámetros con valores default como este mecanismo de selección de parámetros por palabra clave<sup>1</sup>, en sus funciones predefinidas de la librería estándar. Una de las funciones en las que se puede ver claramente este hecho, es la conocidísima función *print()* que usamos para visualizar resultados y mensajes en la consola estándar [1].

Esta función (entre otros) dispone de dos parámetros formales con valores default que son típicamente seleccionados por palabra clave: *sep* y *end*. El primero se usa para indicar a la función qué carácter o cadena de caracteres debe usar para separar las cadenas que se quieren mostrar, y *su valor default es un espacio en blanco (' ')*. El segundo se usa para

---

<sup>1</sup> El uso de palabras clave para seleccionar un elemento o para restringir el acceso (o permitir el paso) a determinados lugares o aplicaciones es obviamente común en programación... pero también en muchos otros ámbitos reales o imaginarios. ¿Quién no recuerda la increíble trilogía de películas de *El Señor de los Anillos* (o *The Lord of the Rings*) dirigida por Peter Jackson y protagonizada por Ian McKellen y Elijah Wood, y concretamente la primera de la saga (*The Fellowship of the Ring* o *La Comunidad del Anillo* del año 2001) en la que Gandalf, Frodo y los demás deben abrir las *Puertas de Durin* de las *Minas de Moria*? Esas puertas sólo se abrían si se decía la frase o palabra correcta... que resultó ser la palabra "*mellon*" ("amigo" en el idioma de los elfos...) Las películas de Jackson están basadas en la aún más extraordinaria obra literaria en tres volúmenes de J. R. R. Tolkien: *The Lord of the Rings* de 1954.

indicar con qué carácter o cadena de caracteres debe terminar la visualización, y *su valor default es un salto de línea ('\n')*. Por ese motivo, en la siguiente secuencia:

```
x, y = 10, 20
print('Valor x:', x)
print('Valor y:', y)
```

la salida producida es de la forma:

```
Valor x: 10
Valor y: 20
```

Como el parámetro *sep* tiene un espacio en blanco como valor default, la función pone un espacio en blanco luego de las cadenas 'Valor x:' y 'Valor y:', haciendo que los números 10 y 20 aparezcan a un blanco de distancia de los dos puntos en cada caso. Y como el valor default del parámetro *end* es un salto de línea, la función muestra su salida y salta al renglón siguiente, provocando que ambas salidas aparezcan a en dos renglones separados.

Se pueden cambiar esos caracteres llamando a la función y seleccionando esos parámetros por su palabra clave:

```
x, y = 10, 20
print('Valor x:', x, sep='-->', end='\n\n')
print('Valor y:', y)
```

La salida producida por el script anterior, será de la forma:

```
Valor x:--> 10
```

```
Valor y: 20
```

Como puede verse, en la primera invocación a la función el parámetro *sep* fue asignado con la cadena '*-->*', lo que hace que la ejecutarse la función se agregue esa cadena después del título 'Valor x:'. Y como el parámetro *end* fue asignado con *dos saltos de línea* en lugar de uno ('\n\n'), entonces ambas líneas de salida aparecen a dos renglones de distancia.

En el ejemplo siguiente, el valor de *end* se reemplaza por un *espacio en blanco*, lo que hace que ambas invocaciones a la función muestren sus salidas en la misma línea de la pantalla:

```
x, y = 10, 20
print('Valor x:', x, end=' ')
print('Valor y:', y)
```

La salida producida es:

```
Valor x: 10 Valor y: 20
```

Note que si la función *print()* es invocada sin ningún parámetro su efecto será simplemente mostrar el valor de *end*, provocando sólo un salto de línea:

```
x, y = 10, 20
print('Valor x:', x)
print()
print('Valor y:', y)
```

La salida producida es de la forma:

```
Valor x: 10
```

Valor y: 20

ya que la primera invocación provoca un salto de línea (*end* tiene en ella su valor default '\n') y la segunda invocación a *print() sin parámetros* provoca un segundo salto.

c.) **Listas de parámetros de longitud arbitraria:** La tercer forma de tratamiento especial de parámetros en Python, consiste en permitir que una función acepte un *número arbitrario* de parámetros. Los parámetros enviados de acuerdo a esta modalidad entrarán a la función empaquetados en una *tupla*: esto es, entrarán a la función como una lista de valores separados por comas y accesibles por sus índices [1] [2].

La forma de indicar que una función tiene una lista variable de parámetros, consiste (en general) en *colocar un asterisco delante del nombre del último parámetro posicional que se prevea para la función*. Ese último parámetro, definido de esta forma, representa la secuencia o tupla de valores de longitud variable.

En el ejemplo siguiente, se muestra una función *procesar\_notas()* que toma dos parámetros posicionales y obligatorios: el *nombre* y la *nota* final de un alumno en un curso. Pero además la función define una *lista de parámetros de longitud variable mediante el tercer parámetro args*. Se supone que los valores adicionales que la función recibirá, son las notas parciales que el alumno haya obtenido (si se decide enviarlas). La función simplemente muestra todos los datos en consola estándar. Observe una posible forma de procesar la secuencia de valores en la tupla *args*, usando un *for* que itera sobre esa tupla, y la forma de chequear si efectivamente hay parámetros adicionales preguntando si la longitud de la tupla es diferente de cero mediante la función *len()* de la librería estándar:

```
__author__ = 'Catedra de AED'

def procesar_notas(nombre, nota, *args):
    # procesamiento de los parámetros normales...
    print('Notas del alumno:', nombre)
    print('Nota Final:', nota)

    # procesar la lista adicional de parámetros, si los hay...
    if len(args) != 0:
        print('Otras notas ingresadas:')
        for d in args:
            print('\tNota Parcial:', d)

def test():
    # una invocacion "normal", sin parámetros adicionales...
    procesar_notas('Carlos', 9)
    print()

    # una invocacion con tres notas adicionales...
    procesar_notas('Juan', 8, 10, 6, 7)

# script principal.....
test()
```

La salida producida por el programa anterior, sería la siguiente:

```
Notas del alumno: Carlos
Nota Final: 9
```

```
Notas del alumno: Juan
```

```
Nota Final: 8
```

```
Otras notas ingresadas:
```

```
Nota Parcial: 10
```

```
Nota Parcial: 6
```

```
Nota Parcial: 7
```

También es interesante notar que Python aplica esta técnica en numerosas funciones de su librería estándar. Conocemos un par de esas funciones: *min()* y *max()*, que determinan y retornan el menor/mayor de una secuencia de valores de longitud arbitraria:

```
mn = min(2, 4, 5, 7, 2, 3)
my = max(2, 6, 3, 7)
```

En ambos casos, los parámetros son ingresados a la función como tuplas en la forma descripta en esta sección, y ambas funciones procesan luego esas tuplas.

## 2.] Definición y uso de módulos en Python.

Un *módulo* es una colección de definiciones (variables, funciones, etc.) contenida en un archivo separado con extensión *.py* que puede ser importado desde un script o desde otros módulos para favorecer el reuso de código y/o el mantenimiento de un gran sistema. Se trata de lo que en otros lenguajes llamaríamos una *librería externa*. Además, un conjunto de módulos en Python puede organizarse en carpetas llamadas *paquetes* (o *packages*) que favorezcan aún más el mantenimiento y la distribución (veremos el uso de *packages* posteriormente).

La idea de usar un *módulo*, es agrupar definiciones de uso común (funciones genéricas, por ejemplo) en un archivo separado, cuyo contenido pueda ser accedido cuando se requiera, sin tener que repetir el código fuente de cada función o declaración en cada programa que se desarrolle. Esto permite que un programador reutilice con más sencillez sus funciones ya desarrolladas, y acorte la longitud de sus programas (entre otras ventajas) [\[1\]](#).

Para crear un módulo sólo debe escribir en un archivo con extensión *.py* las definiciones de funciones y variables que vaya a necesitar. Por ejemplo, mostramos aquí un módulo guardado en el archivo *soporte.py*, con algunas de las funciones que hemos estado usando como ejemplo en esta Ficha o en otras anteriores (vea el proyecto [\[F11\] Ejemplo](#) que acompaña a esta Ficha como anexo, tanto para el código fuente del módulo *soporte.py* como para todos los programas que siguen en esta sección):

```
__author__ = 'Cátedra de AED'

# archivo soporte.py
# Este archivo es un "modulo" que contiene funciones varias...

# una funcion para retornar el menor entre dos numeros...
def menor(n1, n2):
    if n1 < n2:
        return n1
    return n2
```

```
# una función para calcular el factorial de un numero...
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

# una función para ordenar dos numeros...
def ordenar(n1, n2, ascendent=True):
    first, second = n2, n1
    if n1 < n2 :
        first, second = n1, n2
    if not ascendent :
        first, second = second, first
    return first, second
```

Para que un script o un programa pueda usar las funciones y demás declaraciones de un módulo externo, debe *importarlo* usando alguna de las variantes de la instrucción ***import*** [1]. Suponga que el siguiente programa está almacenado en un segundo archivo ***prueba01.py***, dentro de la misma carpeta donde está el módulo ***soporte.py***:

```
__author__ = 'Cátedra de AED'

# Archivo prueba01.py
# Un programa en Python, que incluye al módulo soporte.py

import soporte

def test():
    a = int(input('Cargue el primer número: '))
    b = int(input('Cargue el segundo número: '))

    # invocación a las funciones del módulo "soporte"
    m = soporte.menor(a,b)
    f = soporte.factorial(m)

    # si una función se va a usar mucho, se la puede referenciar con
    # un identificador local...
    men = soporte.menor
    c = men(3,5)

    print('El menor es:', m)
    print('El factorial del menor es:', f)
    print('El segundo menor es:', c)

# script principal...
test()
```

La instrucción ***import soporte*** que se muestra al inicio de este programa introduce el *nombre del módulo* en el contexto del programa, pero ***no*** el nombre de las funciones definidas en él. Por eso, para invocar a una de sus funciones debe usarse el operador *punto* (*.*) en la forma: ***nombre del módulo + punto + función a invocar***:

```
m = soporte.menor(a,b)
f = soporte.factorial(m)
```

Note también que es posible asociar una función a un identificador local (que en realidad es una referencia o un puntero a esa función), a modo de sinónimo que simplifique el uso:

```
men = soporte.menor
c = men(3,5)
```

En el siguiente programa (que suponemos almacenado en otro archivo *prueba02.py*), mostramos que se puede usar la variante *from - import* de modo que se incluya el nombre de la función específica que se quiere acceder desde un módulo, o de varias de ellas separadas por comas, evitando así tener que usar el *operador punto* cada vez que se quiera invocarla:

```
__author__ = 'Cátedra de AED'

# Archivo prueba02.py
# Un programa en Python, que incluye al módulo soporte.py

from soporte import factorial

def test():
    a = int(input('Cargue un número entero: '))
    f = factorial(a)
    print('El factorial del número es:', f)

# script principal...
test()
```

En el ejemplo anterior se incluyó la instrucción *from soporte import factorial*, la cual permite acceder directamente a la función *factorial()*, pero no al resto de las funciones del módulo. Si se quisiera acceder (por ejemplo) a las funciones *factorial()* y *menor()* pero no a otras, podría haberse usado la instrucción de esta otra forma:

```
from soporte import factorial, menor
```

En forma similar, se puede usar un *asterisco (\*)* en lugar del nombre de una función particular, para lograr así el acceso a *todas las funciones del módulo sin usar el operador punto*. El siguiente programa (que suponemos almacenado en tercer archivo *prueba03.py*) muestra la forma de hacerlo:

```
__author__ = 'Cátedra de AED'

# Archivo prueba03.py
from soporte import *

def test():
    a = int(input('Cargue un número entero: '))
    f = factorial(a)
    print('El factorial del número es:', f)
    r = menor(a,f)
    print('Menor:', r)

# script principal...
test()
```

Todo módulo en Python tiene definida una *tabla de símbolos* propia, en la que figuran los nombres de todas las variables y funciones que se definieron en el módulo. Esa tabla está disponible para el programa desde que el módulo se carga por primera vez en memoria con el primer import que pida acceder a ese módulo.

Existen además, en Python, algunas *variables especiales* que es importante comenzar a conocer, tal como la variable global `__name__` que todo módulo contiene en forma automática, y que siempre está asignada con el *nombre del módulo* en forma de cadena de caracteres [1] [2]. Note que el nombre de la variable (al igual que el de todas las *variables especiales* de Python) lleva *dos guiones de subrayado de cada lado del nombre*). Si el módulo fue correctamente cargado con una instrucción *import*, se puede acceder a la variable `__name__` con el consabido operador punto para consultar su valor, como se muestra en el siguiente programa (que suponemos guardado en el archivo *prueba04.py*)

```
__author__ = 'Cátedra de AED'

# Archivo prueba04.py
import soporte

def test():
    # invocacion a alguna funcion del modulo...
    print('Factorial de 4:', soporte.factorial(4))

    # mostrar el nombre del modulo...
    print('Nombre del modulo usado:', soporte.__name__)

test()
```

La salida del programa anterior será la siguiente:

```
Factorial de 4: 24
Nombre del modulo usado: soporte
```

Note que todo archivo fuente de Python lleva la extensión *.py* y que en definitiva, todo archivo con extensión *.py* constituye un *módulo*... Todos los programas que hemos mostrado conteniendo nuestros scripts, funciones de ejemplo y funciones de entrada (como *test()*) eran *módulos*... y sus contenidos podían/pueden ser usados desde otros *módulos* (entiéndase: otros programas) simplemente pidiendo su acceso mediante la instrucción *import*.

Sabemos también que *podemos ejecutar un programa directamente* (ya sea desde el shell de Python, desde un IDE o desde la línea de órdenes del sistema operativo). Es decir, podemos ejecutar un módulo directamente, sin accederlo con *import* desde otro módulo. Esto es lo que hemos hecho desde el inicio del curso: todos nuestros programas constituían *un único módulo*, que ejecutábamos cuando queríamos desde dentro del IDE *PyCharm*.

¿Qué pasa si se pide ejecutar un módulo que *no tiene un script principal*, y sólo contiene definiciones de funciones u otros elementos (como el caso de nuestro sencillo módulo *soporte.py*)? En un caso así, ninguna función de ese módulo será ejecutada, y en la consola de salida simplemente veremos un *mensaje general indicando que el proceso ha terminado*... sin hacer nada:

```
C:\Python34\python.exe "C:/Ejemplo/soporte.py"
Process finished with exit code 0
```

Técnicamente, es útil saber que si se pide directamente la *ejecución* de un módulo (en lugar de *importar* ese módulo para ser usado desde otro), entonces la variable global `__name__` del módulo que se pide ejecutar se asigna automáticamente con el valor "`__main__`" (una cadena de caracteres). Por lo tanto, siempre podemos saber si alguien ha pedido *ejecutar* un módulo (y no *importar* ese módulo) incluyendo (*al final* del mismo) un script con la condición [2] [1]:

```
if __name__ == "__main__":
```

En la rama verdadera de esta condición, se puede escribir el script que el programador considere oportuno ejecutar en ese caso. Por ejemplo, se puede incluir allí la invocación a la función que se haya definido como *función de entrada* (si es que había alguna). El siguiente ejemplo muestra la forma de hacer eso en el archivo `prueba05.py`:

```
__author__ = 'Cátedra de AED'

# Archivo prueba05.py
from soporte import *

def test():
    a = int(input('Cargue un número entero: '))
    f = factorial(a)
    print('El factorial del número es:', f)
    r = menor(a,f)
    print('Menor:', r)

# script principal...
# si se pidió ejecutar el modulo, entonces
# lanzar la funcion test()
if __name__ == "__main__":
    test()
```

Remarcamos: si se utiliza este recurso, la función `test()` será ejecutada *pero sólo si se pidió a su vez ejecutar el módulo `prueba05.py` que la contiene; y no si este módulo fue importado por otro*. En el siguiente ejemplo, el módulo `prueba05.py` está siendo *importado* desde un script sencillo, pero la función `prueba05.test()` no será ejecutada en forma automática: el programador debe invocarla explícitamente para que se ejecute:

```
# esta linea no provoca la ejecución de la función test()...
import prueba05

# ...pero esta sí...
prueba05.test()
```

Esta forma de controlar si se pidió la ejecución de un módulo o no, finalmente **muestra la manera más práctica de plantear un programa en Python**. Hemos visto que siempre se puede tener una función designada *ad hoc* como función de entrada, y podemos hacer que un módulo incluya simplemente una invocación a esa función al final del mismo. Pero en ese caso, si el módulo se importa desde otro (en lugar de pedir que se ejecute), la función de entrada se ejecutará y esto puede provocar un caos en el control de flujo de ejecución. La inclusión del control de ejecución indicado más arriba impide que se produzca esta situación dándole al programador el control total del caso y convirtiendo al módulo en un **módulo ejecutable**.

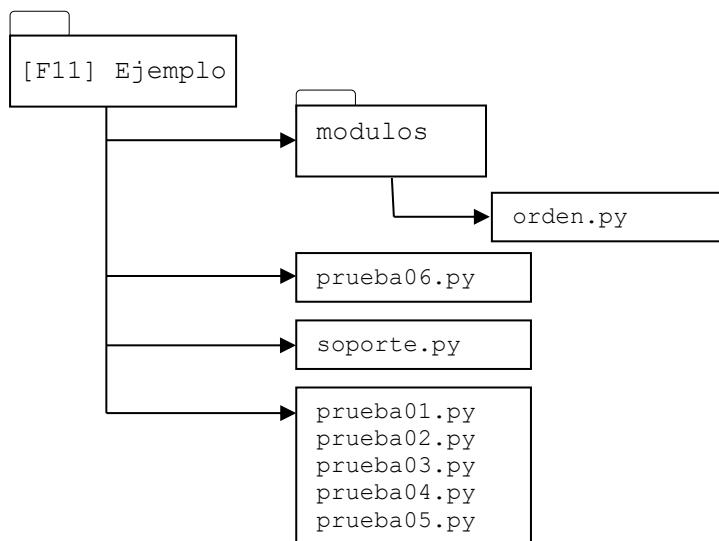
Para terminar esta sección, digamos que en los ejemplos que hemos visto hasta aquí los módulos usados eran o bien nativos (o *internos*) del lenguaje, o bien eran módulos provistos por el programador (y en este último caso, estaban alojados en la misma carpeta que contenía al programa o módulo que lo importaba). Es bueno saber qué reglas sigue el intérprete cuando encuentra una instrucción *import nombre\_modulo* para encontrar el módulo requerido [1]:

1. Lo primero que analiza el intérprete, es si existe algún *módulo interno* con ese nombre.
2. Si no lo encuentra así, busca el nombre del módulo en una lista de carpetas dada por la variable *sys.path* (es decir, la variable *path* contenida en el módulo interno *sys*). Esta variable es automáticamente inicializada con los siguientes valores:
  - El nombre del directorio actual (donde está el script o programa que realizó el *import*)
  - El valor de la variable de entorno PYTHONPATH (que se usa en forma similar a la variable PATH del sistema operativo)
  - Y otros valores default que dependen de la instalación Python.

Note que la variable *sys.path* contiene una cadena de caracteres con la información enumerada en el párrafo anterior asignada por default. Sin embargo, el programador puede modificar esa variable usando funciones directas de manejo de cadenas, como la función *append()* que permite añadir una cadena a otra.

A modo de ejemplo, analicemos el mismo proyecto [F11] Ejemplo que viene anexo a esta Ficha. La estructura de carpetas y archivos de este proyecto es la siguiente:

**Figura 1: Estructura de carpetas y subcarpetas del proyecto [F11] Ejemplo.**



En esta carpeta de proyecto hay una subcarpeta llamada *módulos*, la cual contiene un único módulo muy sencillo llamado *orden.py*, cuyo contenido es el siguiente:

```

__author__ = 'Cátedra de AED'

# archivo orden.py
# Un modulo simple con una función única a modo de ejemplo simple...

# una función para retornar el menor entre dos números...
def menor(n1, n2):
  
```

```

if n1 < n2:
    return n1
return n2

```

Además, en la carpeta del proyecto *[F11] Ejemplo* se encuentran todos los programas que hemos analizado en esta sección (*prueba01.py*, *prueba02.py*, ...) junto con el módulo *soporte.py* que hemos analizado hasta aquí y el programa *prueba06.py* que se muestra más abajo:

```

__author__ = 'Cátedra de AED'

# Archivo: prueba06.py

# Modificación de la variable sys.path para incluir una subcarpeta de modulos
import sys
sys.path.append('.\\modulos')

# Importación del modulo "orden" que está en la carpeta "modulos"...
import orden

# función de entrada a modo de prueba...
def test():
    a = int(input('Ingrese un numero: '))
    b = int(input('Ingrese otro: '))
    men = orden.menor(a, b)
    print('El menor es:', men)

# control de solicitud de ejecución del modulo...
if __name__ == '__main__':
    test()

```

En el ejemplo que se acaba de mostrar, el archivo *prueba06.py* (un módulo ejecutable...) realiza un *import* para acceder al módulo *orden.py*, pero como ese módulo se encuentra a su vez grabado en la subcarpeta *modulos* (como subcarpeta del proyecto *F[11] Ejemplo* que incluye a *prueba06.py*) entonces se cambia el valor de la variable *sys.path* para añadir a ella la ruta de la subcarpeta ".\\modulos". Note que el propio módulo interno *sys* debe ser importado para acceder a la variable *sys.path*.

Otro detalle técnico interesante es que cuando un módulo es importado por primera vez en la ejecución de un programa o script, se crea un archivo cuyo nombre incluye al nombre del módulo como parte (en el caso del módulo *soporte.py*, el archivo creado se llama *soporte.cpython-34*), pero con extensión *.pyc* (por *Compiled Python File*).

Este archivo contiene una versión "precompilada" del módulo (al estilo de los archivos *.class* de Java). Esto se hace para agilizar el proceso de carga de un módulo, lo cual es muy práctico si el programa usa muchos módulos ya que cada módulo no debe ser compilado nuevamente cada vez que se lo cargue: se compila la primera vez que es importado, y luego se toma el archivo *.pyc* cuando se vuelve a importar.

El lugar donde se ubican estos archivos depende del IDE que esté usando: si está trabajando con *PyCharm*, entonces en la misma carpeta del proyecto se creará una subcarpeta *\_pycache\_* conteniendo a ese archivo. Obviamente, esos archivos precompilados *no son editables* (el programador no puede simplemente abrir este tipo de archivos con un editor de texto), pero son independientes del sistemas operativo (se dice que son *independientes de la plataforma*), por lo cual un archivo de módulos Python precompilado puede ser compartido entre diversas arquitecturas y sistemas operativos sin tener que volver a compilarlo.

### 3.] La librería estándar de Python

Como es de esperar, Python provee una amplísima librería de módulos estándar listos para usar. Ya hemos visto, a modo de ejemplo, una mención al módulo `sys` que provee acceso a muchas variables usadas o mantenidas por el intérprete y/o por funciones que interactúan fuertemente con el intérprete. Solo a modo informativo (y confiando en la sana curiosidad de cada estudiante...) mostramos aquí una tabla sencilla, a modo de referencia y descripción muy simple de los distintos *módulos estándar de Python* (la lista no es exhaustiva... solo es una contribución de consulta inmediata):

Módulo estándar	Contenido general
<code>os</code>	Interfaz de acceso a funciones del sistema operativo.
<code>glob</code>	Provee un par de funciones para crear listas de archivos a partir de búsquedas realizadas con caracteres "comodines" (wildcards)
<code>sys</code>	Variables de entorno del shell, y acceso a parámetros de línea de órdenes. También provee elementos para redireccionar la entrada estándar, la salida estándar y la salida de errores estándar.
<code>re</code>	Herramientas para el reconocimiento de patrones (expresiones regulares) en el procesamiento de strings.
<code>math</code>	Funciones matemáticas típicas (logarítmicas, trigonométricas, etc.)
<code>random</code>	Funciones para el trabajo con números aleatorios y decisiones aleatorias.
<code>urllib.request</code>	Funciones para recuperación de datos desde un url.
<code>smtplib, poplib</code>	Funciones para envío y recepción de mails.
<code>email</code>	Gestión integral de mensajes de email, permitiendo decodificación de estructuras complejas de mensajes (incluyendo attachments).
<code>datetime</code>	Clases para manipulación de fechas y horas.
<code>zlib, gzip, bz2, zipfile, tarfile</code>	Los cinco módulos citados contienen funciones para realizar compresión/descompresión de datos, en diversos formatos.
<code>timeit, profile, psstests</code>	Proveen funciones para realizar medición de rendimiento de un programa, sistema, bloque de código o instrucción.
<code>doctest</code>	Herramienta para realizar validaciones de tests que hayan sido incluidos en strings de documentación.
<code>unittest</code>	Similar a doctest, aunque no tan sencillo. Permite realizar tests más descriptivos, con la capacidad de poder mantenerlos en archivos separados.
<code>xmlrpc.client, xmlrpcl.server</code>	Permiten realizar llamadas a procedimientos remotos, como si fuesen tareas triviales.
<code>xml.dom, xml.sax</code>	Soporte para el parsing de documentos XML.
<code>csv</code>	Lectura y escritura en formato común de "comma separated values" (CSV).
<code>gettext, locale, codecs</code>	Soporte de " <i>internationalization</i> ".

El acceso a todos los módulos nombrados (y otros que podrían no figurar en la tabla) requiere el uso de la instrucción `import` o alguna de sus variantes.

Además, y también en forma esperable, Python provee un amplio conjunto de funciones internas, que están siempre disponibles sin necesidad de importar ningún módulo. La lista completa de estas funciones puede verse en la documentación de Python que se instala junto con el lenguaje, y de todos modos en la *Ficha 3, página 61*, hemos mostrado oportunamente una lista de las más comunes de esas funciones.

### 4.] Definición de *paquetes* en Python.

Un *paquete* (o *package*) en Python es una forma de organizar y estructurar *carpetas de módulos* (y los espacios de nombres que esos módulos representan), de forma que luego se

pueda acceder a cada módulo mediante el conocido recurso del operador *punto* para especificar la ruta de acceso y el nombre de cada módulo. Así, por ejemplo, si se tiene un paquete (o sea, una *carpeta de módulos*) llamado *modulos*, y dentro de él se incluye un módulo llamado *funciones*, entonces el nombre completo del módulo sería *modulos.funciones* y con ese nombre deberá ser accedido desde una instrucción *import*.

Además de facilitar una mejor organización y mantenimiento de los módulos disponibles en un gran proyecto o sistema, el uso de *paquetes* permite evitar posibles ambigüedades o conflictos de nombres que podrían producirse si diversos programadores o equipos aportan módulos diferentes que incluyan funciones llamadas igual, y otros tipos similares de conflictos. El nombre completo del *módulo* incluida la ruta de acceso dentro del *paquete* y el *operador punto* para acceder a la función, evita el conflicto.

Si bien la creación de un *paquete de módulos* no conlleva un proceso complicado en Python, tampoco es tan directo como simplemente armar una estructura de carpetas y subcarpetas y luego distribuir en ellas los módulos disponibles. Para que el intérprete Python reconozca la designación de un *paquete* mediante el *operador punto* y la traduzca desde una estructura de carpetas física, esas carpetas deben incluir (*cada una de ellas*) un archivo fuente Python llamado *\_\_init\_\_.py*, el cual puede ser tan sencillo como un archivo vacío, o puede incluir scripts de inicialización para el paquete, o asignar valores iniciales a ciertas variables globales que luego describiremos.

Llegados a este punto, conviene aclarar un hecho práctico: no todos los IDEs para Python proveen la funcionalidad de crear y administrar paquetes de módulos en forma automática a partir de opciones de menú. Pero nuestro IDE *PyCharm* es uno de los que sí lo permiten.

Si está trabajando con *PyCharm*, y quiere crear un *paquete de módulos* dentro de un proyecto, simplemente pida crear un nuevo proyecto (en lugar de un archivo fuente) y dentro de ese proyecto incluya un *paquete*: apunte al proyecto con el mouse, haga click derecho, seleccione la opción *New* en el menú emergente que aparece, y luego seleccione el ítem "*Python Package*".

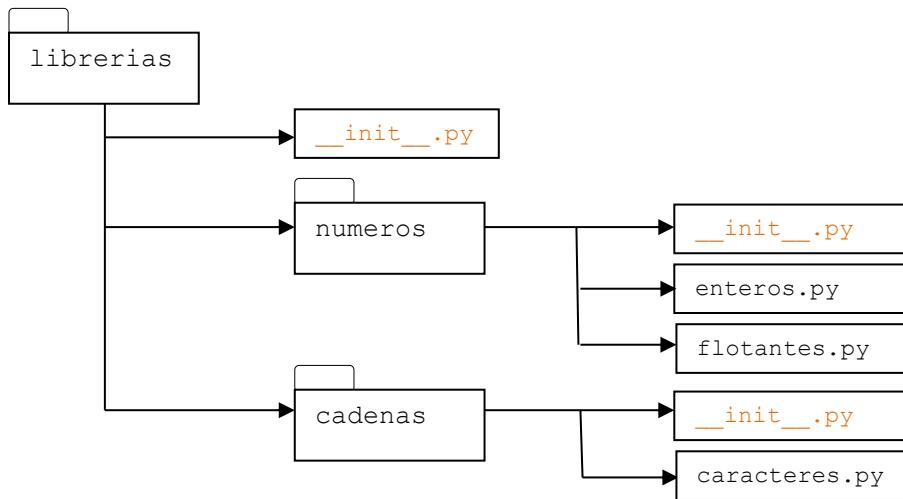
Dentro del proyecto se creará una carpeta con el nombre que usted haya elegido para el paquete, y esa carpeta ya contendrá su correspondiente archivo *\_\_init\_\_.py* (posiblemente vacío o solo conteniendo alguna inicialización de alguna variable global, tal como la variable *\_\_author\_\_* asignada con el nombre del usuario o creador del *paquete*). Luego de creado el *paquete*, puede crear *subpaquetes* dentro de él, repitiendo el procedimiento marcado más arriba, pero ahora apuntando con el mouse al *paquete* dentro del cual desee crear un *subpaquete*.

A modo de ejemplo, junto con esta Ficha se incluye otra carpeta [*F11*] [*Paquetes*], que no es otra cosa que un proyecto creado con *PyCharm*. Dentro de ese proyecto se ha incluido un paquete llamado *librerias*, que contiene a su vez dos subpaquetes llamados *cadenas* y *numeros* respectivamente. Los archivos *\_\_init\_\_.py* de estas tres carpetas solo contienen la inicialización de la variable *\_\_author\_\_* (puede abrir estos archivos con el mismo editor de *PyCharm* si quiere ver y/o editar su contenido).

A su vez, el paquete *librerias.numeros* contiene dos módulos de funciones: uno se llama *enteros.py*, el otro se llama *flotantes.py*, y contienen algunas funciones sencillas para manejar números. El paquete *librerias.cadenas* contiene solo un módulo llamado

*caracteres.py* que incluye alguna función simple para manejo de cadenas y caracteres. El siguiente esquema gráfico muestra la estructura del paquete completo:

Figura 2: Estructura de carpetas del paquete librerias.



Note que cada carpeta o subcarpeta que forma parte de la estructura del paquete se considera ella misma como un paquete o un subpaquete, pero debe tener para ello su propio archivo `__init__.py` (vacío, con scripts de inicialización, o con asignación en alguna variable de control).

El módulo *números.enteros* tiene el siguiente contenido:

```

__author__ = 'Cátedra de AED'

# una función para retornar el menor entre dos números...
def menor(n1, n2):
    if n1 < n2:
        return n1
    return n2

# una función para calcular el factorial de un número...
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

# una función para ordenar dos números...
def ordenar(n1, n2, ascendent=True):
    first, second = n2, n1
    if n1 < n2:
        first, second = n1, n2

    if not ascendent:
        first, second = second, first

    return first, second
  
```

El módulo *numeros.flotantes* tiene a su vez el siguiente contenido:

```
__author__ = 'Cátedra de AED'

# una funcion para obtener el promedio entre los parametros
def promedio (x1, x2, *x):
    suma = x1 + x2
    conteo = 2

    n = len(x)
    if n != 0:
        for i in range(n):
            suma += x[i]
            conteo += 1

    return suma / conteo
```

Finalmente, el módulo *cadenas.caracteres* se compone así:

```
__author__ = 'Cátedra de AED'

# una funcion que determina si la cadena tomada como parametro esta
# formada por un unico caracter que se repite
def caracter_unico(cadena):
    n = len(cadena)

    # si la cadena no tiene caracteres, retornar falso
    if n == 0:
        return False

    c0 = cadena[0]
    for i in range(1, n):
        if c0 != cadena[i]:
            return False
    return True
```

Si se desea acceder a módulos que han sido almacenados en paquetes o subpaquetes, se puede usar la ya conocida instrucción *import*, pero ahora escribiendo la ruta completa de paquetes y subpaquetes delante del nombre del módulo que se quiere acceder, separando los nombres de las carpetas con el operador punto. Desde el proyecto *F[11] Paquetes* mostramos el archivo *test01.py* con un ejemplo de uso:

```
__author__ = 'Cátedra de AED'

# Estos dos import acceden a los modulos "flotantes" y "caracteres"
import librerias.numeros.flotantes
import librerias.cadenas.caracteres

def test():
    p = librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    if librerias.cadenas.caracteres.caracter_unico('abcde'):
        print('La cadena tiene repeticiones de un unico caracter...')
    else:
        print('La cadena tiene varios caracteres distintos...')
```

```
if __name__ == '__main__':
    test()
```

Como de costumbre, la instrucción *import* introduce el nombre del módulo como un nombre válido para el programa, y si se quiere invocar a una función de ese módulo se debe seguir usando el nombre completo de la misma. La siguiente línea invoca a la función *promedio()* que se encuentra dentro del módulo *flotantes* del paquete *librerías.numeros*:

```
librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
```

También puede usar la también conocida variante *from – import* si desea ser más específico en cuanto a los elementos que quiere acceder desde un módulo (ver ahora el archivo fuente *test02.py* del mismo proyecto [F11] Paquetes):

```
__author__ = 'Cátedra de AED'

# el from - import que sigue importa los modulos "flotantes" y "enteros"
from librerias.numeros import flotantes, enteros

# el from - import que sigue importa la funcion caracter_unico()
from librerias.cadenas.caracteres import caracter_unico


def test():
    p = flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    m = enteros.menor(4,6)
    print('El menor es:', m)

    if caracter_unico('aaaaaa'):
        print('La cadena tiene solo repeticiones de un unico caracter...')
    else:
        print('La cadena tiene varios caracteres distintos...')

if __name__ == '__main__':
    test()
```

La primera instrucción *from – import* de este ejemplo da acceso al nombre de los módulos *flotantes* y *enteros*, de forma que ya no es necesario luego incluir toda la ruta de paquetes y subpaquetes para nombrarlos:

```
p = flotantes.promedio(2.34, 4, 5.34)
m = enteros.menor(4,6)
```

El segundo *from – import* del ejemplo, da acceso específicamente a la función *caracter\_unico()* del módulo *caracteres*, evitando por completo el tener que poner la ruta de paquetes:

```
if caracter_unico('aaaaaa'):
```

Es siempre posible usar *from – import* e incluir un asterisco (\*) para garantizar que el nombre de ese paquete sea tomado en forma implícita en todos sus elementos. Una instrucción como:

```
from librerias.numeros import *
```

permite acceder a los elementos incluidos en el paquete *librerias.numeros* sin tener que repetir el prefijo *librerias.numeros*. Sin embargo, considere que un paquete podría contener un sinnúmero de subpaquetes y módulos, y eventualmente el creador del paquete principal podría querer limitar lo que se accede mediante un *from – import \** (por ejemplo, algún módulo que sólo fue creado a los efectos de permitir pruebas (o *testing*). Para lograr esa restricción, se puede recurrir al archivo *\_\_init\_\_.py* del paquete cuyo contenido se quiere controlar, y asignar en la variable *\_\_all\_\_* una lista de cadenas de caracteres con los nombres de los módulos que serán importados con un *from – import \**.

En la carpeta de fuentes que acompaña a esta Ficha, se incluye otro proyecto *F[11] Import*, que a su vez contiene el mismo paquete *librerias* del modelo anterior, pero con un pequeño agregado: el subpaquete *librerias.numeros* agrega un módulo más, llamado *testing.py* con una sola función sencilla llamada *mensaje()*. El archivo *\_\_init\_\_.py* del paquete *librerias.numeros* contiene las siguientes instrucciones Python [1]:

```
# el autor del módulo...
__author__ = 'Cátedra de AED'

# los módulos que serán incluidos en un from numeros import *
__all__ = ['enteros', 'flotantes']
```

Como dijimos, en el proyecto se encuentra el archivo *test01.py*, que contiene lo siguiente:

```
__author__ = 'Cátedra de AED'

# el from - import que sigue importa SOLO los modulos "flotantes"
# y "enteros"... ver archivo __init__.py de este paquete...
from librerias.numeros import *

def test():
    p = flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    m = enteros.menor(4,6)
    print('El menor es:', 4)

    # esta funcion esta en el modulo librerias.numeros.testing...
    # pero ese modulo no fue incluido en el from - import *...
    # testing.mensaje('Solo para probar...') # error...

if __name__ == '__main__':
    test()
```

Puede verse que este programa contiene una instrucción *from librerias.numeros import \**, pero esta instrucción NO importará el nombre del módulo *testing.py*, ya que el mismo no había sido incluido en la lista de nombres de la variable *\_\_all\_\_* del paquete. De hecho, la invocación:

```
testing.mensaje("Solo para probar...")
```

está comentarizada ya que de otro modo provocará un error de interpretación.

## 5.] Cadenas de documentación (*docstrings*) en Python.

Un elemento importante a considerar en la fase de desarrollo de un sistema informático es la documentación técnica del mismo. Sabemos que esa tarea es harto tediosa y larga, y pocos profesionales de la informática gustan de hacerla. Sin embargo, todos comprendemos la importancia de dejar claramente expresado para qué sirve cada programa o módulo que hayamos desarrollado, cuál es el objetivo de cada uno, qué funciones contiene, para qué sirve cada una, qué retorna cada una, o qué significa cada uno de sus parámetros formales. En otras palabras, es importante dejar a otros miembros de un proyecto o a otros eventuales programadores los manuales técnicos de nuestros desarrollos.

Por otra parte, esa misma documentación técnica es utilizada en forma automática por algunos IDEs para ofrecer ayuda de contexto al programador cuando este tiene alguna duda respecto de como usar una función (por ejemplo). En *PyCharm*, puede apuntar con el mouse a una función cualquiera y pulsar la combinación de teclas *<Ctrl> Q* para ver una pequeña ventana emergente conteniendo la documentación técnica de esa función (si es que esa documentación existe...)

En forma similar a lo que se puede hacer en el lenguaje *Java* mediante los llamados comentarios *javadoc*, *Python* permite introducir elementos llamados *cadenas de documentación* o *docstrings* en el código fuente, de tal forma que el contenido de los bloques *docstring* pueda luego ser usado para generar archivos de documentación general sobre nuestros programas y módulos [1].

Una cadena de documentación es un string literal, encerrado entre comillas triples (""""), que puede desplegarse *en una única línea* o *en un bloque de varias líneas*:

```
"""Ejemplo de docstring de unica linea"""

"""Ejemplo de docstring de multiples lineas
Aqui continua...
Y en esta linea termina
"""
```

Los *docstrings* pueden usarse para especificar detalles de contenido y/o funcionamiento de una función, una clase, un método, un módulo o incluso un paquete. Luego, la información contenida en los *docstrings* de cualquiera de estos elementos puede ser obtenida por herramientas analizadoras de código que generen texto de ayuda (designadas en general como *parsers*), como ejemplo, la función *help()* del shell de Python, o la herramienta *pydoc* que viene incluida en el módulo *pydoc.py* del SDK de Python.

Hay algunas reglas y convenciones que se espera que se respeten al incluir información *docstring* en un elemento [1]:

- Una cadena *docstring* *debe aparecer como la primera línea* dentro del elemento que se está describiendo.
- Normalmente, la primera línea dentro de la cadena debe ser una descripción corta y somera del elemento que se está documentando (típicamente, no más de una línea de texto). Esta línea debería comenzar con letra mayúscula y terminar con un punto, y no debería contener información que describa el nombre del elemento (o sus parámetros o su tipo de retorno) ya que estos ítems normalmente son identificados automáticamente por introspección.
- Si el bloque *docstring* va a contener varias líneas, la siguiente línea debería quedar en blanco.

- A partir de la tercera línea debería realizarse un resumen un poco más detallado acerca de la forma de usar el objeto descripto, sus parámetros y valor returned (si los hubiese), excepciones y situaciones a controlar, etc.
- La información *docstring* de un *módulo* debería generalmente listar las clases, excepciones, funciones y cualquier otro elemento que el módulo contenga, mediante una línea de texto breve para cada uno. Esta lista breve debería brindar menos detalles que el resumen general que cada objeto tendrá a su vez en sus docstring.
- La información *docstring* de un *package* se escribe dentro del archivo `__init__.py` que lo describe, y también debería incluir una lista breve de los módulos y subpaquetes exportados por ese paquete.
- Finalmente, el *docstring* de una *función* debería resumir su comportamiento y documentar sus parámetros, tipo de retorno, efectos secundarios, excepciones lanzadas y restricciones de uso (si son aplicables) Los parámetros optionales deberían ser indicados, y también sus parámetros de palabra clave.

Para más información y detalles, invitamos a consultar la página sobre convenciones de uso referidas a *docstring*, en el url <http://www.python.org/dev/peps/pep-0257/> (incluido entre la documentación PEP de consulta del sitio oficial de Python).

El proyecto [F11] *Docstrings* incluido con esta Ficha, contiene el mismo proyecto que ya presentamos como F[11] *Paquetes*, pero de tal forma que ahora los paquetes, módulos y funciones del proyecto han sido brevemente documentados mediante *docstrings*. Así, el archivo `__init__.py` del paquete *librerias* contiene ahora lo siguiente:

```
"""El paquete contiene subpaquetes para operaciones con numeros y con cadenas.
```

```
Lista de subpaquetes incluidos:
:cadenas: Contiene un modulo con funciones para manejo de caracteres
:numeros: Contiene dos modulos con funciones para numeros enteros y en coma
flotante
"""
__author__ = 'Cátedra de AED'
```

Note que el bloque de documentación *docstring* comienza desde la primera línea del archivo `__init__.py`, y que sólo después de este bloque aparece la consabida asignación en la variable `__author__` del archivo. En forma similar se ha procedido con los archivos `__init__.py` de los subpaquetes *numeros* y *cadenas*.

El módulo *enteros.py* luce ahora en la forma siguiente (y el resto de los módulos del proyecto en forma similar):

```
"""Funciones generales para manejo de numeros enteros.

Lista de funciones incluidas:
:menor(n1, n2): Retorna el menor entre dos numeros
:factorial(n): Retorna el factorial de un numero entero
:ordenar(n1, n2, ascendent = True): Ordena dos numeros
"""
__author__ = 'Cátedra de AED'

def menor(n1, n2):
    """Retorna el menor entre dos numeros.

    :param n1: El primer numero a comparar
    :param n2: El segundo numero a comparar
```

```

:rtype: El menor entre n1 y n2
"""
if n1 < n2:
    return n1
return n2

def factorial(n):
    """Retorna el factorial de un numero.

    :param n: El numero al cual se le calculara el factorial
    :return: El factorial de n, si n>=0. Si n<0, retorna None.
    """
    f = 1
    for i in range(2,n+1):
        f *= i
    return f

def ordenar(n1, n2, ascendent = True):
    """Ordena dos numeros.

    :param n1: El primero de los numeros a ordenar
    :param n2: El segundo de los numeros a ordenar
    :param ascendent: True ordena de menor a mayor - False en caso contrario
    :return: Los dos numeros, ordenados segun ascendent
    """
    first, second = n2, n1
    if n1 < n2 :
        first, second = n1, n2
    if not ascendent :
        first, second = second, first
    return first, second

```

Observe que el *docstring* de cada función aparece dentro del bloque de la función, pero inmediatamente luego de la cabecera de la misma (como dijimos: el *docstring* debe ir en la primera línea del objeto descripto). Dependiendo del IDE que esté usando, el programador podrá disponer de mayor o menor ayuda de contexto para generar esta documentación (el IDE *PyCharm*, por ejemplo, incorpora abundantes elementos de ayuda en el editor de textos o en sus menús de opciones).

Las cadenas literales que conforman los *docstrings* de un objeto cualquiera (módulo, función, etc.) se asignan en la variable global *doc* que forma parte de los atributos o elementos contenidos en ese objeto. En ese sentido, el archivo *test01.py* del mismo proyecto que estamos analizando a modo de ejemplo, incluye una simple invocación a la función *print()* para mostrar el valor de *doc* en forma directa [1]:

```

__author__ = 'Cátedra de AED'

import librerias.numeros.flotantes
import librerias.cadenas.caracteres

def test():
    p = librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    if librerias.cadenas.caracteres.caracter_unico('abcde'):
        print('La cadena tiene una o varias repeticiones de un unico caracter...')
    else:
        print('La cadena tiene varios caracteres distintos...')

print('Contenidos docstring del modulo:')
print(librerias.cadenas.__doc__)

```

```
if __name__ == '__main__':
    test()
```

La salida producida por esas dos instrucciones `print()` es la siguiente:

```
Contenidos docstring del modulo:
El paquete contiene un modulo con funciones para operar con caracteres y
cadenas.

Lista de modulos incluidos:
:caracteres: Contiene funciones simples para manejo de caracteres
```

Una vez que los docstrings se han incluido en el código fuente, se puede generar la documentación de ayuda mediante diversas herramientas más o menos sofisticadas. Si ha trabajado en el shell directamente, una invocación a la función `help()` tomará las cadenas que haya incluido como *docstrings* y las mostrará directamente en la consola de salida (sin guardar nada en ningún archivo externo). Sólo debe enviarle como parámetro el nombre del elemento cuyos docstrings quiere observar:

The screenshot shows the Python Shell window with the title '74 Python Shell'. The window contains the following text:

```
File Edit Shell Debug Options Windows Help
>>> def prueba():
        """Una funcion de prueba"""
        print("Ejemplo")

>>> help(prueba)
Help on function prueba in module __main__:

prueba()
    Una funcion de prueba

>>> |
```

The status bar at the bottom right indicates 'Ln: 43 Col: 4'.

Obviamente, lo anterior resulta muy limitado cuando se trata de programas o sistemas grandes desarrollados desde un IDE. Python provee para estos casos el **módulo ejecutable `pydoc.py`** que permite generar la documentación de ayuda en forma similar a la aplicación `javadoc` de Java (aunque `pydoc` es bastante menos flexible).

El módulo `pydoc.py` está almacenado en la carpeta Lib del SDK de Python (típicamente, en Windows la ruta de esa carpeta es algo como `C:\Program Files\Python 3.3.0\Lib`). Lo normal es ejecutar el módulo desde la línea de órdenes del sistema operativo, para lo cual esta ruta debería estar asociada la variable PATH del sistema operativo.

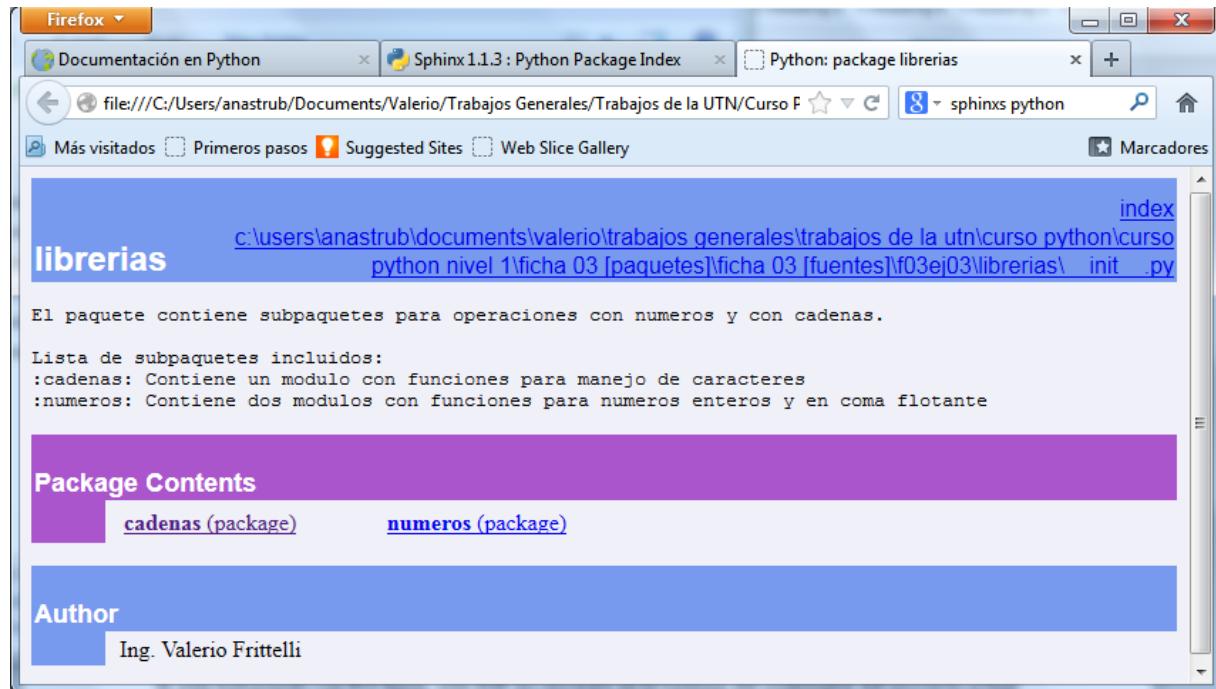
En estas condiciones, se puede acceder a la consola de ejecución por línea de órdenes del sistema operativo, luego cambiarse a la carpeta que contenga al paquete o módulo cuya documentación docstring se quiere analizar, y desde allí ejecutar el módulo `pydoc` enviándole el nombre del objeto a analizar. Por ejemplo, nuestro proyecto está en la carpeta "`C:\F[11] Docstrings`" y dentro de esa carpeta se encuentra ya el paquete `librerías`, entonces desde la línea de órdenes del sistema la siguiente orden mostrará en pantalla los docstrings de ese paquete:

```
C:\F[11] Docstrings>pydoc librerias
```

El módulo `pydoc.py` admite ciertos parámetros de ejecución, como el parámetro `-w` que indica al parser que almacene la documentación en archivos con formato html en lugar de mostrarla por consola:

```
C:\F[11] Docstrings>pydoc -w librerias
```

La orden anterior tomará los *docstrings* del paquete *librerias*, creará un documento html con esos elementos, y almacenará el documento en la misma carpeta actual (en este caso, la carpeta del proyecto). Así se ve el archivo html generado de esta forma, cuando se lo despliega en un navegador web:



Puede acceder a subpaquetes y/o submódulos mediante su nombre calificado vía el operador punto. Lo siguiente creará un archivo html con los docstrings del paquete *librerias.numeros* (y note que el archivo quedará almacenado en la misma carpeta del proyecto, con el nombre *librerias.numeros.html*)

```
C:\F[11] Docstrings>pydoc -w librerias.numeros
```

De todos modos, para no tener que generar estos archivos html uno por uno, puede ejecutar la aplicación con el siguiente formato:

```
C:\F[11] Docstrings>pydoc -w librerias .\
```

Al hacerlo de esta forma, el programa explorará en forma recursiva todos los subpaquetes del paquete *librerias*, y todos los módulos incluidos en esos paquetes, y creará los archivos html de todos ellos en una sola corrida. Todos esos archivos quedarán guardados en la misma carpeta del proyecto.

Si está trabajando con *PyCharm*, este IDE incorporará en la ventana del explorador del proyecto actual los documentos html que se generen de esta forma, y el programador podrá ver su código fuente (y editarla si lo desea). También permitirá acceder a esta información

para un elemento en particular, colocando el puntero del mouse sobre él (el nombre de una función, por ejemplo) y pulsando la combinación de teclas  $<Ctrl> + Q$ .

Si bien el módulo *pydoc.py* permite otros parámetros de ejecución, el hecho es que aún así resulta muy poco flexible como herramienta integrada (por caso, no permite modificar el estilo de los documentos html generados agregando etiquetas html especiales). Esta flexibilidad puede lograrse mediante herramientas especiales para generación de documentación (normalmente, proyectos desarrollados por organizaciones que incluso podrían ser externas a Python Software Foundation), que deben ser descargadas e instaladas por separado. Algunas de estas herramientas son muy comunes y muy usadas por la comunidad Python. Indicamos a continuación una breve lista de algunas de ellas, y dejamos para el lector la tarea de investigar formas de uso y aplicaciones:

- i. ***Docutils***: es un sistema modular para procesamiento de documentación hacia formatos útiles como HTML, XML y LaTeX. A modo de entrada, la herramienta *Docutils* soporta un estándar de marcas de formato fácil de leer, llamado *reStructuredText*. Se puede descargar desde: <https://pypi.python.org/pypi/docutils>.
- ii. ***EpyDoc***: es una de las herramientas de generación de documentación para Python más utilizadas. Incluye la definición de su propio formato (*epytext*). Produce salidas de texto plano, pero soporta también *reStructuredText* y sintaxis *Javadoc*. Se puede descargar desde el siguiente url: <https://pypi.python.org/pypi/epydoc>.
- iii. ***Sphinx***: Originalmente planteada para Python, se puede usar sin problemas con C y C++ y se planea expandirla a otros lenguajes. Soporta HTML, LaTeX, Texinfo y texto plano. Se puede descargar desde el url: <https://pypi.python.org/pypi/Sphinx>.

---

## Bibliografía

- [1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 12

## Arreglos Unidimensionales

### 1.] Introducción.

Hasta aquí se estudió la forma de resolver problemas de diversas características usando instrucciones y técnicas de programación variadas: algunos problemas requerían sólo del uso de condiciones y secuencias de instrucciones simples, en otros se hizo muy necesario el desarrollo de funciones y una fuerte división en subproblemas, y otros requerían ya el uso de ciclos.

Si observamos con detalle los últimos problemas que se han estado planteando (donde se utilizaron instrucciones repetitivas) y se los compara con los primeros problemas presentados en el curso (en los que los ciclos ni siquiera se conocían), podremos notar que las instrucciones de repetición se hicieron imperativamente necesarias para poder manejar un *gran volumen de datos* (o sea, una gran *cantidad* de datos). En los primeros problemas del curso, los datos que el programa necesitaba se cargaban *todos a la vez* en cierto conjunto pequeño de variables, y luego esas variables se procesaban usando instrucciones simples y condiciones [1].

Pero en los últimos problemas analizados, o bien era muy grande la cantidad de datos que pedía cada problema, o bien se desconocía en forma exacta la cantidad de datos que se presentarían. Ante esto, no resultaba ni práctico ni posible cargar todos los datos a la vez, dado que esto conduciría a definir un conjunto demasiado grande de variables que luego sería muy difícil de procesar. El uso de ciclos brindó una buena solución: se define un pequeño conjunto de variables necesario sólo para cargar *un dato o un subconjunto de datos*, se carga *un dato o un subconjunto*, se procesa el mismo, y se *repite el esquema* usando un ciclo, hasta terminar con todos los lotes de datos. El esquema que conocimos como *carga por doble lectura* es un ejemplo de esta técnica.

Si bien pudiera parecer que con esto quedan solucionados todos nuestros problemas, en realidad estamos muy lejos de tal situación. Sólo piense el lector lo que pasaría si el conjunto de datos tuviera que ser procesado *varias veces* en un programa. Con el esquema de “un ciclo procesando un dato a la vez”, cada vez que el ciclo termina una vuelta se carga un *nuevo dato*, y se pierde con ello el dato anterior. Si como sugerimos, el programa requiriera volver a procesar un dato deberíamos volver a cargarlo, con la consecuente pérdida de tiempo y eficiencia general.

El re-procesamiento de un lote de datos es muy común en situaciones que piden *ordenar* datos y/o resultados de un programa, o en situaciones en las que un mismo conjunto de datos debe ser *consultado* varias veces a lo largo de una corrida (como el caso de los datos contenidos en una agenda, o los datos de la lista de productos de un comercio, por ejemplo).

Resulta claro que en casos como los planteados en el párrafo anterior, *no es suficiente* con usar un ciclo para cargar de a un dato por vez, perdiendo el dato anterior en cada repetición.

Ahora se requiere que los datos estén todos juntos en la memoria, y poder accederlos de alguna forma en el momento que se desee, sin perder ninguno de ellos.

¿Cómo mantener en memoria un volumen grande de datos, pero de forma que su posterior acceso y procesamiento sea relativamente sencillo? La respuesta a esta cuestión se encuentra en el uso de las llamadas *estructuras de datos*, que en forma muy general fueron presentadas en la *Ficha 03*.

Como vimos, esencialmente una *estructura de datos* es una variable que puede contener varios valores a la vez. A diferencia de las variables de *tipo simple* (que sólo contienen un único valor al mismo tiempo, y cualquier nuevo valor que se asigne provoca la pérdida del anterior), una *estructura de datos* puede pensarse como un conjunto de varios valores almacenados en una misma y única variable.

En ese sentido, sabemos que Python provee diversos tipos de *estructuras de datos* en forma de *secuencias*, y hemos usado con frecuencia algunas de ellas como las *cadenas de caracteres*, las *tuplas* y los *rangos*: en todos los casos, se trata de conjuntos de datos almacenados en una única variable [2].

Las *estructuras de datos* son útiles cuando se trata de desarrollar programas en los cuales se maneja un volumen elevado de datos y/o resultados, o bien cuando la organización de estos datos o resultados resulta compleja. Piense el alumno en lo complicado que resultaría plantear un programa que maneje los datos de todos los alumnos de una universidad, usando solamente variables simples como se hizo hasta ahora. Las *estructuras de datos* permiten agrupar en forma conveniente y ordenada todos los datos que un programa requiera, brindando además, muchas facilidades para acceder a cada uno de esos datos por separado.

Tan importantes resultan las estructuras de datos, que en muchos casos de problemas complejos resulta totalmente impracticable plantear un algoritmo para resolverlos sin utilizar alguna estructura de datos, o varias combinadas. No es casual entonces que la materia que estamos desarrollando se denomine *Algoritmos y Estructuras de Datos...*

## 2.] Arreglos unidimensionales en Python.

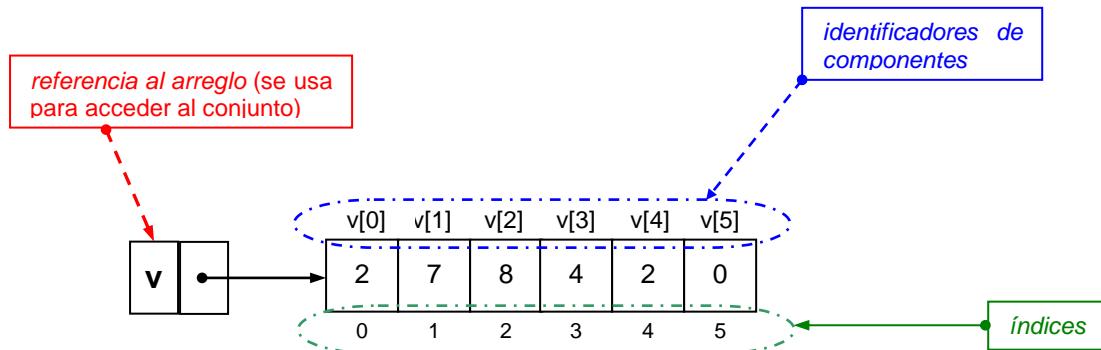
Una estructura de datos básica y muy importante en programación es la que se conoce como *arreglo unidimensional*. Se trata de una colección de valores que se organiza de tal forma que cada valor o componente individual es identificado automáticamente por un número designado como *índice*. El uso de los índices permite el acceso, uso y/o modificación de cada componente en forma individual.

La *cantidad de índices* que se requieren para acceder a un elemento individual, se llama *dimensión* del arreglo. Los *arreglos unidimensionales* se denominan así porque sólo requieren *un índice* para acceder a un componente [1]. Por otra parte, dada la similitud que existe entre el concepto de arreglo unidimensional y el concepto de vector en Álgebra, se suele llamar también *vectores* a los arreglos unidimensionales.

El gráfico de la *Figura 1* muestra la forma conceptual de entender un arreglo unidimensional. Se supone que la variable que permite manejar el arreglo se denomina *v* y que la misma contiene una referencia al arreglo (es decir, contiene la dirección de memoria donde está almacenado realmente el arreglo). En el ejemplo, el arreglo está dividido en seis casilleros,

de forma que en cada casillero puede guardarse un valor. Aquí supusimos que en cada casillero se almacenaron números enteros, pero obviamente podrían almacenarse valores de cualquier tipo (e incluso combinar valores de tipos distintos en un mismo arreglo).

**Figura 1: Esquema básico de un arreglo unidimensional.**



Observar que cada casillero<sup>1</sup> es automáticamente numerado con *índices*, los cuales en Python comienzan a partir del cero: la primera casilla del arreglo es subindicada con el valor cero, en forma automática. A partir de su índice cada elemento del arreglo referenciado por *v* puede accederse en forma individual usando el *identificador del componente*: se escribe el nombre de la variable que referencia al arreglo, luego un par de corchetes, y entre los corchetes el valor del índice de la casilla que se quiere acceder. En ese sentido, el *identificador del componente* cuyo índice es 2, resulta ser *v[2]*.

En Python, en general, se podría pensar que todos los tipos de *secuencias* provistas por el lenguaje (*cadenas*, *tuplas*, *rangos*, etc.) son esencialmente arreglos unidimensionales ya que para todas ellas sería aplicable la definición que acabamos de exponer. Sabemos que podemos acceder a un carácter de una cadena o a un elemento de una tupla a través de su índice y hemos aplicado esta idea en numerosas situaciones prácticas. Sin embargo, todos los tipos de secuencias que hemos visto y usado hasta ahora tenían una restricción: eran secuencias de tipo **inmutable**, lo cual quería decir que una vez creada una cadena o una tupla (por ejemplo) su contenido no podía cambiarse y cualquier intento de modificar el valor contenido en alguno de los casilleros provocaría un error de intérprete [2] [3].

Por lo tanto, estrictamente hablando, las *secuencias de tipo inmutable provistas por Python no representan operativamente arreglos unidimensionales*, ya que el programador esperaría poder *modificar* el valor de un casillero en cualquier momento, sin restricciones. Y en ese sentido, Python provee entonces un tipo de secuencia designado como *lista* (o *list*) que es el tipo que finalmente permite a un programador gestionar un arreglo en forma completa.

<sup>1</sup> La idea de una variable compuesta por casilleros lleva a pensar en tableros de juegos, y desde esa idea surge la asociación con la película *Jumanji* del año 1995, dirigida por *Joe Johnston* y protagonizada por *Robin Williams* y *Kirsten Dunst* (que era apenas una adolescente en ese momento). Un par de niños encuentran un raro juego de tablero (basado en la idea de recorrer una selva) llamado *Jumanji* y cuando comienzan a jugar descubren que las situaciones imaginarias asociadas a cada casillero se convierten en realidad cuando un jugador cae en esa casilla, haciendo del mundo un caos... En el año 2005 se lanzó una película muy similar (considerada como una secuela de *Jumanji*) llamada *Zathura* (dirigida por *Jon Favreau* y protagonizada por *Josh Hutcherson* y *Kristen Stewart* antes de ser la famosa protagonista de la serie *Crepúsculo*). En esta versión, *Zathura* es también un juego de tablero que hace realidad lo que ocurre en cada casilla, pero basado en una aventura espacial.

Una variable de tipo *list* en Python representa una secuencia ***mutable*** de valores de cualquier tipo, de forma que cada uno de esos valores puede ser accedido y/o *modificado* a partir de su índice [2]. Es el tipo *list* el que permite crear y usar variables que representen arreglos unidimensionales operativamente completos en Python y por lo tanto, de aquí en adelante, siempre supondremos que al hablar de *arreglos* (de cualquier dimensión) estaremos refiriéndonos a variables de tipo *list* en Python.

Para crear un arreglo unidimensional *v* inicialmente vacío en Python, se debe definir una variable de tipo *list* en cualquiera de las dos formas que mostramos a continuación [2] [3]:

```
v = []
v = list()
```

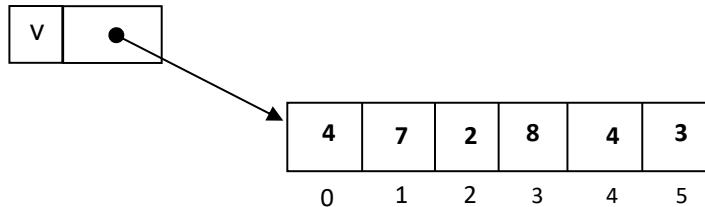
El par de corchetes vacíos representa justamente un arreglo vacío. La función constructora *list()* sin parámetros, también permite crear un arreglo vacío.

También se puede definir un arreglo no vacío por *enumeración de componentes*. La siguiente instrucción crea un arreglo *v* con seis componentes de tipo *int* (que en este caso serán los números 4, 7, 2, 8, 4 y 3):

```
v = [4, 7, 2, 8, 4, 3]
```

La instrucción anterior crea un arreglo de seis componentes, inicializa cada casilla de ese arreglo con los valores enumerados entre los corchetes, y retorna la dirección del arreglo (que en este caso se almacena en la referencia *v*):

Figura 2: Un arreglo con seis componentes de tipo int.



Observar que si el arreglo tiene 6(seis) elementos, entonces la última casilla del mismo lleva el índice 5 debido a que los índices comienzan desde el 0. *En un arreglo en Python, no existe una casilla cuyo índice coincida con el tamaño del arreglo.* Lo mismo vale para cualquier secuencia de cualquier tipo (cadenas, tuplas, etc.)

Al igual que con las cadenas y las tuplas, el contenido completo de una variable de tipo *list* puede mostrarse directamente en la consola estándar mediante una simple invocación a la función *print()*. La siguiente secuencia de instrucciones:

```
print('Contenido del arreglo:', end=' ')
print(v)
```

producirá la siguiente salida:

```
Contenido del arreglo: [4, 7, 2, 8, 4, 3]
```

Como queda dicho, un arreglo en Python puede contener valores de tipos distintos, aunque luego el programador deberá esforzarse por evitar un procesamiento incorrecto. La

instrucción siguiente crea un arreglo *datos* con cuatro componentes, que serán respectivamente una *cadena*, un *int*, un *boolean* y un *float*:

```
datos = ['Juan', 35, True, 5678.56]
```

Una vez que se creó el arreglo, se usa la referencia que lo apunta para acceder a sus componentes, ya sea para consultarlos o para modificarlos, colocando a la derecha de ella un par de corchetes y el índice del casillero que se quiere acceder. Los siguientes son ejemplos de las operaciones que pueden hacerse con los componentes de una variable de tipo *list* en Python, recordando que un variable de tipo *list* es *mutable* y por lo tanto los valores de sus casilleros pueden cambiar de valor (tomamos como modelo el arreglo *v* anteriormente creado con valores de tipo *int*):

```
# asigna el valor 4 en la casilla 3
v[3] = 4

# suma 1 al valor contenido en la casilla 1
v[1] += 1

# muestra en consola el valor de la casilla 0
print(v[0])

# asigna en la casilla 4 el valor de la casilla 1 menos el de la 0
v[4] = v[1] - v[0]

# carga por teclado un entero y lo asigna en la casilla 5
v[5] = int(input('Valor: '))

# resta 8 al valor de la casilla 2
v[2] = v[2] - 8
```

Si se necesita crear un arreglo (variable de tipo *list*) con cierta cantidad de elementos iguales a un valor dado (por ejemplo, un arreglo con *n* elementos valiendo cero), el truco consiste en usar el operador de *multiplicación* y repetir *n* veces una *list* que solo contenga a ese valor [2]:

```
ceros = 15 * [0]
print('Arreglo de 15 ceros:', ceros)
# muestra: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

La función constructora *list()* también puede usarse para crear una variable de tipo *list* a partir de otras secuencias ya creadas. Por ejemplo, el siguiente bloque de instrucciones:

```
x = 'Mundo'
letras = list(x)
print(letras)
# muestra: ['M', 'u', 'n', 'd', 'o']
```

crea primero una variable *x* como cadena de caracteres (que es un tipo particular de secuencia inmutable en Python). Luego se usa la función *list()* tomando a la cadena *x* como parámetro para crear un arreglo (una variable de tipo *list*) llamado *letras* que contendrá en cada casillero a cada una de las letras de la cadena *x* original. Lo mismo puede hacerse a partir de cualquier tipo de secuencia de datos de Python, como se ve en el siguiente bloque simple:

```
t = 3, 4, 5, 6
print('Tupla:', t)

v = list(t)
print('Lista:', v)
```

El ejemplo anterior crea primero una tupla *t* con cuatro números e inmediatamente muestra su contenido. Como se trata de una tupla, los valores aparecerán en consola encerrados entre **paréntesis**. Luego se crea una variable de tipo *list* llamada *v*, usando la función *list()* y pasándole como parámetro la tupla *t*. La lista o arreglo *v* se crea de forma que contiene los mismos valores que la tupla *t* y luego se muestra en consola. Como se trata de una lista los valores aparecerán encerrados entre **corchetes**:

```
Tupla: (3, 4, 5, 6)
Lista: [3, 4, 5, 6]
```

Si el arreglo se creó vacío y luego se desea agregar valores al mismo, debe usarse el método *append* provisto por el propio tipo *list* (que en realidad es una *clase*). Por ejemplo, la siguiente secuencia crea un arreglo *notas* inicialmente vacío, y luego se usa *append()* para agregar en él las cuatro notas de un estudiante [2] [3]:

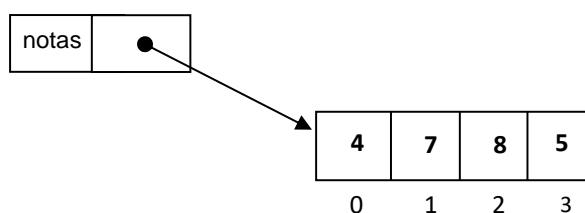
```
notas = []

notas.append(4)
notas.append(7)
notas.append(8)
notas.append(5)

print('Notas:', notas)
# muestra: Notas: [4, 7, 8, 5]
```

Note que el método *append()* agrega al contenido de la variable de tipo *list* el valor tomado como parámetro, pero lo hace de forma que ese valor se agrega *al final* del arreglo. En la secuencia anterior, el valor 4 quedará en la casilla 0 del arreglo, mientras que el 7, el 8 y el 5 quedarán en las casillas 1, 2 y 3 respectivamente, por orden de llegada:

**Figura 3: Efecto de la función *append()* al crear una variable de tipo *list*.**



Si se desea procesar un arreglo de forma que la misma operación se efectúe sobre cada uno de sus componentes, es normal usar un *ciclo for* de forma se aproveche la variable de control del ciclo como índice para entrar a cada casilla, realizando lo que se conoce como un *recorrido secuencial* del arreglo. El siguiente esquema muestra la forma de hacer la carga de cuatro valores por teclado para el mismo arreglo de notas que mostramos en el ejemplo anterior. El arreglo se muestra una vez cargado, y luego se recorre para obtener el promedio de las notas que contiene:

```

notas = []

for i in range(4):
    x = int(input('Nota: '))
    notas.append(x)

print('Notas:', notas)

suma = 0
for i in range(4):
    suma += notas[i]
promedio = suma / 4

print('Promedio:', promedio)

```

Recuerde que la invocación `range(4)` crea un rango o intervalo de cuatro elementos con los valores [0, 1, 2, 3] (comenzando en 0 y sin incluir al 4). Esto no es importante en el ciclo de carga del arreglo, pero sí lo es en el ciclo que acumula sus elementos para luego calcular el promedio, ya que el valor de la variable *i* se está usando como índice para entrar a cada casilla de la variable *notas*.

Siempre se puede usar la función `len()` para saber el tamaño o cantidad de elementos que tiene una variable de tipo *list*. La secuencia que calcula el promedio en el ejemplo anterior podría re-escribirse así:

```

suma = 0
n = len(notas)
for i in range(n):
    suma += notas[i]

promedio = suma / n

```

En Python también se puede definir una variable de tipo *list* empleando una técnica conocida como *definición por comprensión*, que dejaremos para ser analizada en la sección 5 de esta misma Ficha.

### 3.] Acceso a componentes individuales de una variable de tipo list.

El proyecto *[F12] Arreglos* dentro de la carpeta *Fuentes* que acompaña a esta Ficha, contiene algunos modelos desarrollados con *PyCharm* a modo de ejemplo de cada uno de los temas que analizaremos en esta sección.

Vimos que el tipo *list* se usa en Python para representar lo que en general se conoce como un arreglo en el campo de la programación. En muchos lenguajes de programación un arreglo es una estructura de *tamaño fijo*: esto significa que una vez creado, no se pueden agregar ni quitar casillas al arreglo, lo cual hace que los programadores deban seguir ciertas pautas para evitar un uso indebido de la memoria.

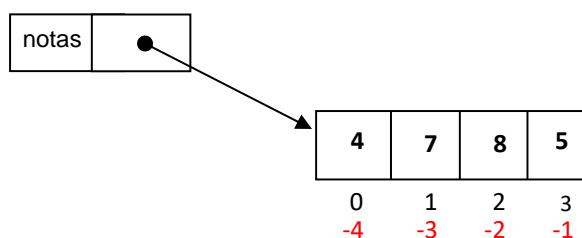
En ese sentido, un detalle a favor de Python es que *las variables de tipo list que se usan para representar arreglos no son de tamaño fijo*, sino que pueden aumentar o disminuir en forma dinámica su tamaño, agregar o quitar elementos en cualquier posición o incluso cambiar el tipo de sus componentes (como era de esperar en un lenguaje de tipado dinámico).

Al igual que para todos los tipos de secuencias, los elementos individuales de una variable de tipo *list* pueden accederse mediante índices. El primer elemento de una lista lleva el índice

cero, y los demás siguen en forma consecutiva. Pero además de esto, en Python se pueden usar **índices negativos** con cualquier tipo de secuencia (listas, tuplas, cadenas, etc.) sabiendo que el índice -1 corresponde al último elemento, el -2 al anteúltimo, y así sucesivamente [2].

Esto es: se puede asumir que en Python, cada casillero de cualquier tipo de *secuencia de datos* está identificado por dos números índices que pueden ser usados indistintamente por el programador, según lo crea conveniente: uno de esos índices es un número mayor o igual a 0 de forma que el 0 identifica a la primera casilla de la izquierda, y el otro es un número negativo menor o igual a -1 de forma que el -1 identifica *siempre* a la última casilla de la derecha. El arreglo *notas* que hemos mostrado como ejemplo más arriba, puede entenderse entonces como se ve en la gráfica siguiente:

**Figura 4: Esquema de índices para una secuencia de cualquier tipo en Python.**



El programa *test01.py* del citado proyecto [F12] *Arreglos* incluye una función general llamada *lists\_test()* en la cual se crea una lista de números del 1 al *n* y luego se la recorre desde ambos extremos con ciclos *for* iterando sobre sus índices, para realizar algunas operaciones simples (*incluyendo un recorrido de derecha a izquierda usando índices negativos*):

```
# crear una lista con n numeros del 1 al n...
n = 10
numeros = []
for i in range(1, n+1):
    numeros.append(i)

print("Lista original: ", numeros)

# recorrer de izquierda a derecha y mostrar los numeros pares...
print("Valores pares contenidos en la lista: ", end=" ")
for i in range(len(numeros)):
    if numeros[i] % 2 == 0:
        print(numeros[i], end=" ")

# recorrer de derecha a izquierda y mostrar todos los numeros...
print("\nContenido de la lista, en forma invertida: ", end=" ")
for i in range(-1, (-len(numeros) - 1), -1):
    print(numeros[i], end=" ")
```

En Python, las variables de tipo *list* son de naturaleza *mutable*. Esto quiere decir que se puede cambiar el valor de un elemento sin tener que crear una *list* nueva. Lo siguiente, cambia el valor del elemento en la posición 2 por el valor -1:

```
numeros = [3, 5, 8, 6]
numeros[2] = -1
print(numeros)
# muestra: [3, 5, -1, 6]
```

Si se desea eliminar un elemento en particular, se puede recurrir a la instrucción ***del***:

```
numeros = [3, 5, 8, 6, 9, 2]
del numeros[3]
print(numeros) # muestra: [3, 5, 8, 9, 2]
```

Note que la instrucción ***del*** puede usarse para remover completamente toda la lista o arreglo, pero eso hace también que la variable que referenciaba al arreglo **quede indefinida de allí en más**. Una secuencia de instrucciones de la forma siguiente, provocará un error de variable no inicializada cuando se intente mostrar el arreglo:

```
numeros = [1, 2, 3]
del numeros
print(numeros) # error: variable no inicializada...
```

Todo tipo de secuencia en Python permite acceder a un subrango de casilleros "**rebanando**" o "**cortando sus índices**". La siguiente instrucción crea una variable de tipo *list nueva* en la que se copian los elementos de los casilleros 1, 2 y 3 del arreglo original *numeros* [2]:

```
numeros = [4, 2, 3, 4, 7]
nueva = numeros[1:4]
print("\nNueva lista: ", nueva)
# muestra: [2, 3, 4]
```

La instrucción siguiente también usa un **corte de índices** pero esta vez para copiar toda una lista:

```
numeros = [2, 4, 6, 5]
copia = numeros[:] # copia toda la lista
print(copia) # muestra: [2, 4, 6, 5]
```

Llegados a este punto, es muy importante notar la diferencia entre asignar entre sí dos variables de tipo *list*, o usar **corte de índices** para copiar un arreglo completo. Si la variable *numeros* representa un arreglo, y se asigna en la variable *copia1*, entonces **ambas variables quedarán apuntando al mismo arreglo** (y por lo tanto, cualquier cambio en el contenido de uno afectará al contenido del otro):

```
numeros = [3, 5, 8, 3]

# una copia a nivel de referencias (copia superficial)...
copial = numeros # ambas variables apuntan a la misma lista...

numeros[2] = -1
print("Original: ", numeros)
print("Copia 1: ", copial)
```

El script anterior produce la siguiente salida:

```
Original: [3, 5, -1, 3]
Copia 1: [3, 5, -1, 3]
```

Sin embargo, cuando se usa el operador de **corte de índices** se está recuperando **una copia** de los elementos de la sublista pedida. Por lo tanto, el siguiente script **no copia** las referencias, **sino que crea un segundo arreglo**, cuyos elementos son iguales a los del primero (y cualquier modificación que se haga en una de las listas, no afectará a la otra):

```

numeros = [3, 5, 8, 3]

# una copia a nivel valores...
copia2 = numeros[:] # cada variable es una lista diferente...

numeros[2] = -1
print("Original: ", numeros)
print("Copia 2: ", copia2)

```

El script anterior produce la siguiente salida:

```

Original: [3, 5, -1, 3]
Copia 2: [3, 5, 8, 3]

```

El acceso a subrangos de elementos mediante el corte de índices permite operaciones de consulta y/o modificación de sublistas en forma muy dinámica y flexible. En el programa *test01.py* se incluyen las siguientes secuencias de instrucciones que son en sí mismas muy ilustrativas. Los comentarios agregados en el código fuente ayudan a aclarar las ideas [2]:

```

# operaciones varias con cortes de indices...
lis = [47, 17, 14, 41, 37, 74, 48]
print('\nLista de prueba:', lis)

# reemplazar elementos en un rango...
lis[2:5] = [10, 20, 30]
print('Lista modificada 1:', lis)

# eliminar elementos en un rango...
lis[1:4] = []
print('Lista modificada 2:', lis)

# insertar un elemento... en este caso en la posición 2...
lis[2:2] = [90]
print('Lista modificada 3:', lis)

# inserta una copia de si misma al principio...
lis[:0] = lis
print('Lista modificada 4:', lis)

# agregar un elemento al final, mediante un corte...
lis[len(lis):] = [100]
print('Lista modificada 5:', lis)

# agregar un elemento al final...
lis.append(200)
print('Lista modificada 6:', lis)

# copiar desde el casillero 2 hasta el ante-último...
l1 = lis[2:-1]
print('Sub-lista 1:', l1)

# copiar dos veces el subrango entre 2 y 4...
l2 = 2 * lis[2:5]
print('Sub-lista 2:', l2)

# concatenar dos listas...
l3 = lis[:4] + [1, 2, 3]
print('Sub-lista 3:', l3)

```

```
# vaciar una lista...
lis[:] = []
print('Lista original vacía:', lis)
```

En el modelo anterior se puede ver que usando cortes de índices es posible realizar casi cualquier operación de consulta o modificación. Observe que una instrucción como:

```
l2 = 2 * lis[2:5]
```

recupera desde el arreglo *lis* la sublista que incluye a los elementos desde la posición 2 hasta la posición 4, replica 2 veces esa sublista (debido al uso del operador *producto*) y asigna la sublista replicada en la variable *l2*.

Si se asigna *una lista* en un subrango de índices de otra lista, entonces la sublista que corresponde al subrango original será *reemplazada por la lista asignada*, aún si los tamaños no coinciden. Las instrucciones:

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20, 30]
print(lis)
# muestra: [47, 17, 10, 20, 30, 74, 48]
```

hace que los elementos que originalmente tenía el arreglo *lis* en los casilleros 2, 3 y 4 se reemplacen por los valores 10, 20 y 30 respectivamente. Si la lista asignada es de distinto tamaño que el subrango accedido, el reemplazo se lleva a cabo de todas formas, y la lista original puede ganar o perder elementos. En el siguiente ejemplo, un subrango de tres elementos es reemplazado por una lista de dos elementos (y el arreglo original queda con un elemento menos):

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20]
print(lis)
# muestra: [47, 17, 10, 20, 74, 48]
```

Y en el siguiente ejemplo, un corte de tres elementos es reemplazado por una lista de cuatro valores, y obviamente el arreglo original pasa a tener un elemento más:

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20, 30, 40]
print(lis)
# muestra: [47, 17, 10, 20, 30, 40, 74, 48]
```

Note también que si en un corte de índices el primer valor *i1* es mayor o igual al segundo *i2* (pero *i2* no es negativo), entonces el subrango accedido incluye solo *una lista vacía* en la posición *i1*. Por eso la instrucción:

```
lis[2:2] = [90]
```

en la práctica *agrega* el valor 90 exactamente en la posición 2 de la lista, pero no reemplaza al elemento en la posición 2 (en estricto rigor de verdad, *reemplaza la lista vacía* que "comienza en la posición 2 y termina en la 1", por la lista que contiene al 90). En ese sentido, sería exactamente lo mismo escribir

```
lis[2:1] = [90]    # o también lis[2:0] = [90]
```

Se deduce de lo anterior, que si quiere agregar un elemento (o una lista de varios elementos) *al final de la lista*, la siguiente instrucción permite hacerlo:

```
lis[len(lis):] = [100]
```

El corte usado en el ejemplo anterior comienza en la posición que coincide con el tamaño de la lista, y desde allí hasta el final. Como no existe una lista en esa posición (pero la posición en sí misma es válida) se considera que la sublista accedida es la *lista vacía* que se encuentra exactamente al final de la lista original. Y esa lista se reemplaza por la lista [100].

Es útil saber que en realidad lo que están haciendo las instrucciones vistas es asignar en el subrango pedido los elementos del arreglo que aparece en el lado derecho de la asignación, el cual es recorrido (iterado) de forma que cada componente se inserta en el arreglo de la izquierda. Sería un error asignar directamente un número, ya que en este caso no se tendría una secuencia iterable:

```
lis[2:2] = 90 # error!!!
```

Si el primer índice de un corte es positivo y el segundo es negativo, recuerde que el índice negativo *-1* accede al último elemento del arreglo, y por lo tanto, un corte de este tipo *no está reemplazando una lista vacía*, sino que reemplaza a un *subrango no vacío* perfectamente definido [2] [3]:

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:-1] = [10, 20]
print(lis)
# muestra: [47, 17, 10, 20, 48]
```

En el ejemplo anterior, el corte *lis[2 : -1]* está accediendo al subrango que comienza en el índice 2 (el 14 en el arreglo) y llega hasta el índice -1 pero sin incluir a éste. Como el índice -1 equivale al último casillero, entonces el corte llega hasta el anteúltimo (que en la lista sería el 74). Y todos esos elementos (marcados en rojo en la lista original del ejemplo) se reemplazan por la lista [10, 20].

#### 4.] Aplicaciones y casos de análisis.

En esta sección se analizan y resuelven varios problemas para poner en práctica lo visto hasta ahora sobre *arreglos unidimensionales* [1].

**Problema 34.)** *Cargar por teclado un arreglo de n componentes y multiplicarlo por el valor k que también se ingresa por teclado.*

**Discusión y solución:** La lógica del programa a desarrollar es directa: En la función *test()* (que será la función de arranque del programa) primero se debe cargar por teclado el valor de *n*, que será usado para indicar cuántos casilleros tendrá el vector o arreglo (al hacer la carga se debe validar que el valor cargado no sea cero ni negativo). Después se puede crear un arreglo *v* que contenga ya *n* casillas inicializadas en 0 (con lo cual la carga del vector puede hacerse después accediendo a cada casilla por su índice, sin necesidad de recurrir a la función *append()* puesto que esas casillas ya existen en el arreglo)

Luego se puede desarrollar una función *read()* que cargue por teclado el arreglo, usando un ciclo *for* ajustado de forma que la variable de control *i* varíe desde 0 hasta *n – 1*. En cada

vuelta del ciclo se debe cargar el componente  $v[i]$  con lo que, al finalizar el ciclo, el arreglo quedará cargado completamente. Esta función está incluida en el módulo *arreglos.py* dentro del proyecto [F12] Arreglos y es la siguiente:

```
def read(v):
    # carga por teclado de un vector de n componentes...
    n = len(v)
    print('Cargue los elementos del vector:')
    for i in range(n):
        v[i] = int(input('casilla[' + str(i) + ']: '))
```

Finalmente podemos definir una función *product()* que también recorra el arreglo usando un *for*, pero en cada vuelta multiplique en forma acumulativa el componente  $v[i]$  por el valor  $k$ : el valor contenido en la casilla  $v[i]$  se multiplica por  $k$  y el resultado se vuelve a almacenar en la misma casilla  $v[i]$ . En el proyecto [F12] Arreglos que acompaña a esta Ficha, la función *product()* está incluida dentro del módulo *arreglos.py* y es la siguiente:

```
# función incluida en el módulo arreglos.py
def product(v, k):
    # multiplica por k al arreglo y lo retorna...
    n = len(v)
    for i in range(n):
        v[i] *= k
```

La solución a este problema se muestra a continuación (corresponde al modelo *test02.py* del proyecto [F12] Arreglos que acompaña a esta ficha de clase):

```
__author__ = 'Cátedra de AED'

import arreglos

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (>' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')

    return n

def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # cargar el arreglo por teclado...
    arreglos.read(v)

    # cargar el número k por el cual se multiplicará al vector...
    k = int(input('Ingrese el valor de k: '))

    # multiplicar el vector por k...
    arreglos.product(v, k)

    # mostrar el vector final...
    print('El vector quedó así:', v)
```

```
# script principal...
if __name__ == '__main__':
    test()
```

Un ejemplo de pantalla de salida para este programa es el que sigue (suponiendo que se cargarán  $n = 4$  elementos en el vector y que el valor  $k$  será  $k = 3$ ):

```
Ingrese cantidad de elementos (mayor a 0 por favor): 4
Cargue los elementos del vector:
casilla[0]: 3
casilla[1]: 4
casilla[2]: 5
casilla[3]: 6
Ingrese el valor de k: 3
El vector quedó así: [9, 12, 15, 18]
```

Veamos ahora otro problema sencillo:

**Problema 35.) Cargar por teclado dos vectores de tamaño  $n$  y obtener y mostrar el vector suma.**

**Discusión y solución:** La suma de dos vectores  $a$  y  $b$  de longitud  $n$  es otro vector  $c$  de longitud  $n$  en el cual cada componente es igual a la suma de las componentes con el mismo índice (u *homólogas*) en los vectores  $a$  y  $b$ . La función *add()* (que hemos incluido en el módulo *arreglos.py* del mismo proyecto [F12] Arreglos) efectúa la suma de  $a$  y  $b$  usando un ciclo *for*. En cada vuelta, toma el componente  $a[i]$ , lo suma con el componente  $b[i]$ , y guarda el resultado en el componente  $c[i]$ . Al finalizar el ciclo, el vector suma está completo y se lo devuelve mediante la instrucción *return*:

```
def add(a, b):
    # suma los arreglos a y b, y retorna el arreglo suma...
    n = len(a)
    c = n * [0]
    for i in range(n):
        c[i] = a[i] + b[i]

    return c
```

Aquí también usaremos la función *read()* del módulo *arreglos.py* que ahora será invocada dos veces para hacer la carga por teclado los arreglos  $a$  y  $b$ . Note cómo el uso de parámetros permite que una misma y única función pueda ser invocada tantas veces como sea necesario para procesar variables distintas. La función tiene un *parámetro formal* de tipo arreglo llamado  $v$ , el cual se equipara con cualquiera de los parámetros actuales ( $a$  o  $b$ ) enviados desde *test()* y finalmente se hace la carga correcta. El programa completo es el siguiente, que corresponde al modelo *test03.py* del proyecto [F12] Arreglos:

```
__author__ = 'Cátedra de AED'

import arreglos

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
```

```

    return n

def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear dos arreglos de n elementos (valor inicial 0)...
    a = n * [0]
    b = n * [0]

    # cargar los dos arreglos por teclado...
    print('\nVector a:')
    arreglos.read(a)
    print('\nVector b:')
    arreglos.read(b)

    # obtener el vector suma...
    c = arreglos.add(a, b)

    # mostrar el vector suma...
    print('\nEl vector suma es:', c)

# script principal...
if __name__ == '__main__':
    test()

```

Un ejemplo de ejecución y salida por pantalla de este programa se muestra a continuación:

```

Ingrese cantidad de elementos (mayor a 0 por favor): 3

Vector a:
Cargue los elementos del vector:
casilla[0]: 1
casilla[1]: 2
casilla[2]: 3

Vector b:
Cargue los elementos del vector:
casilla[0]: 4
casilla[1]: 5
casilla[2]: 6

El vector suma es: [5, 7, 9]

```

Analicemos ahora un último problema:

**Problema 36.) Cargar por teclado dos vectores de tamaño  $n$  y obtener el producto escalar de ambos.**

**Discusión y solución:** Recordemos que el *producto escalar* (o *producto interno* o también *producto punto*) entre dos vectores de longitud  $n$ , es un *número* (y *no* otro vector) que se calcula acumulando los productos de las componentes del mismo índice (u *homólogas*) de los dos vectores que se multiplican. El valor final de ese acumulador es el producto escalar entre los vectores.

En el programa que sigue, el producto escalar es calculado por la función *scalar\_product()* incluida en el mismo módulo *arreglos.py* del proyecto [F12] *Arreglos*. Dicha función pone en cero el acumulador *sp*, y con un ciclo *for* efectúa los productos del tipo *a[i] \* b[i]*. Cada uno

de esos productos se acumula en *sp*, y al terminar el ciclo se retorna el valor final contenido en ese acumulador:

```
# función incluida en el módulo arreglos.py
def scalar_product(a, b):
    # calcula el producto escalar entre a y b...
    n = len(a)

    sp = 0
    for i in range(n):
        sp += a[i]*b[i]

    return sp
```

Mostramos aquí el programa completo (ver modelo *test04.py* en el proyecto [F12] Arreglos que acompaña a esta Ficha):

```
import arreglos

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('cantidad de elementos (> ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')

    return n

def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear dos arreglos de n elementos (valor inicial 0)...
    a = n * [0]
    b = n * [0]

    # cargar los dos arreglos por teclado...
    print('\nVector a:')
    arreglos.read(a)

    print('\nVector b:')
    arreglos.read(b)

    # obtener el producto escalar...
    pe = arreglos.scalar_product(a, b)

    # mostrar el producto escalar...
    print('\nEl producto escalar es:', pe)

# script principal...
if __name__ == '__main__':
    test()
```

## 5.] Definición por comprensión de variables de tipo list.

Hemos mostrado en la sección 2 de esta misma Ficha diversas formas de creación de variables de tipo *list* para representar arreglos unidimensionales en Python. Analizaremos ahora otra forma de creación de listas que consiste en *definir el contenido de la lista por*

*comprensión*, recurriendo a alguna forma de iterador [2]. En el ejemplo siguiente (ver modelo *test05.py* del proyecto [F12] Arreglos que acompaña a esta Ficha), cada una de las letras de la cadena "Hola" es tomada por el ciclo iterador *for* y almacenada en la lista *letras* que se está creando:

```
# una cadena...
cadena = 'Hola'

# una lista creada por comprensión...
letras = [c for c in cadena]

print(letras)
# muestra: ['H', 'o', 'l', 'a']
```

Técnicamente, la variable *cadena* del ejemplo anterior puede ser de cualquier tipo *que admita un recorrido con iterador* (una cadena, una tupla, otra lista, etc.):

```
# una tupla...
secuencia = 1, 4, 7, 2

# una lista creada por comprensión a partir de la tupla...
numeros = [num for num in secuencia]

print(numeros)
# muestra: [1, 4, 7, 2]
```

El mismo arreglo *numeros* conteniendo los números del 1 al 10 que hemos creado en la sección 3 de esta Ficha mediante la siguiente secuencia de instrucciones:

```
# crear una lista con n numeros del 1 al n...
n = 10
numeros = []
for i in range(1, n+1):
    numeros.append(i)

print("Lista original: ", numeros)
```

*podría equivalentemente ser creado mediante comprensión* en la forma que mostramos a continuación:

```
# crear un arreglo con n numeros del 1 al n...
n = 10
numeros = [i for i in range(1, n+1)]
print('Lista original:', numeros)
```

La *creación de listas por comprensión* es una técnica sumamente flexible en Python. Sintácticamente, consiste en escribir dentro de un par de corchetes una expresión (que puede ser no solamente una variable...) seguida de un *for* que a su vez puede contener *otros ciclos for* y/o instrucciones condicionales *if...* El resultado será una nueva lista en la que cada uno de sus elementos será a su vez el resultado de cada una de las iteraciones del *for* sobre la expresión inicial.

Un ejemplo más complejo aclarará el panorama: supongamos que se desea crear una lista en la que cada elemento sea el cubo de los primeros 10 números naturales. Una forma de hacerlo (*sin* aplicar *creación por comprensión*) sería:

```
cubos = []
for i in range(1, 11):
    cubos.append(i**3)
```

Pero aplicando *creación por comprensión*, lo anterior es exactamente lo mismo que:

```
cubos = [i**3 for i in range(1, 11)]
print('Lista de cubos:', cubos)
# muestra: [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Como sabemos, en el modelo anterior la expresión  $i^{**3}$  equivale a elevar al cubo el valor de  $i$ , y a su vez el valor de  $i$  es aportado por el *for* en cada iteración sobre el rango de valores  $[1..10]$ .

Expresiones más complejas son también posibles. Supongamos que se tienen dos listas de números  $v1$  y  $v2$  y queremos generar una tercera lista que contenga tuplas o pares  $(x, y)$ , tales que  $x$  esté en  $v1$ ,  $y$  esté en  $v2$  pero  $x$  sea menor que  $y$ . La forma básica de hacerlo *sin* aplicar comprensión sería:

```
v1 = [7, 3, 4, 8]
v2 = [1, 3, 5, 9]

pares = []
for x in v1:
    for y in v2:
        if x < y :
            pares.append((x, y))

print('Pares formados:', pares)
# muestra: [(7, 9), (3, 5), (3, 9), (4, 5), (4, 9), (8, 9)]
```

Pero *aplicando comprensión*, la creación de la lista se resume a:

```
v1 = [7, 3, 4, 8]
v2 = [1, 3, 5, 9]

pares = [(x, y) for x in v1 for y in v2 if x < y]
print('Pares formados:', pares)
# muestra: [(7, 9), (3, 5), (3, 9), (4, 5), (4, 9), (8, 9)]
```

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 13

## Arreglos: Algoritmos y Técnicas Básicas

### 1.] Ordenamiento de un arreglo unidimensional (Algoritmo: Selección Directa).

Una operación muy común cuando se trabaja con arreglos unidimensionales es la de *ordenar* sus elementos, ya sea en secuencia de menor a mayor o de mayor a menor. Entre otras ventajas, la importancia del ordenamiento de un arreglo (y en general, de cualquier estructura de datos que admita algún tipo de ordenamiento) está en relación directa con la operación de *buscar* valores dentro de ella. Como se verá, pueden plantearse algoritmos que resultan más eficientes para buscar un valor en un arreglo *ordenado* que para buscarlo en uno desordenado.

Existen muchísimos algoritmos diferentes para proceder al ordenamiento de un arreglo de  $n$  componentes<sup>1</sup>. El estudio de todos y cada uno de ellos, sería una tarea ajena al propósito introductorio de esta Ficha (que está orientada a técnicas básicas, aunque analizaremos muchos de esos algoritmos en una Ficha posterior), y por lo tanto nos concentraremos en comprender el modo de funcionamiento de sólo uno de ellos, de naturaleza sencilla, conocido como *Ordenamiento de Selección Directa*. No obstante, aclaramos que incluso este sencillo método ofrece variantes y mejoras que no serán analizadas aquí.

Si se desea ordenar el arreglo de menor a mayor usando *selección directa*, la idea central es la adaptación y repetición sistemática del simple algoritmo que mostramos a continuación (suponemos que el arreglo a ordenar es  $v$  y que ya está cargado con valores de tipo *int*) [1]:

```
i = 0
for j in range(i+1, n):
    if v[i] > v[j]:
        v[i], v[j] = v[j], v[i]
```

Un rápido análisis muestra que el algoritmo anterior recorre el arreglo buscando el valor *menor* del mismo, y al terminar lo deja en la casilla indicada por la variable  $i$  (que en este caso vale cero). Se comienza asumiendo que el menor está en la casilla cuyo índice es  $i = 0$ , y

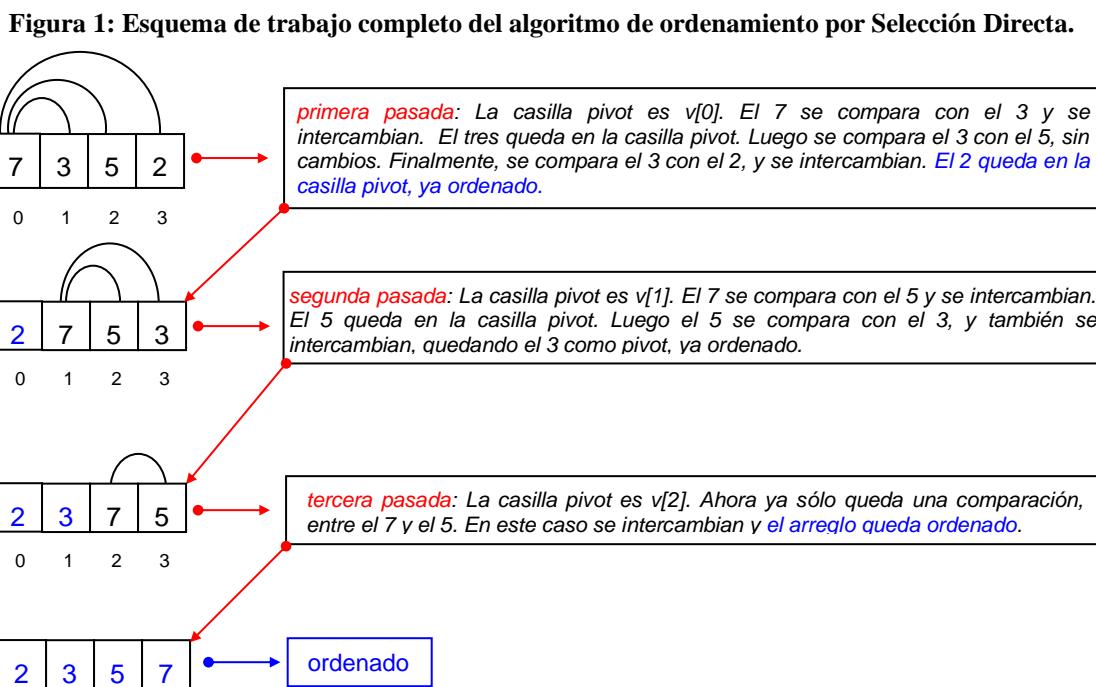
<sup>1</sup> Si se trata de mundos ordenados, no se puede menos que citar la película *Divergent* (en español conocida como *Divergente*) del año 2014, dirigida por *Neil Burger* y protagonizada por *Shailene Woodley*. A su vez, está basada en la primera parte de la trilogía de novelas juveniles escrita por *Veronica Roth*. El mundo ha sido devastado por la guerra y la sinrazón, y los sobrevivientes se reorganizaron en una sociedad estrictamente ordenada y dividida en clanes o facciones. Cada persona es asignada a una de esas facciones de acuerdo a sus aptitudes, y durante toda su vida contribuye al desarrollo de la sociedad desde esa facción. La protagonista Tris Prior descubre que no encaja en ninguna de las facciones (y por eso es "divergente") pero ese descubrimiento es peligroso porque altera el orden pre-establecido... En 2015 llegó al cine la segunda parte conocida como *The Divergent Series: Insurgent* (o *La Serie Divergente: Insurgente*) dirigida por *Robert Schwentke*. Y como era de esperar, la tercera parte se dividió en dos películas: *The Divergent Series: Allegiant* (*La Serie Divergente: Leal*) estrenada en 2016 y otra vez dirigida por *Robert Schwentke*, y *The Divergent Series: Ascendant* (*La Serie Divergente: Ascendente*) con la dirección anunciada de *Lee Toland Krieger* (agendada para 2017, pero aún en espera... será desarrollada como película para televisión debido al bajo nivel de taquilla de *Allegiant*).

se recorre el resto del arreglo con  $j$  comenzando desde el valor  $i+1$ . Cada valor en la casilla  $j$  se compara con el valor en la casilla  $i$ . Si el valor en  $v[i]$  resulta mayor que  $v[j]$  entonces los valores se intercambian, y así se prosigue hasta que el ciclo controlado por  $j$  termina.

El algoritmo presentado *no ordena* el arreglo: sólo garantiza que el menor valor será colocado en la casilla con índice  $i = 0$ . Pero entonces sólo basta un pequeño cambio para que se ordene **todo** el arreglo: hacer que la variable  $i$  modifique su valor con *otro* ciclo, comenzando desde cero y terminando antes de llegar a la última casilla (para evitar que  $j$  comience valiendo un índice fuera de rango). La función `selection_sort()` que sigue (incluida en el módulo `arreglos.py` del proyecto [F13] Ordenamiento y Búsqueda que acompaña a esta Ficha) hace justamente eso:

```
def selection_sort(v):
    # ordenamiento por seleccion directa
    n = len(v)
    for i in range(n-1):
        for j in range(i+1, n):
            if v[i] > v[j]:
                v[i], v[j] = v[j], v[i]
```

Gráficamente, el proceso completo puede mostrarse así:



El mismo proyecto [F13] Ordenamiento y Búsqueda incluye el programa `test01.py`, en el cual se carga por teclado y se ordena un arreglo usando esta función:

```
__author__ = 'Cátedra de AED'

import arreglos

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
```

```

    return n

def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # cargar el arreglo por teclado...
    print('\nVector v:')
    arreglos.read(v)

    # ordenar el arreglo...
    c = arreglos.selection_sort(v)

    # mostrar el vector ordenado...
    print('\nEl vector ordenado es:', v)

# script principal...
if __name__ == '__main__':
    test()

```

## 2.] Búsqueda secuencial.

Muchas veces se presente el problema de determinar si cierto valor  $x$  está contenido o no en un arreglo  $v$  de  $n$  componentes. La convención suele ser que si se encuentra se informa en qué posición, y si no se encuentra se informa con un mensaje.

Si el arreglo está desordenado, o no se sabe nada acerca de su estado, la única forma inmediata de buscar un valor en él consiste en hacer una **búsqueda secuencial**: con un ciclo `for` se comienza con el primer elemento del arreglo. Si allí se encuentra el valor buscado, se detiene el proceso y se retorna el índice del componente que contenía al valor. Si allí no se encuentra el valor, se salta al componente siguiente y se repite el esquema. Si se llega al final del arreglo sin encontrar el valor buscado, lo cual ocurrirá cuando  $i$  (la variable de control del ciclo) sea igual al tamaño del arreglo, la función de búsqueda retorna el valor  $-1$ , a modo de indicador para avisar que la búsqueda falló [1]. El algoritmo de **búsqueda secuencial** en un arreglo se aplica en la función `linear_search()` que sigue (incluida en el módulo `arreglos.py` del proyecto [F13] Ordenamiento y Búsqueda que acompaña a esta Ficha):

```

def linear_search(v, x):
    # búsqueda secuencial...
    for i in range(len(v)):
        if x == v[i]:
            return i

    return -1

```

El programa `test02.py` (en el mismo proyecto [F13] Ordenamiento y Búsqueda), carga un arreglo por teclado y procede luego a buscar un valor  $x$  en el mismo por medio de la función anterior:

```

__author__ = 'Cátedra de AED'

import arreglos

```

```

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # cargar el arreglo por teclado...
    print('\nVector v:')
    arreglos.read(v)

    x = int(input('Valor a buscar en el arreglo: '))

    # aplicar búsqueda secuencial...
    ind = arreglos.linear_search(v, x)

    # avisar por pantalla el resultado de la búsqueda...
    if ind >= 0:
        print('\nEstá en la casilla', ind)
    else:
        print('\nNo está en el arreglo')

# script principal...
if __name__ == '__main__':
    test()

```

### 3.] Búsqueda binaria.

Si el arreglo estuviera ordenado (por ejemplo de menor a mayor), se puede aplicar un método de búsqueda mucho más eficiente (en el sentido de su velocidad de ejecución) que la búsqueda secuencial ya vista, conocido como *Búsqueda Binaria*. La idea básica es la siguiente [1]: se usan dos índices auxiliares *izq* y *der* cuya función es la de marcar dentro del arreglo los límites del intervalo en donde se buscará el valor. Como al principio la búsqueda se hace en *todo* el vector, originalmente *izq* comienza valiendo 0 y *der* comienza valiendo *n*-1 (siendo *n* la cantidad de componentes del arreglo). Dentro del intervalo marcado por *izq* y *der*, se toma el elemento central, cuyo índice *c* es:

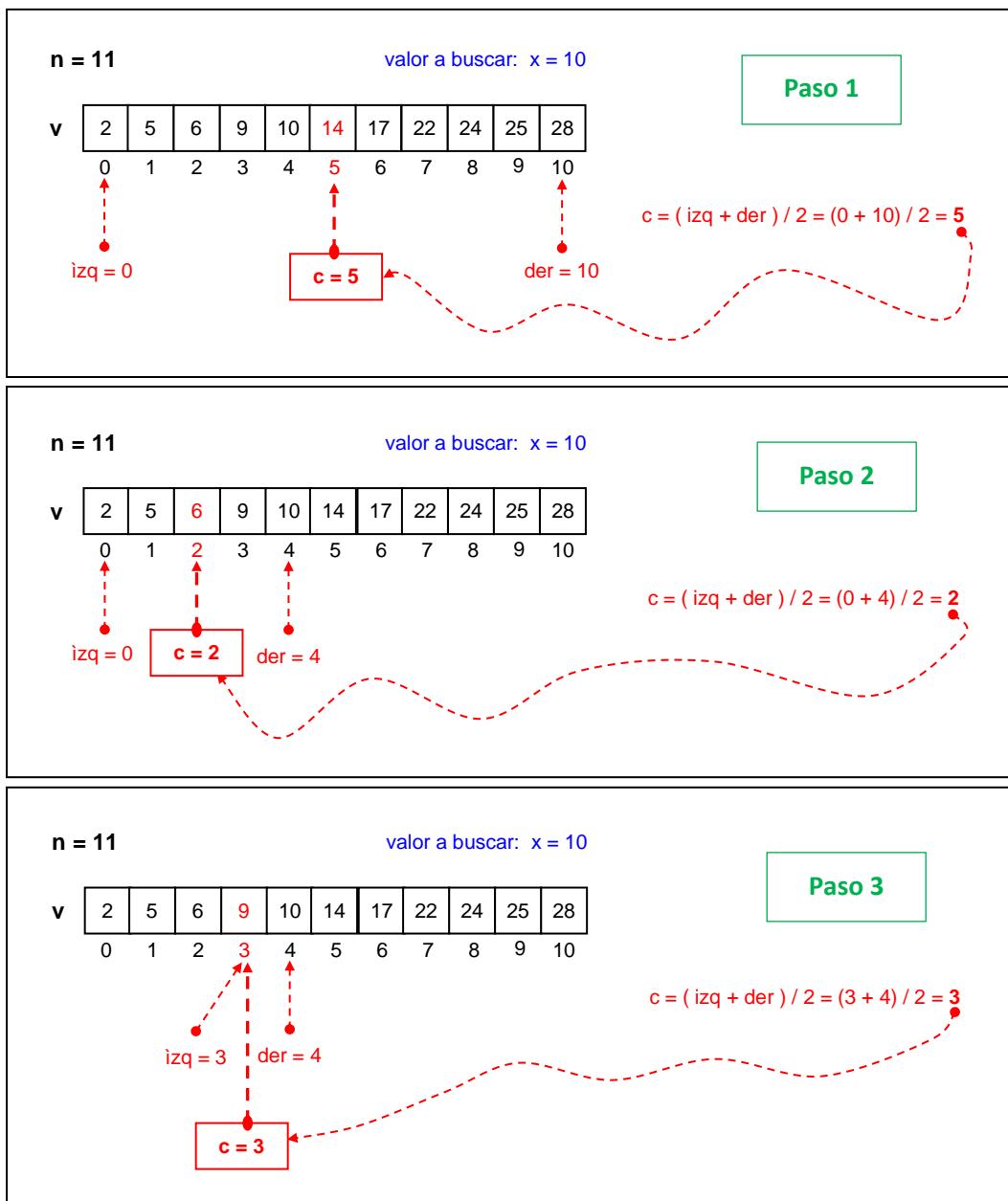
$$c = (izq + der) // 2$$

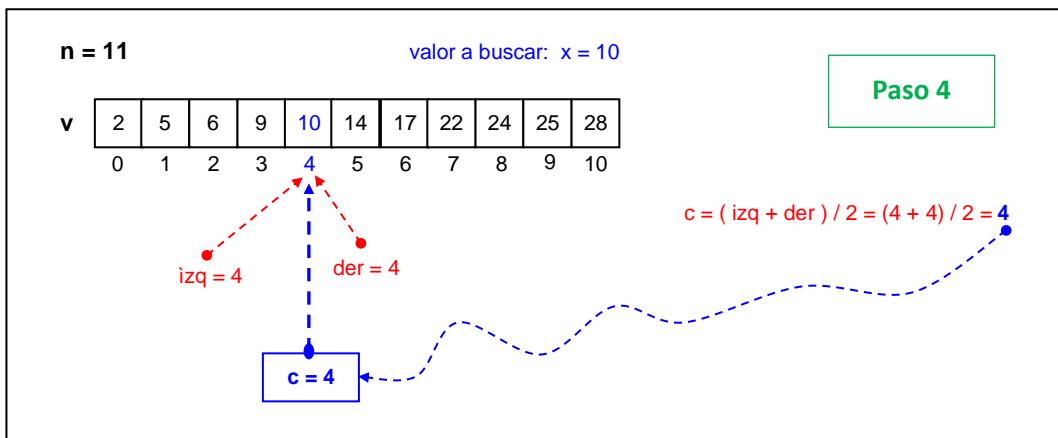
o sea, el promedio de los valores de *izq* y *der*. Luego de esto, se verifica si el valor contenido en *v[c]* coincide o no con el número buscado. Si coincide, se termina la búsqueda, y se retorna el valor de *c* para indicar la posición del número dentro del arreglo. Si no coincide, entonces cuando se aprovecha que el arreglo está ordenado de menor a mayor: si el valor buscado *x* es menor que *v[c]*, entonces si *x* está en el arreglo debe estar a la izquierda de *v[c]* y por lo tanto, el nuevo intervalo de búsqueda debe ir desde el valor de *izq* hasta el valor de *c-1*. Entonces, se ajusta *der* para que valga el valor *c-1*, y se repite el proceso descripto. Si el valor *x* es mayor que *v[c]*, entonces la situación es la misma pero simétrica hacia la derecha, y debe ajustarse *izq* para valer *c+1*.

El proceso continúa hasta que se encuentre el valor (en cuyo caso se retorna el último valor c calculado), o hasta que el valor de *izq* se haga mayor que el de *der* (es decir, hasta que los índices se crucen), lo cual indicará que ya no quedan intervalos en los que buscar, y por lo tanto el valor x no estaba en el arreglo (y en este caso, la función que hace la búsqueda retornará el valor -1).

El proceso que se describió se denomina *búsqueda binaria* porque parte al arreglo en dos intervalos a partir del valor central, y a cada intervalo en otros dos, hasta dar con el valor o no poder generar nuevos intervalos. La eficiencia del método se basa en que se toma sólo uno de los dos intervalos para buscar al valor, y el otro se desecha por completo. En la primera partición, la mitad de los elementos del arreglo se desechan, en la segunda se desecha la cuarta parte (o sea, la mitad de la mitad), y así sucesivamente. Con muy pocas comparaciones, el valor es encontrado (si existe). La siguiente secuencia de gráficos ilustra el proceso completo:

Figura 2: Esquema de trabajo completo del algoritmo de Búsqueda Binaria.





La desventaja obvia es que el arreglo *debe estar ordenado*. Si se trabaja en un contexto donde el arreglo sufre cambios permanentes de contenido y debe ser ordenado periódicamente para facilitar la búsqueda, entonces se pierde la ventaja de la rapidez del algoritmo, pues se supera largamente con el tiempo que se pierde ordenando. En todo caso, la enseñanza final de toda esta cuestión es que existen herramientas algorítmicas de tipos y condiciones diversas, algunas muy buenas en ciertas condiciones; pero depende de la capacidad de análisis del programador el poder determinar en qué casos usar una u otra técnica.

El algoritmo de *búsqueda binaria* se aplica en la función *binary\_search()* que sigue (incluida también en el módulo *arreglos.py* del proyecto [F13] *Ordenamiento y Búsqueda* que acompaña a esta Ficha):

```
def binary_search(v, x):
    # búsqueda binaria... asume arreglo ordenado...
    izq, der = 0, len(v) - 1
    while izq <= der:
        c = (izq + der) // 2
        if x == v[c]:
            return c
        if x < v[c]:
            der = c - 1
        else:
            izq = c + 1
    return -1
```

El programa *test03.py* (en el mismo proyecto [F13] *Ordenamiento y Búsqueda*), crea un arreglo y procede luego a buscar un valor  $x$  en el mismo por medio de la función anterior. Observe que para facilitar que el arreglo  $v$  esté ordenado, en este programa ese arreglo no se carga por teclado sino que se genera su contenido a través de una función *generate()* que simplemente asigna en el arreglo una secuencia ordenada de múltiplos de 3 (dejamos su análisis para el estudiante):

```
__author__ = 'Cátedra de AED'

import arreglos

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
    if n <= inf:
        print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
```

```

    return n

def generate(v):
    # carga el arreglo con una secuencia ordenada simple...
    n = len(v)
    for i in range(n):
        v[i] = 3*i

def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # generar el arreglo ordenado...
    generate(v)
    print('\nEl vector v fue creado con los siguientes valores:')
    print(v)

    x = int(input('\nValor a buscar en el arreglo: '))

    # aplicar búsqueda secuencial...
    ind = arreglos.binary_search(v, x)

    # avisar por pantalla el resultado de la búsqueda...
    if ind >= 0:
        print('\nEstá en la casilla', ind)
    else:
        print('\nNo está en el arreglo')

# script principal...
if __name__ == '__main__':
    test()

```

#### 4.] Fusión de arreglos ordenados.

Este proceso consiste en generar un *arreglo ordenado c* a partir de otros dos ya cargados y *ordenados previamente*, que designaremos como *a* (con *n* componentes) y *b* (con *m* componentes).

La idea es plantear una función *merge()* para recorrer en paralelo los arreglos *a* y *b*, usando respectivamente los índices *i* y *j*, que originalmente se inicializan en cero. Se toma el valor *a[i]* y se lo compara con *b[j]*. El menor de ambos valores es copiado en el arreglo *c*, en el primer componente libre (que se indica con la variable *k*) [1].

Si el menor fue *a[i]*, entonces se incrementa *i* en uno y el nuevo *a[i]* se compara con el mismo *b[j]* anterior. Si el menor fue *b[j]*, entonces se incrementa *j* en uno, y se compara el nuevo *b[j]* con el mismo valor *a[i]* anterior. En cualquiera de los dos casos, se incrementa también el índice *k* del arreglo *c*, para habilitar un nuevo componente en ese arreglo. El proceso continúa hasta que se llega al final de uno de los dos arreglos originales. O sea, cuando *i* llega a valer *n* (en cuyo caso se llegó al final de *a*), o cuando *j* llega a valer *m* (en cuyo caso se llegó al final de *b*).

Sin embargo, todavía quedan por procesar todos los elementos ubicados al final del arreglo que *aún no se terminó de recorrer*. Aquí la cuestión es simple: todos los valores que

quedaron en ese arreglo, se llevan sin más trámite al final del arreglo *c*, dado que esos elementos están ordenados y son mayores que todos los que ya se cargaron en *c*.

La función *merge()* que aplica este algoritmo se muestra a continuación, y está incluida en el módulo *arreglos.py* del proyecto [F13] Ordenamiento y Búsqueda que acompaña a esta Ficha:

```
def merge(a, b):
    # crear el tercer arreglo con lugar para n + m elementos...
    n, m = len(a), len(b)
    t = n + m
    c = t * [0]

    # aplicar proceso de fusión...
    i = k = j = 0
    while i < n and j < m:
        if a[i] < b[j]:
            c[k] = a[i]
            i += 1
        else:
            c[k] = b[j]
            j += 1
        k += 1

    # determinar cuál de los vectores (a o b) terminó primero...
    # ... apuntar con v al otro...
    v, pos = b, j
    if i < n:
        v, pos = a, i

    # copiar en el vector de salida todos los valores que
    # quedaban en el vector v...
    while pos < len(v):
        c[k] = v[pos]
        pos += 1
        k += 1

    # retornar el vector fusionado...
    return c
```

A su vez, el programa que mostramos más abajo genera los dos arreglos *a* y *b*, de tamaños *n* y *m* respectivamente, con secuencias previamente ordenadas (evitando tener que hacerlo por teclado), y luego invoca a la función *merge()* para proceder a la fusión (ver modelo *test05.py* en el proyecto [F13] Ordenamiento y Búsqueda):

```
__author__ = 'Cátedra de AED'

import arreglos

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def generate(v, factor=3):
    # carga el arreglo con una secuencia ordenada simple...
    n = len(v)
```

```
for i in range(n):
    v[i] = factor*i

def test():
    # generar el primer vector...
    print('Vector a:')
    n = validate(0)
    a = n * [0]
    generate(a, 2)

    # generar el segundo vector...
    print('\nVector b:')
    m = validate(0)
    b = m * [0]
    generate(b, 5)

    # fusionar y obtener el tercer vector ordenado...
    c = arreglos.merge(a, b)

    # mostrar el vector fusionado...
    print('\nEl vector fusionado es:', c)

# script principal...
if __name__ == '__main__':
    test()
```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [3] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

# Ficha 14

## Arreglos – Casos de Estudio I

### 1.] Arreglos correspondientes (o "paralelos").

En muchas ocasiones será necesario almacenar información en varios arreglos unidimensionales a la vez, pero de tal forma que haya correspondencia entre los valores almacenados en las casillas con el mismo índice (llamadas *casillas homólogas*). Por ejemplo, podría ocurrir que se pida guardar en un arreglo los nombres de ciertas personas, y en otro arreglo el importe del sueldo que perciben:

**Figura 1: Dos arreglos correspondientes o paralelos.**

<i>nombres</i>	Juan	Ana	Alejandro	María	Pedro
<i>sueldos</i>	1100	2100	1300	750	800
	0	1	2	3	4

Como se está pidiendo almacenar los datos en *dos* arreglos, sólo se debe cuidar que el nombre y el importe del sueldo de una misma persona aparezcan en ambos arreglos en *casillas homólogas*. Así, en nuestro ejemplo, la persona representada por la casilla 0 del arreglo *nombres* (o sea, *nombres[0]*) se llama "Juan", y el sueldo que percibe aparece en el arreglo *sueldos* también en la casilla con índice 0 (o sea, *sueldos[0]* ) y es el valor 1100. Cuando dos o más arreglos se usan de esta forma, se los suele designar como *arreglos correspondientes o paralelos* [1].

El manejo de ambos arreglos es simple: sólo debe recordar el programador que hay que usar el mismo índice en *ambos arreglos* para entrar a la información de una misma persona. El siguiente esquema, muestra por pantalla el nombre y el sueldo de la persona en la casilla 2:

```
print('Nombre:', nombres[2])      # Alejandro
print('Sueldo:', sueldos[2])      # 1300
```

El siguiente problema servirá para aclarar la forma general de manejar *arreglos paralelos*:

**Problema 37.) Desarrollar un programa que permita cargar tres arreglos con n nombres de personas, sus edades y los sueldos que ganan. Luego de realizar la carga de todos los datos, mostrar los nombres de las personas mayores de 18 años que ganan menos de 10000 pesos, pero de forma que el listado salga ordenado en forma alfabética.**

**Discusión y solución:** El programa usará una función *read()* cuyo objetivo es cargar por teclado los tres arreglos. Cuando se trabaja con arreglos paralelos, en los que la componente en la casilla *i* de cada arreglo contiene información relacionada al mismo objeto o entidad del problema, es conveniente que la carga por teclado se haga de forma de barrer los tres arreglos al mismo tiempo: se carga primero la componente 0 de *todos* los arreglos, luego la componente 1 de *todos* ellos, y así sucesivamente. De esta forma, quien hace la carga de

datos ve facilitada su tarea pues carga los datos de una persona de una sola vez, sin tener que volver atrás luego de cargar los nombres o las edades para cargar los sueldos (lo que sería realmente incómodo). La función `read()` de nuestro programa sería la siguiente:

```
def read(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n):
        nombres[i] = input('Nombre[' + str(i) + ']: ')
        edades[i] = int(input('Edad: '))
        sueldos[i] = int(input('Sueldo: '))
    print()
```

El proceso de ordenamiento será realizado por la función `sort()` mediante nuestro ya conocido algoritmo de ordenamiento por *Selección Directa*. El arreglo de nombres debe ser ordenado en forma alfabética o *lexicográfica*: como sabemos, la idea intuitiva es que las palabras que en un diccionario aparecerían primero se consideran "menores" (alfabéticamente hablando) que las palabras que en el diccionario aparecerían después. Así, el nombre "*Ana*" es lexicográficamente menor que el nombre "*Luis*", pues "*Ana*" aparecería primero en un diccionario que "*Luis*". En Python, La comparación de cadenas es simple y directa: como sabemos, dos cadenas pueden ser comparadas directamente mediante los operadores relacionales típicos, y Python hará el trabajo correcto [2] [3].

Note, no obstante, que la función `sort()` no sólo debe ordenar el arreglo de nombres, sino que debe cuidar que al ordenar ese arreglo se mantenga la correspondencia entre los elementos de ese arreglo y los elementos de los otros dos que contienen las edades y los sueldos. El principio es simple: si se realiza un cambio en el arreglo de nombres, el mismo cambio debe reflejarse en los otros dos, por lo cual se realizan tres operaciones de intercambio (una para el arreglo de nombres, otra para el arreglo de edades y otra para el de sueldos). La función que hace el ordenamiento sería la que sigue:

```
def sort(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n-1):
        for j in range(i+1, n):
            if nombres[i] > nombres[j]:
                nombres[i], nombres[j] = nombres[j], nombres[i]
                edades[i], edades[j] = edades[j], edades[i]
                sueldos[i], sueldos[j] = sueldos[j], sueldos[i]
```

Finalmente, la función `display()` usa un ciclo `for` para analizar cada componente del arreglo de edades y del arreglo de sueldos. Si se encuentra que la edad en la posición *i* es mayor a 18 y al mismo tiempo el sueldo es menor a 10000, se muestran los tres datos que corresponden a esa persona, que se encuentra en cada arreglo en la misma posición *i*. La función podría ser la siguiente:

```
def display(nombres, edades, sueldos):
    n = len(nombres)
    print('ayores de 18 que ganan menos de 10000 pesos:')
    for i in range(n):
        if edades[i] > 18 and sueldos[i] < 10000:
            print('Nombre:', nombres[i], 'Edad:', edades[i], 'Sueldo:', sueldos[i])
```

El programa completo que se muestra a continuación. está incluido en el modelo `tes01.py` que forma parte del proyecto [F14] Arreglos que acompaña a esta Ficha.

```
def validate(inf):
    n = inf
    while n <= inf:
```

```

n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
if n <= inf:
    print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
return n

def read(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n):
        nombres[i] = input('Nombre[' + str(i) + ']: ')
        edades[i] = int(input('Edad: '))
        sueldos[i] = int(input('Sueldo: '))
    print()

def sort(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n-1):
        for j in range(i+1, n):
            if nombres[i] > nombres[j]:
                nombres[i], nombres[j] = nombres[j], nombres[i]
                edades[i], edades[j] = edades[j], edades[i]
                sueldos[i], sueldos[j] = sueldos[j], sueldos[i]

def display(nombres, edades, sueldos):
    n = len(nombres)
    print('Personas mayores de 18 que ganan menos de 10000 pesos:')
    for i in range(n):
        if edades[i] > 18 and sueldos[i] < 10000:
            print('Nombre:', nombres[i], 'Edad:', edades[i], 'Sueldo:', sueldos[i])

def test():
    # cargar cantidad de personas...
    n = validate(0)

    # crear los tres arreglos de n elementos...
    nombres = n * [' ']
    edades = n * [0]
    sueldos = n * [0]

    # cargar los tres arreglos por teclado...
    print('\nCargue los datos de las personas:')
    read(nombres, edades, sueldos)

    # ordenar alfabéticamente el sistema...
    sort(nombres, edades, sueldos)

    # avisar por pantalla el resultado de la búsqueda...
    display(nombres, edades, sueldos)

# script principal...
if __name__ == '__main__':
    test()

```

## 2.] Conteo y/o acumulación por acceso directo (vectores de conteo y/o acumulación).

Los arreglos son estructuras de datos especialmente útiles, puesto que brindan la posibilidad del *acceso directo* a cada componente. Es decir: si se necesita acceder directamente a un elemento, sólo se requiere conocer el índice del mismo y no es necesario pasar en forma secuencial por todos los elementos anteriores. Existen situaciones (el caso de la *búsqueda binaria* es una de ellas) en las cuales esta propiedad permite resolver problemas en forma notablemente sencilla y rápida. Por ejemplo, considérese el siguiente problema:

**Problema 38.)** *Cargar por teclado un conjunto de valores tales que todos ellos estén comprendidos entre 0 y 99 (incluidos ambos). Se indica el fin de datos con el número -1. Determinar cuántas veces apareció cada número.*

**Discusión y solución:** A primera vista, el problema parece de solución obvia: se usa un ciclo para cargar de a un valor (*num*) por vez, de forma que el ciclo sólo se detenga cuando sea *num == -1*. Como se pide determinar cuántas veces apareció cada valor posible de *num*, se usa un contador distinto por cada valor diferente que *num* pueda asumir. Así, para contar cuántas veces apareció el 1, se puede usar el contador *c1*, para el 2 se podrá usar *c2*, y así sucesivamente. En cada repetición del ciclo se usa un *if encadenado* para determinar qué número se cargó en esa vuelta, y en función del número se elige el contador correspondiente. Al finalizar el ciclo, se muestran todos los contadores, y asunto terminado...

Sin embargo, esa solución dista mucho de ser buena y sobre todo en casos como éste, en que la variable *num* puede asumir valores en un rango muy amplio (0 a 99). El programador debería declarar y poner en cero *cien contadores*, y luego, dentro del ciclo, usar *icien condiciones anidadas!* para determinar qué contador debería usar de acuerdo al número ingresado. Además del gran esfuerzo de codificación que el programador deberá realizar, se tendrá el problema de la enorme redundancia de instrucciones semejantes, y por si esto fuera poco, el programa resultante será muy poco flexible en la práctica: si luego de plantearlo nos cambian las condiciones supuestas (por ejemplo: si los valores que puede asumir *num* pasaran a ser entre 0 y 150) entonces el programa original ya no serviría de nada...

La solución correcta es usar un *arreglo unidimensional c de cien componentes*, de forma que cada componente se use como uno de los contadores que se están necesitando [1]. Se pone inicialmente en cero cada componente del vector (dado que esos componentes serán contadores). Luego comienza el ciclo para cargar los valores *num*. La idea central, es que si *num* vale 0, entonces debe usarse *c[0]* para contarla. Si *num* vale 1, se usa *c[1]*, y así sucesivamente. En general, para cada valor de la variable *num* tal que  $0 \leq num \leq 99$ , debe usarse *c[num]* para contarla... En otras palabras: dentro del ciclo *no es necesario* usar condiciones anidadas para determinar qué casillero del arreglo usar para contar cada número: se accede *directamente* al casillero cuyo índice es *num* y se incrementa el mismo en uno... Al cortar el ciclo, se muestra el contenido del arreglo, y ahora sí el problema queda resuelto de manera convincente.

Cuando un arreglo unidimensional se usa de esta forma, se lo suele designar como *vector de conteos* o simplemente como *vector de contadores*<sup>1</sup>. Si el arreglo se usara para *acumular* valores (en vez de *contarlos* como se supuso en el ejemplo), se hablaría entonces de un *vector de acumulación*.

---

<sup>1</sup> Una situación de conteo muy conocida se da en las famosas *cuentas regresivas* que tienen lugar en distintos tipos de eventos (lanzamientos de naves espaciales, llegada del Año Nuevo, etc.) A su vez, es muy común en el cine encontrar escenas basadas en cuentas regresivas: Una de esas escenas se dio en la película *Independence Day* (o *Día de la Independencia*), de 1996, dirigida por *Roland Emmerich* y protagonizada por *Will Smith* y *Jeff Goldblum*. Una avanzada (y hostil) civilización extraterrestre llega a la Tierra y posiciona numerosas naves en el espacio aéreo de muchas ciudades importantes de todo el mundo, como si estuviesen vigilándolas. Estas naves emiten una extraña señal mientras se están posicionando, y pronto alguien descubre que la señal es en realidad una cuenta regresiva... que en cuanto llegue a cero provocará el ataque simultáneo de todas esas naves a todas las ciudades vigiladas, iniciando una invasión.

Se muestra a continuación el programa que aplica lo dicho y resuelve el problema (ver modelo *test02.py* en el proyecto *[F14] Arreglos*): en la función *test()* se define el vector de conteos *c* de  $n = 100$  elementos valiendo cada uno de ellos inicialmente 0, que se usará para contar cuántas veces aparecen los números en el rango  $[0, n-1]$ . Luego se cargan los números mediante un ciclo de doble lectura, *y se aplica la técnica vista para proceder a contarlos*. Al finalizar, se recorre el vector de conteos y se visualizan los resultados, pero de forma que sólo se muestran los conteos de los números que hayan entrado al menos una vez (lo cual ahorra algo de espacio en la consola de salida...):

```
def test():
    # crear el vector de conteo...
    n = 100
    c = n * [0]

    # cargar y contar los números...
    num = int(input('Ingrese un valor entre 0 y 99 (con -1 corta): '))
    while num != -1:
        if 0 <= num < n:
            c[num] += 1
        else:
            print('Error... el número debía ser >= 0 y <', n)
        num = int(input('Ingrese otro valor entre 0 y 99 (con -1 corta): '))

    # mostrar los resultados...
    print('Resultados:')
    for i in range(n):
        if c[i] != 0:
            print('Número', i, '- Frecuencia de aparición', c[i])

if __name__ == '__main__':
    test()
```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 15

## Arreglos – Casos de Estudio II

### 1.] Arreglos: análisis y solución de problemas básicos.

El contenido completo de esta Ficha está orientado al análisis, discusión y solución de diversos problemas que requieren el uso y aplicación de arreglos. La Ficha aporta entonces algunas técnicas elementales en ese marco y sirve al mismo tiempo como un elemento de repaso de los temas generales que surgieron hasta ahora. El primero de los problemas a discutir es el siguiente:

**Problema 39.)** *Desarrollar un programa que permita cargar un arreglo con las alturas de n personas. Determinar la altura media del grupo, e informar cuántas de esas personas tienen altura mayor a la media, y cuántas tienen altura menor o igual a la media.*

**Discusión y solución:** El programa completo se muestra directamente a continuación:

```
def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Ingrese n (mayor a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def read(alt):
    n = len(alt)
    print('Cargue ahora las alturas (en centímetros) del grupo...')
    for i in range(n):
        alt[i] = int(input('Altura[' + str(i) + ']: '))

def average(alt):
    n, s = len(alt), 0
    for i in range(n):
        s += alt[i]

    return s/n

def count(alt, med):
    n = len(alt)
    c1 = c2 = 0
    for i in range(n):
        if alt[i] > med:
            c1 += 1
        else:
            c2 += 1
```

```

return c1, c2

def test():
    # cargar cantidad de personas...
    n = validate(0)

    # crear el arreglo de n elementos...
    alturas = n * [0]

    # cargar arreglo por teclado...
    read(alturas)

    # calcular el promedio y efectuar el conteo...
    media = average(alturas)
    mayores, menores = count(alturas, media)

    # mostras resultados...
    print('La altura media del grupo es:', media)
    print('Alturas mayores a la media:', mayores)
    print('Alturas menores a la media:', menores)

# script principal...
if __name__ == '__main__':
    test()

```

La creación y carga del arreglo con todas las alturas a procesar no ofrece dificultades. La función *test()* (que sirve como punto de arranque del programa) carga por teclado el valor *n* (validando que sea mayor a cero), crea el arreglo de componentes (initialmente con ceros en cada casillero), e invoca a la función *read()* para cargar por teclado efectivamente los datos. Por razones de brevedad, la carga de estos datos se hizo sin validación (dejamos para los estudiantes la tarea de hacer esas validaciones).

El paso siguiente es el cálculo de la *altura media* (o *altura promedio*). La función *test()* invoca para eso a la función *average()*, la cual toma como parámetro al arreglo con las alturas, acumula esas alturas, y retorna el cociente entre el acumulador y el valor de *n* (lo cual es efectivamente la altura media pedida).

La función *count()* desarrolla el proceso final: toma como parámetro el arreglo de alturas y la altura media, recorre nuevamente el arreglo y simplemente usa dos contadores para llevar la cuenta de los valores que sean mayores que la media y los que sean menores o iguales a ella. Esos dos contadores son retornados a la función *test()*, que finalmente muestra en pantalla los resultados así obtenidos.

Está claro que el problema que se acaba de presentar es de naturaleza sencilla y no ofrece dificultades especiales, pero resulta de interés para remarcar un hecho práctico: el conteo de los valores que son mayores (o menores) que el promedio de los datos contenidos en un arreglo, requiere obligatoriamente dos recorridos de ese arreglo: uno para hacer la acumulación necesaria para el cálculo del promedio (función *average()* en nuestro caso), y otro para comparar los valores contra el promedio ya calculado y efectuar el conteo (función *count()* en nuestro caso) [1].

Presentamos ahora un nuevo problema para analizar y discutir:

**Problema 40.)** Desarrollar un programa que permita cargar por teclado dos arreglos de números enteros de  $n$  y  $m$  componentes respectivamente, y genere y muestre un tercer arreglo que contenga los valores que aparecen repetidos en los dos arreglos originales. Es decir, el nuevo arreglo debe contener todos los números que estando en uno de los dos arreglos originales, están también el otro. Si un número está dos o más veces en el mismo arreglo (y también figura en el segundo arreglo), sólo debe aparecer una vez en el tercer arreglo.

**Discusión y solución:** Diseñaremos una función *generate()* que será la encargada de crear el tercer arreglo *v3* a partir de los dos originales *v1* y *v2* que reciba como parámetro. La consigna general es que todo elemento de *v1* que esté también en *v2*, se agregue a *v3*. En principio, la estrategia básica es simple: se recorre el arreglo *v1* con un ciclo, y se compara el elemento actual *v1[i]* contra cada uno de los elementos de *v2* (usando otro ciclo). Cada vez se detecte que *v1[i]* está en *v2*, se agrega *v1[i]* a *v3* con el método *append()*. El planteo mínimo y esencial de esa función podría ser el siguiente:

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):
        for j in range(len(v2)):
            if v1[i] == v2[j]:
                # si v1[i] está en v2, agregarlo en v3...
                v3.append(v1[i])

    # retornar el nuevo arreglo...
    return v3
```

Así planteada, la función *generate()* efectivamente encontrará todos los números que estén a la vez en *v1* y *v2*, y los copiará en *v3*. Si los arreglos fuesen (por ejemplo) *v1* = [5, 3, 2, 4] y *v2* = [1, 5, 7, 3, 8] entonces la función *generate()* crearía y retornaría correctamente el arreglo *v3* = [5, 3].

Sin embargo, hay al menos un problema con el planteo propuesto para *generate()*: podría ocurrir que *v2* contuviese más de una vez un mismo número que también esté en *v1*. Por ejemplo, si el arreglo *v2* del ejemplo anterior fuese *v2* = [1, 5, 7, 3, 5] entonces la función *generate()* que propusimos generaría un arreglo de la forma *v3* = [5, 5, 3] con el 5 repetido... Esto se debe a que dentro del ciclo que recorre a *v2* (el ciclo regulado por *j*) se comparan **todos** los elementos de *v2* contra el elemento actual *v1*, y cada vez que se la condición es *True* el elemento *v1[i]* se agrega en *v3*. La forma obvia de arreglar este problema es cortar con **break** el ciclo más interno [2] (el que recorre a *v2*, regulado con la variable *j*) en el primer momento en que la condición sea *True*. El cambio es simple de implementar:

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):
        for j in range(len(v2)):
            if v1[i] == v2[j]:
                # si v1[i] está en v2, agregarlo en v3...
                v3.append(v1[i])

                # ... y cortar el ciclo para evitar repeticiones...
                break
```

```
# retornar el nuevo arreglo...
return v3
```

Con este cambio, si el valor que se agrega en v3 está repetido dos o más veces en v2, será añadido sólo una vez en v3, con lo que parecería que el problema ha quedado resuelto...

Pero queda un caso más: el valor que se toma de v1 podría estar repetido a su vez en el propio arreglo v1, y eso también provocaría que se copie más de una vez en v3. Por ejemplo, si fuese v1 = [5, 3, 5, 4] y v2 = [1, 5, 7, 3, 8] entonces se crearía el arreglo v3 = [5, 3, 5] con el 5 repetido... La solución a este caso consiste en que antes de buscar el valor v1[i] en v2, **se lo busque primero en v3** para comprobar si ya había sido agregado anteriormente, y en ese caso no buscarlo en v2. La versión final de la función *generate()* sería la que sigue:

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):

        # comprobar si v1[i] ya está en v3...
        exists = False
        for k in range(len(v3)):
            if v3[k] == v1[i]:
                exists = True
                break

        # si v1[i] no está en v3, buscarlo en v2...
        if not exists:
            for j in range(len(v2)):
                if v1[i] == v2[j]:
                    # si está en v2 agregarlo y cortar el ciclo...
                    v3.append(v1[i])
                    break

    # retornar el nuevo arreglo...
    return v3
```

El programa completo se muestra a continuación (dejamos el resto de los detalles del programa para ser analizados por cada estudiante):

```
def validate(inf):
    t = inf
    while t <= inf:
        t = int(input('Cantidad (mayor a ' + str(inf) + ' por favor): '))
        if t <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')

    return t

def read(v):
    n = len(v)
    print('Cargue ahora los datos de este arreglo...')
    for i in range(n):
        v[i] = int(input('Valor[' + str(i) + ']: '))

def generate(v1, v2):
    v3 = []
```

```

for i in range(len(v1)):

    # comprobar si v1[i] ya está en v3...
    exists = False
    for k in range(len(v3)):
        if v3[k] == v1[i]:
            exists = True
            break

    # si v1[i] no está en v3, buscarlo en v2...
    if not exists:
        for j in range(len(v2)):
            if v1[i] == v2[j]:
                # si está en v2 agregarlo en v3 y cortar el ciclo...
                v3.append(v1[i])
                break

# retornar el nuevo arreglo...
return v3

def test():
    # primer arreglo...
    print('Primer arreglo...')
    n = validate(0)
    v1 = n * [0]
    read(v1)
    print()

    # segundo arreglo...
    print('Segundo arreglo...')
    m = validate(0)
    v2 = m * [0]
    read(v2)
    print()

    # generar y mostrar el tercer arreglo...
    v3 = generate(v1, v2)
    print('Los valores presentes en ambos arreglos son:')
    print(v3)

if __name__ == '__main__':
    test()

```

El último problema que veremos es el siguiente:

**Problema 41.)** *Cargar por teclado un arreglo de longitud n, y determinar si los elementos del mismo están en secuencia de k en k, siendo k un número que se ingresa también por teclado. Por ejemplo:*

$v = [2, 5, 8, 11, 14]$  está en secuencia de 3 en 3 ( $k = 3$ )  
 $v = [4, 6, 8, 10]$  está en secuencia de 2 en 2 ( $k = 2$ )

**Discusión y solución:** El análisis del contenido del arreglo para comprobar si sus elementos están en secuencia de  $k$  en  $k$  será realizado a través de la función  $sequence(v, k)$ , la cual toma como parámetro al propio arreglo  $v$  ya creado y cargado, y al valor  $k$  que indica la amplitud de la secuencia. La función  $sequence(v, k)$  retorna *True* si efectivamente los datos contenidos en  $v$  están en secuencia de  $k$  en  $k$ , o retorna *False* en caso contrario.

La idea es directa: se recorre el arreglo con un ciclo comenzando desde la casilla 1, y se comprueba en cada vuelta si el valor de la casilla  $v[i]$  coincide con el de la casilla inmediatamente anterior  $v[i-1]$  pero sumado a  $k$ . Si la secuencia se cumple en todo el arreglo, entonces todas las comprobaciones de la forma  $v[i] \neq v[i-1] + k$  deben ser falsas. Por lo tanto, si alguna de esas comprobaciones fuese *True*, la función puede terminar en ese mismo momento retornando *False* sin necesidad de analizar el resto del arreglo (la secuencia se rompe con un solo par de casillas en las que no se cumpla). La función *sequence()* puede entonces quedar así:

```
def sequence(v, k):
    n = len(v)
    for i in range(1, n):
        if v[i] != v[i-1] + k:
            return False

    return True
```

El resto del programa (que se muestra completo a continuación) no presenta mayores dificultades, por lo que dejamos su análisis para el estudiante:

```
def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')

    return n

def read(v):
    n = len(v)
    print('Cargue ahora los datos...')
    for i in range(n):
        v[i] = int(input('v[' + str(i) + ']: '))

def sequence(v, k):
    n = len(v)
    for i in range(1, n):
        if v[i] != v[i-1] + k:
            return False

    return True

def test():
    # cargar tamaño del arreglo...
    print('Ingrese tamaño del arreglo...')
    n = validate(0)
    print()

    # crear el arreglo de n elementos...
    v = n * [0]

    # cargar arreglo por teclado...
    read(v)
    print()
```

```

# cargar el valor de k...
print('Ingrese el valor de k...')
k = validate(0)

# comprobar la secuencia...
ok = sequence(v, k)

# mostras resultados...
if ok:
    print('El arreglo está en secuencia de', k, 'en', k)
else:
    print('El arreglo no está en secuencia de', k, 'en', k)

# script principal...
if __name__ == '__main__':
    test()

```

## 2.] Encriptación simple de mensajes.

La humanidad desarrolló la escritura hace aproximadamente 6000 años... y es posible que también desde esa misma época haya surgido la necesidad de proteger un escrito importante para que sólo pueda ser entendido y leído por las personas adecuadas. Muy probablemente haya sido el campo militar el que primero requirió desarrollar técnicas para *encriptar* (o *cifrar*) un mensaje: es común en este campo que un comandante deba enviar órdenes a algún subalterno o a algún aliado, pero de tal forma que esas órdenes sólo puedan ser interpretadas por el subjefe o aliado correcto. Si el mensaje cayera en manos extrañas (por ejemplo, en manos del enemigo) el daño recibido podría ser tan grande como la pérdida de la guerra.

A medida que la civilización progresó, aumentaron también las necesidades de proteger un mensaje contra ojos extraños u hostiles. Sobran los ejemplos. Hoy en día, es muy común (e indispensable...) proteger con algún tipo de encriptado las claves que se usan para acceder a información bancaria o a cualquier tipo de información que un usuario considere tan importante como para impedir a toda costa su acceso por parte de personas no autorizadas.

Como se ve, la *criptología* (ciencia que estudia la forma de ocultar un mensaje o una transmisión) existe desde mucho tiempo antes que la ciencia de la computación, pero esta última ha provisto nuevos elementos (en velocidad y potencia de cálculo) que han abierto el juego e incluso han cambiado sus reglas. En esta Ficha de estudio veremos una aproximación a un par de métodos ***muy simples y muy elementales*** para encriptar un mensaje, y sus implementaciones básicas, pero quede claro que esas implementaciones son sólo ejemplos o modelos de análisis, y no son implementaciones finales ni listas para ser usadas en situaciones reales. La aplicación de cualquier técnica de encriptación a una situación real, con verdadero peligro de transgresión por parte de un “atacante”, debe ser estudiada con mucho cuidado y profesionalismo, sin subestimar ningún detalle ni hacer suposiciones que luego puedan significar caer en una trampa...

En general, llamaremos *mensaje en claro* (o *mensaje abierto*) al texto o información original que se quiere proteger; y designaremos como *mensaje encriptado* (o *mensaje cifrado*) al mensaje o cadena de símbolos que se obtiene una vez aplicado el mecanismo de ocultación que se haya elegido. Ese mecanismo de ocultación, a su vez, será designado como el

algoritmo o *método de encriptación* (o *método de cifrado*). Un *criptógrafo* es la persona que intenta encriptar un mensaje, y un *criptoanalista* es la persona que intenta descifrarlo o desencriptarlo (sin conocer el algoritmo ni las claves originales para hacerlo) [3]. Dependiendo el cristal con que se mire, algunos considerarán como “buenos” a los criptógrafos y como “malos” a los criptoanalistas... y otros lo harán al revés. El hecho es que parece inevitable que haya “bandos” que querrán ocultar sus transmisiones, y otros “bandos” que querrán violar las medidas de seguridad del bando opuesto para ver esas transmisiones.

Uno de los métodos más simples y más antiguos para encriptar un mensaje se atribuye a Julio César, el general y luego emperador romano. Se designa por ese motivo como el *método de César*, y la idea es simple y directa: Si una letra del mensaje ocupa el *n*-ésimo lugar del alfabeto, reemplácela por la que ocupa el  $(n+k)$ -ésimo lugar, donde  $k$  es un número entero prefijado y sólo conocido por el encriptador y sus aliados [3].

Por ejemplo si  $k = 3$  y el mensaje original es ‘CASA’, entonces el mensaje codificado sería ‘FDVD’ (suponiendo que el alfabeto original consta sólo de las letras mayúscula y ningún otro símbolo): cada letra se reemplaza por la que esté a  $k = 3$  lugares más adelante en el alfabeto, y así la letra ‘C’ se reemplaza por la ‘F’, la letra ‘A’ se reemplaza por la ‘D’ y la ‘S’ por la ‘V’. Puede verse que esta técnica es muy fácil de transgredir: en nuestro ejemplo, el criptoanalista o atacante sólo debería saber cuál es el alfabeto original, y probar un máximo de 27 valores diferentes de  $k$  (suponiendo que el alfabeto original tiene las 27 letras simples (sin la LL ni la CH pero incluida la Ñ) del idioma español, hasta dar con un mensaje legible. En nuestros días, aun cuando el alfabeto original conste de muchos más símbolos (como podría ser la tabla ASCII completa con 256 símbolos) se pueden probar con rapidez todas las variantes de  $k$  usando una computadora.

Para mejorar la resistencia de esta técnica frente a un ataque, se puede ensayar una sencilla variante: en vez de usar el alfabeto original para obtener la letra o símbolo de reemplazo, se puede usar una *segunda tabla* (que llamaremos *tabla de transposición*) la cual puede estar formada por los mismos símbolos del alfabeto original (ipero en otro orden!) o bien puede estar formada directamente por otros símbolos [3]. Si suponemos que el alfabeto original sólo consta de la letras mayúsculas, podemos usar (a modo de ejemplo) la *tabla de transposición* que se ve en el ejemplo siguiente:

<b>Alfabeto:</b>	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z
<b>Transposición:</b>	R	I	S	Q	P	A	N	O	W	X	U	M	D	H	Z	T	F	G	B	L	E	Y	K	C	J	V	Ñ

Así, si el mensaje fuera “UN OJO EN EL CIELO” tendría el siguiente cifrado (ignorando los espacios en blanco): “YH TXT PH PM SWPMT”. Como se ve, la idea es simplemente reemplazar cada letra del texto en claro por su homóloga en la tabla de transposición. Obviamente, la tabla de transposición no debería ser un elemento conocido más que por los criptógrafos...

Con esta técnica, un atacante debería probar con una gran cantidad de tablas diferentes (si el alfabeto tiene  $n$  símbolos, serían alrededor de  $n!$  tablas diferentes) Sin embargo, todavía un criptoanalista podría violar el esquema haciendo un análisis minucioso: sabemos que en cada idioma hay letras que aparecen con más frecuencia que otras: en español la ‘A’ o en

inglés la '*E*' son las letras que más aparecen. Por lo tanto, un atacante podría buscar cuál es el símbolo que más se repite en el mensaje encriptado, y casi seguro tendría una letra en claro definida. También hay ciertas combinaciones que aparecen mucho (como '*la*' o '*en*' o '*el*' en español o '*the*' o '*and*' en inglés) y por lo tanto, analizando grupos de dos o tres símbolos en el mensaje encriptado, un criptoanalista también tendría razonables posibilidades de éxito (con cierta facilidad un atacante podría deducir del ejemplo anterior que la secuencia encriptada '*PH*' corresponde a la secuencia en claro '*EN*', y de allí sería factible deducir el resto, sobre todo si el mensaje encriptado es largo).

En esta Ficha de estudios mostraremos solamente una forma de implementar la *técnica de César* y la *técnica de Tabla de Transposición*, pero hay que dejar dos hechos en claro: el primero, es que obviamente existen muchísimas estrategias adicionales de encriptación [3], como la *encriptación de Vigenère* (que permite mejorar la encriptación basada en tablas de transposición usando una secuencia numérica auxiliar designada como *clave de cifrado*) y que por mucho tiempo fue considerado indescifrable hasta que surgió el *método de Kasiski* especialmente diseñado para romper el *cifrado de Vigenère*), el *cifrado de Vernam* (una variante de la *encriptación de Vigenère* en la que la clave de cifrado tiene la misma longitud que el mensaje a encriptar) o el *cifrado de Mauborgne* (también conocido como *cifrado de un solo uso*, y que esencialmente es el mismo cifrado de Vernam pero haciendo que la clave de cifrado sea aleatoria y se use sólo una vez<sup>1</sup>).

El segundo hecho a dejar claro, es que las técnicas que someramente hemos descripto en los párrafos anteriores son estrategias antiguas y hoy en día obsoletas (se las designa como *métodos de encriptación clásicos*). Todos ellos cayeron en desuso con el advenimiento de las computadoras, cuya potencia de cálculo y velocidad de ejecución para probar miles de combinaciones de patrones en poco tiempo hicieron que los métodos clásicos se convirtiesen en simples juegos de ingenio.

En la civilización moderna, los sistemas de cifrado deben ser capaces de resistir un ataque informático masivo y por ello las técnicas usadas son dramáticamente diferentes a las clásicas. La mayor parte de las técnicas modernas (como el famoso *algoritmo RSA de clave pública* [3], diseñado por *Rivest, Shamir y Adleman* –de cuyas iniciales toma su nombre la técnica-) están basadas en algoritmos de base matemática que en alguna parte incluyen la necesidad de encontrar números primos muy pero muy grandes (del orden de 100 dígitos o más cada uno) o de factorizar números muy grandes para encontrar sus factores primos (que también son muy grandes). Sabemos (por lo que hemos mostrado en la Ficha 09) que el proceso de factorización de un número en sus divisores primos puede demorar una increíble cantidad de tiempo (incluso para una computadora) si el número es muy grande o alguno de sus factores primos lo es. No se conocen hasta ahora algoritmos rápidos para hacer ese trabajo, y de alguna manera toda la seguridad de los procesos informáticos modernos

---

<sup>1</sup> El tema de las consecuencias de cualquier evento aleatoriamente generado y su impacto en los hechos posteriores, es un tema recurrente en el cine y la literatura. En 2004, la película *The Butterfly Effect* (o *El Efecto Mariposa*) (dirigida por *Eric Bress* e interpretada por *Ashton Kutcher*) planteó en forma cruda la historia de un joven estudiante de psicología que de pronto descubre que puede regresar en el tiempo, y decide hacerlo para cambiar sus decisiones pasadas y alterar con eso algunos sucesos tristes o lamentables de su vida... sin tener en cuenta el significado profundo del eslogan básico de la película: "el aleteo de una simple mariposa puede provocar un tsunami al otro lado del mundo". Y las cosas cambiaron, pero no en la forma que él esperaba...

(transacciones bancarias, acceso a sitios académicos, redes sociales, etc.) descansa en nuestra confianza de que esos algoritmos rápidos de factorización no existen... aún.

Formalmente, la implementación de los algoritmos de *cifrado de César* y de *Transposición* se muestra en la solución al siguiente problema:

**Problema 42.)** *Desarrollar un programa que permita cargar por teclado un mensaje compuesto sólo por mayúsculas y espacios en blanco, y lo encripte usando el cifrado de César y el cifrado de Transposición. Diseñe también los algoritmos de desencriptado.*

**Discusión y solución:** El programa completo que resuelve este problema se puede ver en el modelo *test04.py* del proyecto *F[15] Arreglos* que acompaña a esta Ficha.

Comencemos por el cifrado de César. La función *code\_cesar(mens, ABC, k)* que mostramos a continuación, toma como parámetro una cadena con el mensaje a encriptar (parámetro *mens*), otra cadena con el alfabeto empleado (parámetro *ABC*) y el factor de desplazamiento a usar para el cifrado (parámetro *k*, un número entero que debería ser mayor a 1). A su vez, La función auxiliar *is\_ok(cad, ABC)* controla si la cadena recibida en el parámetro *cad* está formada sólo por blancos y caracteres que estén en el alfabeto recibido en el parámetro *ABC* (retorna *True* si ese fuese el caso, o *False* si algún carácter en *mens* no es válido):

```
def is_ok(cad, ABC):
    if cad is None:
        return False

    if cad == '':
        return False

    for i in range(len(cad)):
        # si cad[i] no es blanco y no está en el alfabeto, retornar False...
        if cad[i] != ' ' and ABC.find(cad[i]) == -1:
            return False

    return True

def code_cesar(mens, ABC, k):
    if not is_ok(mens, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(mens)):
        # si el carácter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        # si el carácter original no es blanco, reemplazarlo...
        else:
            # índice del carácter mens[i] en el alfabeto...
            im = ABC.find(mens[i])

            # índice del carácter de reemplazo en el alfabeto...
            ir = (im + k) % na

            # efectuar el reemplazo en el arreglo de salida b...
            b.append(ABC[ir])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr
```

La función `code_cesar(mens, ABC, k)` comienza controlando que la cadena recibida en el parámetro `mens` sea válida, invocando a la función `is_ok()`. Si algo estuviese mal, `code_cesar()` retorna `None` (a modo de aviso de procesamiento imposible).

Si la cadena que viene en `mens` es correcta, se crea un arreglo `b` inicialmente vacío para ir llenándolo luego con los caracteres encriptados que correspondan. Podría pensarse que `b` se inicie como una cadena vacía (en lugar de un arreglo vacío), pero como las cadenas son inmutables el proceso de ir llenando la cadena con los caracteres encriptados requeriría un mecanismo poco práctico y algo más lento que si se hace con un arreglo.

A continuación, la función `code_cesar()` emplea un ciclo `for` para recorrer la cadena `mens`, carácter a carácter. Si carácter actual `mens[i]` es un blanco, se copia el mismo blanco en el arreglo `b` (por razones de claridad, no encriptaremos los blancos, pero en un contexto real serían encriptados como cualquier otro carácter). Si el carácter actual `mens[i]` no es un blanco, se lo busca en el alfabeto (contenido en la cadena que viene en `ABC`) con el método `find()` provisto por la clase `str` (que representa a las cadenas de caracteres en Python). La instrucción [2]:

```
im = ABC.find(mens[i])
```

busca en la cadena `ABC` la cadena (en este caso, el carácter) representada por `mens[i]` y si la encuentra, retorna el índice donde `mens[i]` comienza dentro de `ABC`. Si no es encontrada, el método retorna el valor `-1`. Sabiendo entonces que `im` contiene el índice que le corresponde al carácter `mens[i]` dentro del alfabeto `ABC`, sólo quedaría sumar a `im` el valor `k` para obtener el índice del carácter de reemplazo. Pero hay un pequeño problema: la cadena `ABC` tiene cierta longitud `n` (que en nuestro caso es `n = 27`) y la suma `im + k` podría ser mayor a `n`. Por ejemplo, si el carácter `mens[i]` fuese la letra 'Y' (cuyo índice en `ABC` es `im = 25`) y el valor de `k` es 3, entonces la suma `im + k` sería  $25 + 3 = 28$  que no es un índice válido para `ABC`. Pero el caso es que la letra 'Y' necesita de todos modos un carácter de reemplazo, y la idea obvia en situaciones como esta es considerar al alfabeto como una *tabla circular*: si al buscar el reemplazo se supera el límite derecho de la tabla, continuar desde el principio y recorrer desde allí. En el caso de la 'Y', si tenemos que ir  $k = 3$  casillas hacia "adelante", habría que avanzar hasta la letra 'B' (es decir, llegar hasta la 'Z' en la casilla 26, dar la vuelta y pasar por la 'A' en la casilla 0, y parar en la 'B' en la 1). La forma de calcular el índice del carácter de reemplazo si la tabla `ABC` se toma en forma "circular", consiste en aplicar aritmética modular, tal como se vio en la Ficha 02: el índice del carácter de reemplazo puede calcularse con sencillez mediante la expresión:

```
na = len(ABC)
ir = (im + k) % na
```

Efectivamente, si el valor `im + k` es mayor que el tamaño `na` de la tabla entonces el resto de dividir por ese tamaño dará el valor del "exceso" a partir del final, que es justamente el índice del casillero de reemplazo comenzando desde el principio. En nuestro caso, si `im = 25` (el índice la letra 'Y'), y el valor de `k` es 3, con `na = 27`, entonces  $(25 + 3) \% 27 = 28 \% 27 = 1$ , que es el índice la letra 'B' en la cadena `ABC`... Es interesante recordar que si `im + k` es menor que `na`, entonces el resto es igual al propio valor `im + k`, y en ese caso el índice de reemplazo es exactamente el que se habría usado si la tabla no se tomase como circular. Por caso, el índice la letra 'C' es `im = 2`. Si se toma `k = 3` con `na = 27`, la expresión de cálculo resulta

entonces  $(2 + 3) \% 27 = 5 \% 27 = 5$ , con lo que la letra 'C' sería reemplazada (correctamente) por la 'F'.

La función `code_cesar()` calcula uno a uno los índices de los caracteres de reemplazo aplicando las técnicas explicadas más arriba, y agrega en el arreglo `b` esos caracteres de reemplazo con el método `append()`. Al terminar el ciclo, sólo restaría entonces convertir el arreglo `b` a una cadena de caracteres, y retornar esa cadena. Pero esto nos lleva al último detalle técnico a resolver: si `b` es un arreglo (un objeto de tipo `list`) que contiene caracteres, podría pensarse que la función `str()` bastaría para convertir ese arreglo en una cadena... pero `str()` no hace exactamente lo que esperaríamos. Supongamos que el arreglo `b` quedó con los valores `b = ['F', 'D', 'V', 'D']`. Si entonces hacemos algo como:

```
encr = str(b)
```

para obtener la cadena final, lo que `str()` haría en realidad sería almacenar en `encr` la cadena `"['F', 'D', 'V', 'D']"` que claramente no es lo que necesitamos. La manera final y correcta de hacer la conversión consiste en invocar al método `join()` de la clase `str()` en la forma que se muestra a continuación:

```
encr = ''.join(b)
```

El método `join()` **concatena** los elementos que contiene la secuencia que toma como parámetro (el arreglo `b` en nuestro caso) creando una cadena de caracteres normal, y retorna la cadena así creada [2]. El método usa como separador de las cadenas que concatena a la cadena que se empleó para invocar al método (en nuestro caso, la cadena vacía: `"`). Así, si el arreglo `b` fuese `b = ['F', 'D', 'V', 'D']`, entonces la instrucción `encr = ''.join(b)` asignaría en `encr` la cadena `'FDVD'` esperada. Note que si la cadena usada para invocar al método fuese un espacio en blanco (o sea: `' '`) en lugar de la cadena vacía, el resultado sería una cadena en la que todas las letras estarían separadas por un espacio en blanco. El siguiente script:

```
b = ['F', 'D', 'V', 'D']
encr = ' '.join(b)
print('Cadena:', encr)
```

produciría la siguiente salida:

```
Cadena: F D V D
```

La función `decode_cesar(encr, ABC, k)` que mostramos a continuación, es la encargada de desencriptar el mensaje almacenado en la cadena `encr` que toma como parámetro, usando el alfabeto `ABC` y el factor de desplazamiento `k` que también entran como parámetros:

```
def decode_cesar(encr, ABC, k):
    if not is_ok(encr, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(encr)):
        # si el carácter encriptado es blanco, dejar ese blanco...
        if encr[i] == ' ':
            b.append(' ')
        # si el carácter encriptado no es blanco, reemplazarlo...
        else:
            index = (ABC.index(encr[i]) - k) % na
            b.append(ABC[index])

    return b
```

```

else:
    # indice del caracter encr[i] en el alfabeto...
    ie = ABC.find(encr[i])

    # indice del caracter original en el alfabeto...
    io = (ie - k + na) % na
    b.append(ABC[io])

# convertir el arreglo b a cadena de caracteres y retornar...
mens = ''.join(b)
return mens

```

Esencialmente, esta función hace el proceso inverso del que hacía *code\_cesar()*: ahora cada carácter que aparece en la cadena *encr* es un carácter encriptado y debe ser reemplazado por el carácter original, que en el alfabeto *ABC* estará *k* posiciones *hacia atrás*. Si *ie* es el índice que el carácter *encr[i]* tiene en el alfabeto *ABC*, entonces la expresión *ie - k* obtendría el índice del carácter original en ese mismo alfabeto. Pero como también aquí el resultado de *ie - k* podría ser menor que cero, se aplica la misma idea de tabla circular. Si el tamaño del alfabeto es *na*, se puede mostrar con relativa sencillez que la expresión

$$\text{io} = (\text{ie} - \text{k} + \text{na}) \% \text{na}$$

asigna en la variable *io* el índice del carácter original que corresponde al carácter encriptado cuya posición en el alfabeto *ABC* es *ie*. Como el valor *ie - k* podría ser negativo, se suma el valor de *na* para ajustar el signo del resultado, sin alterar las propiedades de la aritmética modular. Por caso, si la letra encriptada fuese una 'B' (cuyo índice es *ie = 1* en el alfabeto) y el valor de *k* fuese *k = 5* con *na = 27*, entonces el índice de la letra original sería *io = (1 - 5 + 27) % 27 = 23 % 27 = 23* que corresponde a la letra 'W'. El resto de los detalles de la función *decode\_cesar()* no presenta mayor dificultad, y se dejan para el análisis del estudiante.

En cuanto a la *encriptación con Tabla de Transposición*, el mismo modelo *test04.py* del proyecto [F15] Arreglos que acompaña a esta Ficha incluye dos funciones similares a las que presentamos antes: una se llama *code\_transposition(mens, ABC, TRANS)* y la otra se llama *decode\_transposition(encr, ABC, TRANS)*. Se muestran ambas a continuación:

```

def code_transposition(mens, ABC, TRANS):
    if not is_ok(mens, ABC):
        return None

    b = []
    for i in range(len(mens)):
        # si el carácter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        else:
            # si el carácter original no es blanco, reemplazarlo...
            im = ABC.find(mens[i])
            b.append(TRANS[im])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr

def decode_transposition(encr, ABC, TRANS):
    if not is_ok(encr, ABC):
        return None

    b = []
    for i in range(len(encr)):
        # si el carácter encriptado es blanco, dejar ese blanco...

```

```

if encr[i] == ' ':
    b.append(' ')

# si el caracter encriptado no es blanco, reemplazarlo...
else:
    # si el caracter encriptado no es blanco, reemplazarlo...
    ie = TRANS.find(encr[i])
    b.append(ABC[ie])

# convertir el arreglo b a cadena de caracteres y retornar...
mens = ''.join(b)
return mens

```

En ambas funciones, el parámetro *ABC* es el alfabeto original, y el parámetro *TRANS* es la tabla de transposición propuesta (o sea, el mismo alfabeto original, pero mezclado). En la función *code\_transposition()* el parámetro *mens* es el mensaje a encriptar y la función retorna una cadena con ese mensaje encriptado. En la función *decode\_transposition()* el parámetro *encr* es el mensaje ya encriptado, y la función retorna una cadena con el mensaje original desencriptado. Ambas usan un arreglo auxiliar *b* para ir armando la secuencia de salida (en la misma forma que ya vimos para el *cifrado de César*), y ese arreglo se convierte a cadena de caracteres al final (antes de ser retornado) mediante el método *join()* (como también se explicó para el *cifrado de César*).

Tanto en una como en la otra no hay demasiado misterio: la primera función debe tomar cada carácter *mens[i]*, buscar su posición o índice en el alfabeto *ABC*, y reemplazar *mens[i]* (en el arreglo *b* de salida) por el carácter que esté exactamente en la misma posición de la tabla *TRANS*, como se muestra en el siguiente segmento:

```

im = ABC.find(mens[i])
b.append(TRANS[im])

```

Y la segunda función hace lo mismo, con el carácter *encr[i]*, pero al revés:

```

ie = TRANS.find(encr[i])
b.append(ABC[ie])

```

Confiamos en que el estudiante podrá analizar y comprender por sí mismo los detalles que restan en ambas funciones. Para finalizar, mostramos ahora el programa completo (incluyendo la función *test()* de entrada):

```

def validate(inf):
    k = inf
    while k <= inf:
        k = int(input('Ingrese valor de k (mayor a ' + str(inf) + ' por favor): '))
        if k <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return k

def is_ok(cad, ABC):
    if cad is None:
        return False

    if cad == '':
        return False

    for i in range(len(cad)):
        # si cad[i] no es blanco y no está en el alfabeto, retornar False...
        if cad[i] != ' ' and ABC.find(cad[i]) == -1:
            return False

    return True

```

```

def code_cesar(mens, ABC, k):
    if not is_ok(mens, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(mens)):
        # si el caracter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        # si el caracter original no es blanco, reemplazarlo...
        else:
            # indice del caracter mens[i] en el alfabeto...
            im= ABC.find(mens[i])

            # indice del caracter de reemplazo en el alfabeto...
            ir = (im + k) % na

            # efectuar el reemplazo en el arreglo de salida b...
            b.append(ABC[ir])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr


def decode_cesar(encr, ABC, k):
    if not is_ok(encr, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(encr)):
        # si el caracter encriptado es blanco, dejar ese blanco...
        if encr[i] == ' ':
            b.append(' ')

        # si el caracter encriptado no es blanco, reemplazarlo...
        else:
            # indice del caracter encr[i] en el alfabeto...
            ie = ABC.find(encr[i])

            # indice del caracter original en el alfabeto...
            io = (ie - k + na) % na
            b.append(ABC[io])

    # convertir el arreglo b a cadena de caracteres y retornar...
    mens = ''.join(b)
    return mens


def code_transposition(mens, ABC, TRANS):
    if not is_ok(mens, ABC):
        return None

    b = []
    for i in range(len(mens)):
        # si el caracter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        else:
            # si el caracter original no es blanco, reemplazarlo...
            im = ABC.find(mens[i])
            b.append(TRANS[im])

```

```

# convertir el arreglo b a cadena de caracteres y retornar...
enqr = ''.join(b)
return enqr

def decode_transposition(enqr, ABC, TRANS):
    if not is_ok(enqr, ABC):
        return None

    b = []
    for i in range(len(enqr)):
        # si el caracter encriptado es blanco, dejar ese blanco...
        if enqr[i] == ' ':
            b.append(' ')

        # si el caracter encriptado no es blanco, reemplazarlo...
        else:
            # si el caracter encriptado no es blanco, reemplazarlo...
            ie = TRANS.find(enqr[i])
            b.append(ABC[ie])

    # convertir el arreglo b a cadena de caracteres y retornar...
    mens = ''.join(b)
    return mens

def test():
    # el alfabeto general...
    ALFABETO = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"

    # Cifrado de César...
    # cargar el mensaje a encriptar (sin validación)...
    mensaje = input('Mensaje a encriptar (sólo mayúsculas y blancos...): ')
    print()

    print('Encriptación de César...')

    # cargar el valor del desplazamiento k...
    k = validate(1)

    # encriptar el mensaje con encriptación de César...
    encriptado = code_cesar(mensaje, ALFABETO, k)

    # desencriptar el mensaje encriptado (para comprobación)...
    desencriptado = decode_cesar(encriptado, ALFABETO, k)

    # si no hubo problemas, mostrar mensaje en claro y mensaje encriptado...
    if encriptado is not None:
        print('Mensaje original (tal como se cargó por teclado):', mensaje)
        print('Encriptado con clave k=', k, ': ', encriptado, sep='')
        print('Desencriptado (a partir del encriptado):', desencriptado)
    else:
        print('Error: posiblemente está mal el formato del mensaje original...')

    # Cifrado de Transposición...
    print()
    print('Encriptación de Transposición...')

    # una tabla de transposición para el alfabeto dado...
    TRANSPOSICION = "RISQPANOWXUMDHZTFGBLEYKCJVÑ"

    # encriptar el mensaje con encriptación de Transposición...
    encriptado = code_transposition(mensaje, ALFABETO, TRANSPOSICION)

    # desencriptar el mensaje encriptado (para comprobación)...
    desencriptado = decode_transposition(encriptado, ALFABETO, TRANSPOSICION)

```

```
# si no hubo problemas, mostrar mensaje en claro y mensaje encriptado...
if encryptado is not None:
    print('Mensaje original (tal como se cargó por teclado):', mensaje)
    print('Encriptado (tabla: ', TRANSPOSICION, '): ', encryptado, sep='')
    print('Desencriptado (a partir del encriptado):', desencriptado)
else:
    print('Error: posiblemente está mal el formato del mensaje original...')

# script principal...
if __name__ == '__main__':
    test()
```

---

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: <https://docs.python.org/3/>.
- [3] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

# Ficha 16

## Arreglos Bidimensionales

### 1.] Introducción.

Hemos visto la manera de usar arreglos unidimensionales para procesar conjuntos grandes de datos sin tener que declarar un número elevado de variables de tipo simple. Hemos visto también la gran importancia que tiene el concepto de *acceso directo* a los componentes de un arreglo (de hecho, la característica principal de un arreglo es el acceso directo).

Se define como *dimensión de un arreglo* a la cantidad de índices que se requieren para acceder a uno de sus elementos. En ese sentido, los problemas que hemos analizado para resolver usando arreglos eran de naturaleza *unidimensional*: en todos los casos, los datos (o los resultados) se almacenaban en un arreglo de forma que para luego accederlos era suficiente conocer *un índice* (*y sólo uno*). En ningún caso enfrentamos la necesidad de organizar los datos de forma de accederlos con dos o más índices.

Sin embargo es muy común que esa necesidad se haga presente. Piense el estudiante en las siguientes situaciones cotidianas: un organizador de horarios (o simplemente un "horario") por lo general es una *tabla* que tiene una fila (horizontal) por cada día de la semana, y una columna (vertical) por cada bloque de horas que tenga sentido. En cada casilla de esa tabla de días y horas, normalmente se anota la actividad que debe desarrollarse en un día y hora particular. La consulta de esa tabla se hace entrando por la fila de un día dado (que actúa a modo de *primer índice*) y por la columna de una hora dada (que se usa como *segundo índice*). Como se ve, la organización de las actividades de una persona requiere, en este caso, considerar *dos índices* de acceso a la tabla la cual resulta entonces *bidimensional* (también se dice que la tabla es de *dos entradas*) [1].

No es el único ejemplo: una persona que quiera hacer un resumen de sus gastos en el año, suele plantear una tabla de dos entradas. En cada fila escribe uno de los rubros o ítems en los cuales efectuó algún gasto, en cada columna escribe el nombre de un mes del año, y finalmente en cada intersección de la tabla formada anota los importes que gastó en cada rubro en cada mes. Esta tabla tendrá doce columnas (una por cada mes), y tantas filas como rubros quiera controlar la persona. Se suele llamar *orden de una tabla* al *producto expresado* de la cantidad de filas por la cantidad de columnas. Si los rubros a controlar fueran diez, entonces la tabla del ejemplo sería de *orden 10\*12* (observar que no importa tanto el resultado del producto, sino sólo dejarlo expresado).

La forma de implementar el concepto de *tabla bidimensional* o de *dos entradas* es usar *arreglos bidimensionales* (también llamados comúnmente *matrices*) y en Python esto implica la idea de *listas de listas* (variables de tipo *list* que en cada casilla contienen a otra *list*) [2]. Como veremos a lo largo de esta Ficha, la definición y uso de un arreglo de este tipo es una extensión natural del concepto de *arreglo unidimensional* ya estudiado.

## 2.] Creación y uso de arreglos bidimensionales en Python.

Básicamente, un *arreglo bidimensional* o *matriz* es un arreglo cuyos elementos están dispuestos en forma de tabla, con varias filas y columnas. Aquí llamamos *filas* a las disposiciones horizontales del arreglo, y *columnas* a las disposiciones verticales.

Hemos visto que en Python un arreglo unidimensional se implementa como una variable de tipo *list*. Pero sabemos también que una colección de tipo *list* puede contener otras *list* embebidas en ella, y este hecho es el que permite definir arreglos bidimensionales en Python. La forma más directa y simple de hacerlo es la *asignación directa de valores constantes*: Está claro que si sabemos de antemano qué valores necesitamos en cada *fila* y *columna*, podemos asignar en forma directa la lista de listas que finalmente será nuestra matriz [2]:

```
m0 = [[1, 3, 4],
      [3, 5, 2],
      [4, 7, 1]]
# se puede indentar aqui o en la columna del corchete de apertura...

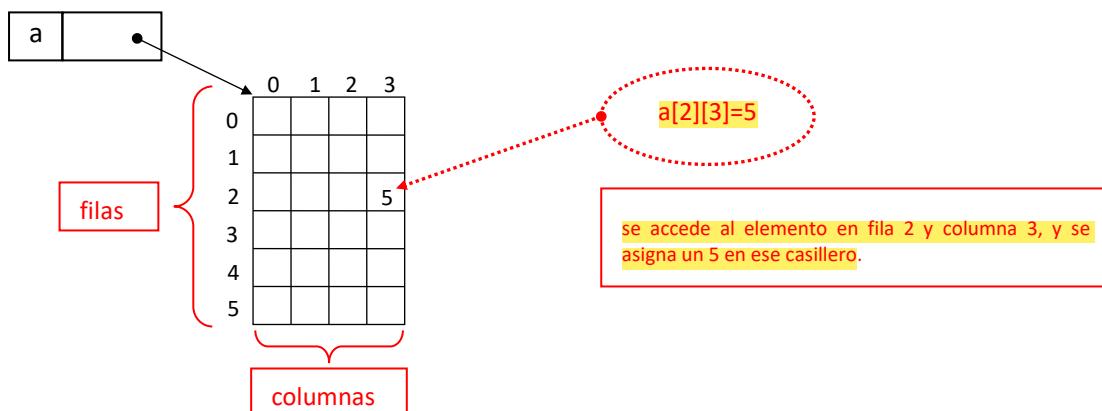
print('Matriz con valores fijos:', m0)
```

En el ejemplo anterior, la *lista de listas* se escribió en renglones separados e indentada, por razones de claridad, pero es totalmente equivalente a hacerlo en una sola línea:

```
m0 = [[1, 3, 4], [3, 5, 2], [4, 7, 1]]
print('Matriz con valores fijos:', m0)
```

Si la matriz está ya creada, para entrar a un componente debe darse el *índice de la fila del mismo* y también el *índice de la columna*. Como los índices requeridos son dos, el arreglo entonces es de dimensión dos. El siguiente esquema ilustra la manera de declarar y crear un arreglo *bidimensional de orden 6\*4* (conteniendo números enteros) en Python, la forma conceptual de representarlo, y la manera de acceder a uno de sus componentes [1]:

**Figura 1: Esquema de representación de una matriz de orden 6\*4.**



```
a = [[2, 3, 4, 5],
      [1, 5, 6, 3],
      [7, 2, 4, 0],
      [8, 4, 1, 7],
      [0, 0, 2, 9],
      [1, 3, 4, 1],]
```

```
a[2][3] = 5 # accede a una casilla y cambia su valor.
```

En el ejemplo anterior, suponemos que casillero de la matriz<sup>1</sup> queda valiendo los valores asignados, y luego específicamente el casillero  $a[2][3]$  cambia su valor inicial (que era un 0) por un 5.

Para lo que sigue, el proyecto *F[16] Arreglos Bidimensionales* que acompaña a esta Ficha contiene un primer modelo llamado *test01.py* en el cual se implementan todas las técnicas que mostraremos a continuación.

Si en lugar de una matriz con valores iniciales fijos queremos crear en Python una lista de listas que represente una *matriz de n filas y m columnas*, con  $n$  y  $m$  variables, el inconveniente es que Python es un lenguaje de tipado dinámico y no es posible indicarle al intérprete en forma directa y simple cuántos elementos queremos que reserve para nuestra matriz en cada dimensión. Eso lleva a que las *matrices* deban ser construidas de alguna forma en tiempo de ejecución [2] [3].

Hay varias formas de hacer esto. Comencemos por la más descriptiva (pero también la más larga y posiblemente la más ineficiente). Supongamos que queremos  $n$  filas (suponga  $n = 3$ ) y  $m$  columnas (suponga  $m = 4$ ). La idea es partir de una lista inicialmente vacía, y asignar en ella  $n$  listas vacías:

```
m1 = []
for f in range(n):
    m1.append([])
```

El aspecto que tendría la matriz  $m1$  en este momento es algo como:

```
m1 ⇒ [[], [], []]
```

Cada una de las  $n = 3$  listas vacías de  $m1$  puede entenderse como una de las filas de la matriz que estamos construyendo, y claramente se acceden como  $m1[0]$  la primera,  $m1[1]$  la segunda, y  $m1[2]$  la tercera.

El paso final, es agregar en cada una de las  $n$  listas  $m1[f]$  un total de  $m$  valores del tipo que se requiera, lo cual ciertamente "abrirá" espacio en cada fila generando las columnas buscadas. Si no estamos seguros de qué tipo de valores necesitaremos almacenar, podemos usar *None*:

```
m1 = []
for f in range(n):
    m1.append([])
    for c in range(m):
        m1[f].append(None)
```

---

<sup>1</sup> En este contexto una *matriz* es una colección de datos organizados en forma de tabla... pero en la famosísima película *Matrix* (de 1999) esa *matrix* (o *matriz*) es un mundo virtual al cual la humanidad está conectada sin saberlo. Las computadoras tomaron el control del mundo y vencieron a los seres humanos en una devastadora guerra, y ahora los tienen prisioneros en la gran simulación que es la *Matrix*, a la cual no sólo están sometidos sino que también alimentan con la energía de sus cuerpos. Un puñado de humanos libres lucha contra la *Matrix*, mientras espera la llegada de un mesías que cumplirá la profecía de derrotar a las máquinas y liberar a la especie humana (con lo que el argumento cae en un fastidioso lugar común...) La película fue dirigida por los hermanos *Wachowsky* en 1999 (hoy esos hermanos cambiaron de sexo y se conocen como las *hermanas Wachowsky*) y fue protagonizada por *Keanu Reeves*, *Laurence Fishburne* y *Carrie-Ann Moss*. Hubo dos conocidas secuelas: *The Matrix Reloaded* (de 2003) y *The Matrix Revolutions* (también de 2003) para conformar así una trilogía que ya se ha convertido en objeto de culto para sus seguidores.

La matriz así construida tendría el siguiente aspecto final, con  $n = 3$  y  $m = 4$ :

```
m1 ⇒ [[None, None, None, None], [None, None, None, None], [None, None, None, None]]
```

O bien:

```
m1 ⇒ [
    [None, None, None, None],
    [None, None, None, None],
    [None, None, None, None]
]
```

Si en lugar de  $n*m$  valores *None* se quisiera disponer de  $n*m$  valores *0(cero)*, lo único que debe hacerse es reemplazar el valor *None* del ejemplo anterior por el valor *0*.

Otra forma de crear la matriz, no tan detallada ni tan intuitiva pero más compacta y eficiente, consiste en comenzar creando las  $n$  filas con el operador de multiplicación, de forma que arranquen con valores *None* (o cero o lo que se requiera):

```
m2 = [None] * n # crea n componentes None (serán las filas...)
```

Si  $n = 3$ , el aspecto hasta aquí sería el siguiente, con los elementos *None* ya creados y por lo tanto con el espacio (y el índice...) ya asociado a ellos:

```
m2 ⇒ [None, None, None]
```

Para completar la matriz, *se itera sobre los  $n$  elementos ya creados, y se los reemplaza por una lista (creada con el operador multiplicación) de  $m$  elementos *None* o del tipo que se prefiera*. Como Python es de tipado dinámico, los valores *None* originalmente asignados en las  $n$  posiciones cambiarán a lo que sea que le indique [2] [3]:

```
m2 = [None] * n
for f in range(n):
    m2[f] = [None] * m # ...expande cada fila a 4 elementos None
```

El resultado es, otra vez, una matriz de  $n*m$  elementos *None*. Si en lugar de valores *None* se quisieran valores *0(cero)* o cualesquiera otros, solo se debe reemplazar el segundo *None* por *0* o por el valor que se quiera. En rigor, el primer *None* carece de importancia y podría en la práctica ser cualquier valor simple, ya que durante la corrida del ciclo *for* esos *None* serán reemplazados por las listas de tamaño  $m$ .

Finalmente, una tercera vía para *crear una matriz de  $n*m$*  elementos consiste en usar alguna *expresión de comprensión*, lo cual es aún más compacto y eficiente, aunque posiblemente menos claro. Si analizamos con atención el modelo de creación que acabamos de mostrar, notaremos que el ciclo *for* realiza una iteración de  $n$  giros, y en cada uno de esos giros crea una lista de  $m$  elementos *None*. En rigor, eso es todo lo que se necesita para crear la matriz, si la expresión comprensiva se encierra entre corchetes [2]:

```
m3 = [[None] * m for f in range(n)]
```

Note que en este caso no se requiere indicar en qué casillero se asigna la lista de tamaño  $m$  que se está creando: solo se crean  $n$  listas de tamaño  $m$ , y cada una de ellas se *inserta en la lista representada por los corchetes externos*. Cada una llevará un índice desde *0* en adelante, por orden de llegada. El resultado es el esperado: *una matriz de  $n * m$  elementos *None**. Reemplace el *None* por el valor que necesite, si fuese el caso.

Como vimos, una vez que la matriz ha sido creada el acceso a sus elementos individuales se hace con dos índices: el primero selecciona la "fila" (o sea, una de las  $n$  sublistas o subarreglos) y el segundo selecciona la "columna" (o sea, el elemento dentro de la sublista que fue seleccionada con el primer índice):

```
m1[0][3] = 10
m1[1][2] = 20
```

Observar que para acceder a un elemento, se escribe primero el nombre de la referencia al arreglo ( $m$  en el caso del ejemplo), luego el número de la *fila* del elemento que se quiere acceder, pero encerrado entre corchetes, y por último el número de la *columna* de ese elemento, también encerrado entre corchetes. Notar además, que en el lenguaje Python los arreglos de cualquier dimensión están *referidos a cero*, lo cual significa que el *primer índice de cada dimensión es siempre cero*. En la Figura 1 (página 322) puede verse que el arreglo tiene seis filas, pero numeradas del 0(cero) al 5(cinco), y cuatro columnas, numeradas del 0(cero) al 3(tres). No hay excepciones a esta regla, por lo cual debe tenerse cuidado de ajustar correctamente los ciclos para el recorrido de índices.

Si se quiere *recorrer por filas* en forma completa una matriz de  $n*m$  elementos, la idea es prácticamente igual a la forma de hacerlo en cualquier otro lenguaje. El siguiente ejemplo asigna en cada elemento el producto entre su número de fila y su número de columna:

```
# un recorrido por filas
for f in range(len(m2)):
    for c in range(len(m2[f])):
        m2[f][c] = f * c
```

El primer ciclo *for* recorre con la variable  $f$  el rango de índices de las "filas" (o sublistas) de la matriz. Note que la función *len()* aplicada sobre la variable  $m2$  que representa a la matriz completa, retornará la cantidad de sublistas (filas) que  $m2$  contiene. El segundo ciclo *for* recorre con la variable  $c$  el rango de índices de la sublista  $m2[f]$  (o sea, las "columnas" de esa fila). De nuevo, note que la expresión  $len(m2[f])$  retorna la cantidad de elementos de la sublista  $f$  (la cantidad de columnas de la fila  $f$ ). De hecho, con este planteo se podría recorrer sin problemas una matriz "dentada", cuyas filas tuviesen tamaños diferentes [1].

Ligeramente diferente es el problema si se quiere *recorrer la matriz por columnas*. Recuerde que en esencia, lo que tenemos es una lista de listas en la que las sublistas (primera dimensión) representan las filas, pero en la segunda dimensión *no tenemos otras listas* sino *directamente los elementos a acceder*. Para el recorrido por columnas, se deben invertir los dos ciclos *for*: llevar más afuera el que recorre las columnas (*for c*) y más adentro el que recorre las filas (*for f*). El tema es que ahora no podemos usar la función *len()* en el ciclo externo, ya que la fila a la que corresponde el índice  $c$  aún no ha sido seleccionada. En principio, debemos asumir que la matriz es regular (no dentada) y conocer de antemano la cantidad de columnas:

```
# recorrido por columnas - matriz regular
filas = 3
columnas = 4
for c in range(columnas):
    for f in range(filas):
        m3[f][c] = f * c
```

El recorrido completo de una matriz (por filas o por columnas) en definitiva equivale al *recorrido secuencial de esa matriz* y es un proceso muy común: el programador deberá aplicarlo cada vez que desee procesar todos y cada uno de los elementos de una matriz. Por ejemplo, la *carga por teclado de una matriz m4 de 3 filas y 4 columnas* se puede hacer mediante un recorrido del tipo que prefiera el programador. En este caso, aplicamos el *recorrido por filas* que ya hemos citado: dos ciclos *for* anidados, de forma que el primero recorra las filas de la matriz, y el segundo las columnas [1]:

```
# carga por teclado... recorrido por filas en orden creciente...
filas, columnas = 3, 4
m4 = [[0] * columnas for f in range(filas)]
for f in range(filas):
    for c in range(columnas):
        m4[f][c] = int(input('Valor: '))
print('Matriz 4 leida:', m4)
```

Otra vez, la idea básica del proceso es que la variable *f* del ciclo más externo se usa para indicar qué *fila* se está procesando en cada vuelta. Dado un valor de *f*, se dispara otro ciclo controlado por *c*, cuyo objetivo es el de recorrer todas las *columnas* de la fila indicada por *f*. Notar que mientras avanza el ciclo controlado por *c* permanece fijo el valor de *f*. Sólo cuando corta el ciclo controlado por *c*, se retorna al ciclo controlado por *f*, cambiando ésta de valor y comenzando por ello con una nueva fila. El proceso de recorrer secuencialmente una matriz avanzando fila por fila empezando desde la cero, como aquí se describe, se denomina *recorrido en orden de fila creciente*.

Como vimos, el mismo proceso de carga (o el que sea que requiera el programador) se puede hacer con recorridos de otros tipos, simplemente cambiando el orden de los ciclos. El siguiente esquema realiza un *recorrido en orden de fila decreciente*: comienza con la última fila, y barre cada fila hacia atrás hasta llegar a la fila cero [1]:

```
# carga por teclado... recorrido por filas en orden decreciente...
filas, columnas = 3, 4
m5 = [[0] * columnas for f in range(filas)]
for f in range(filas-1, -1, -1):
    for c in range(columnas):
        m5[f][c] = int(input('Valor: '))
print('Matriz 5 leida:', m5)
```

Notar que el cambio sólo consistió en hacer que la variable *f* (usada para barrer las filas), comience en el *valor filas-1* (que es el índice de la última fila), y se *decremente* hasta llegar a cero. El ciclo controlado por *c* se dejó como estaba.

Si se desea un *recorrido en orden de columna creciente* (*o decreciente*), sólo deben invertirse los ciclos: el ciclo que *recorre las columnas* (controlado por *c* en nuestro ejemplo) debe ir por fuera, y el ciclo que *recorre las filas* (controlado por *f* en este caso) debe ir por dentro. De esta forma, el valor de *c* no cambia hasta que el ciclo controlado por *f* termine todo su recorrido. Sin embargo, no debe olvidarse que si queremos que *c* indique una columna, entonces *c* debe usarse en el *segundo par de corchetes* al acceder a la matriz. Y si la variable *f* va a indicar filas, entonces debe usarse en el *primer par de corchetes*. Esto es independiente del orden en que se presenten los ciclos para hacer cada recorrido [1]:

**La variable que indica la fila va en el primer par de corchetes, y la variable que indica la columna va en el segundo par de corchetes, sin importar cuál ciclo va por fuera y cuál por dentro.**

El siguiente esquema muestra un recorrido en *orden de columna creciente*, para leer por teclado una matriz *m6* de *n* filas y *m* columnas:

```
# carga por teclado... por columnas en orden creciente...
filas, columnas = 3, 4
m6 = [[0] * columnas for f in range(filas)]
for c in range(columnas):
    for f in range(filas):
        m6[f][c] = int(input('Valor: '))
print('Matriz 6 leida:', m6)
```

### 3.] Totalización por filas y columnas.

En muchas situaciones se requerirá procesar los datos contenidos en una matriz, de forma de obtener los totales acumulados de cada una de sus filas y/o de cada una de sus columnas. Los procesos para realizar estas tareas son sencillos, y se deducen directamente de los procesos de recorrido por filas y por columnas que ya hemos analizado. Veamos un problema típico a modo de caso de análisis [1]:

**Problema 43.)** Un comercio mayorista trabaja con cierta cantidad *n* de artículos, numerados del 0 al *n-1*. Dispone de un plantel de *m* vendedores para su venta, los cuales están enumerados del 0 al *m-1* inclusive, en forma contigua. El gerente de dicho comercio desea obtener cierta información estadística respecto de las ventas realizadas en el mes. El programa que se pide, deberá cargar una matriz *cant*, de orden *m\*n*, en la que cada fila represente un vendedor, cada columna un artículo, y cada componente *cant[i][j]* almacene la cantidad del artículo *j* vendida por el vendedor *i*.

Se pide emitir un listado con las cantidades totales realizadas por cada vendedor y las cantidades totales que se vendieron de cada artículo.

**Discusión y solución:** En el proyecto F[16] Arreglos Bidimensionales que acompaña a esta Ficha, se incluye el modelo *test02.py* en el que se resuelve el problema propuesto.

En problema se pide efectuar una operación típica en el procesamiento de matrices: la *totalización por filas y/o columnas*. Usaremos una matriz *cant* en la que cada fila represente un vendedor, y cada columna un artículo. En cada casillero se guarda el total de unidades que cada vendedor vendió de cada artículo. Por lo tanto, si se quiere saber cuántos artículos en total vendió un vendedor, sólo se deben acumular los componentes de la fila de ese vendedor. Si esa operación se realiza para cada fila, se está haciendo una *totalización por filas*. En forma similar se procede para los artículos, haciendo una *totalización por columnas*. En el primer caso, se hace un proceso por *fila creciente* acumulando cada componente. Y en el segundo, es un proceso por *columna creciente*, también acumulando cada casillero.

La función *totales\_por\_vendedor()* realiza una *totalización por filas* de la matriz, acumulando los valores de cada fila y mostrándolos por consola de salida a medida que termina de acumular cada fila. Como siempre, en el proceso de *totalización por filas* que desarrolla la función, el ciclo que recorre las filas va por fuera, y el que recorre las columnas va por dentro:

```

def totales_por_vendedor(cant):
    # totalización por filas...
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades vendidas por cada vendedor')
    for f in range(m):
        ac = 0
        for c in range(n):
            ac += cant[f][c]
        print('Vendedor', f, '\t- Cantidad total vendida:', ac)

```

Del mismo modo, la función ***totales\_por\_articulo()*** realiza una *totalización por columnas*, **acumulando** los valores de cada columna y mostrando los resultados por pantalla directamente antes de cambiar de columna. Otra vez, notar que la *totalización por columnas* se logra ubicando por fuera el ciclo de las columnas, y por dentro el de las filas, pero el orden de los índices en los corchetes para acceder a la matriz es siempre el mismo: primero el índice de fila y segundo el índice de columna:

```

def totales_por_articulo(cant):
    # totalización por columnas...
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades totales vendidas de cada artículo')
    for c in range(n):
        ac = 0
        for f in range(m):
            ac += cant[f][c]
        print('Artículo', c, '\t- Cantidad total vendida:', ac)

```

El programa completo se muestra a continuación:

```

def validate(inf):
    t = inf
    while t <= inf:
        t = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
        if t <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return t

def read(m, n):
    # crear y cargar por teclado una matriz... filas en orden creciente...
    cant = [[0] * n for f in range(m)]
    for f in range(m):
        for c in range(n):
            cant[f][c] = int(input('Valor [' + str(f) + '][' + str(c) + ']: '))
    return cant

def totales_por_vendedor(cant):
    # totalización por filas...
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades vendidas por cada vendedor')
    for f in range(m):
        ac = 0
        for c in range(n):
            ac += cant[f][c]
        print('Vendedor', f, '\t- Cantidad total vendida:', ac)

def totales_por_articulo(cant):
    # totalización por columnas...

```

```

m, n = len(cant), len(cant[0])
print()
print('Cantidades totales vendidas de cada artículo'):
for c in range(n):
    ac = 0
    for f in range(m):
        ac += cant[f][c]
    print('Artículo', c, '\t- Cantidad total vendida:', ac)

def test():
    print('Cantidad de vendedores...')
    m = validate(0)

    print('Cantidad de artículos...')
    n = validate(0)

    print('Cargue las cantidades de artículos por vendedor...')
    cant = read(m, n)

    totales_por_vendedor(cant)
    totales_por_articulo(cant)

if __name__ == '__main__':
    test()

```

#### 4.] Matrices de conteo y/o acumulación.

Así como en muchas situaciones prácticas se puede usar un **arreglo unidimensional** de forma que cada casilla actúe como un **contador** o como un **acumulador** (usándolo entonces como un **vector de conteo** o un **vector de acumulación**), también es posible que en ciertas ocasiones se deba emplear una matriz de contadores o de acumuladores. En algunos problemas se necesita seleccionar un contador entre muchos posibles, pero esa selección debe hacerse en base a dos variables que actúan como índices. El siguiente problema es un ejemplo típico:

**Problema 44.)** *Se desea almacenar en dos arreglos paralelos la información de los  $n$  clientes de una compañía de viajes que adquirieron algún viaje con esa empresa. En el primer arreglo se almacena en cada casillero un número entre 0 y 4 que indica el destino del viaje, y en el segundo arreglo se almacena otro número pero ahora entre 0 y 2 que indica la forma de pago que usó ese cliente. Se desea saber cuántos clientes viajaron a cada destino posible usando cada forma de pago disponible (es decir: cuántos clientes que viajaron al destino 0 usaron la forma de pago 0; cuántos clientes que viajaron al destino 0 usaron la forma de pago 1, y así sucesivamente. En total, se necesitan entonces  $5 \times 3 = 15$  contadores, pues los destinos posibles son 5, y las formas de pago posibles son 3).*

**Discusión y solución:** En el proyecto F[16] Arreglos Bidimensionales que acompaña a esta Ficha, el modelo *test03.py* resuelve el problema propuesto.

Para resolver el conteo pedido en este ejercicio, se usa el concepto de *matriz de conteos*. Una *matriz de conteos* es análoga a un *vector de conteos*, con la diferencia que en la matriz de conteos se requieren *dos índices* para seleccionar el casillero que hará las veces de contador [1].

En este ejercicio, la matriz de conteos es usada para contar cuántos clientes que viajan a cada destino posible, usaron cada forma de pago posible. Como las formas de pago

disponibles son tres, y los destinos son cinco, hay entonces un total de quince combinaciones, cada una de las cuales requiere un contador. Como cada uno de los quince contadores debe seleccionarse con dos códigos (uno para la forma de pago y otro para el destino), se evidencia la *naturaleza bidimensional* del proceso de conteo.

Simplemente, se define una matriz de 5\*3 elementos, para que cada uno de ellos funcione como un contador. Cada fila representa un destino de viaje y cada columna representa una forma de pago. En este programa, la matriz *conteo* se crea en la función *count()*, y luego para cada cliente se toma su destino de viaje (del primer arreglo) y la forma de pago que usó (del segundo arreglo). Ambos valores se usan como índices para acceder en forma directa al casillero correspondiente en la matriz, y se procede a contar en ese casillero:

```
def count(destinos, formas):
    conteo = [[0] * 3 for f in range(5)]

    n = len(destinos)
    for i in range(n):
        f = destinos[i]
        c = formas[i]
        conteo[f][c] += 1

    return conteo
```

La función *display\_count()* muestra los resultados obtenidos, considerando sólo aquellos casilleros que hayan quedado con valor diferente de cero:

```
def display_count(conteo):
    filas, columnas = len(conteo), len(conteo[0])
    print()
    print('Conteo de clientes por destino y forma de pago')
    for f in range(filas):
        for c in range(columnas):
            if conteo[f][c] != 0:
                print('Destino', f, '\tForma', c, '\tCantidad:', conteo[f][c])
```

El programa completo se muestra a continuación. Dejamos el análisis del resto de los detalles para el estudiante.

```
def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def validate_range(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor (entre ' + str(inf) + ' y ' + str(sup)+ '): '))
        if n < inf or n > sup:
            print('Se pidió entre', inf, 'y', sup, '... cargue de nuevo...')
    return n

def read(destinos, formas):
    n = len(destinos)
    for i in range(n):
        print('Destino del viaje -', end=' ')
        destinos[i] = validate_range(0, 4)
        print('Forma de pago -', end=' ')
```

```

formas[i] = validate_range(0, 2)
print()

def display(destinos, formas):
    n = len(destinos)
    print('Datos de los clientes registrados:')
    for i in range(n):
        print('Destino[' + str(i) + ']: ' + destinos[i], sep=' ', end=' - ')
        print('Forma de pago:', formas[i])

def count(destinos, formas):
    conteo = [[0] * 3 for f in range(5)]

    n = len(destinos)
    for i in range(n):
        f = destinos[i]
        c = formas[i]
        conteo[f][c] += 1

    return conteo

def display_count(conteo):
    filas, columnas = len(conteo), len(conteo[0])
    print()
    print('Conteo de clientes por destino y forma de pago')
    for f in range(filas):
        for c in range(columnas):
            if conteo[f][c] != 0:
                print('Destino', f, '\tForma', c, '\tCantidad:', conteo[f][c])

def test():
    # cargar cantidad de clientes...
    print('Cantidad de clientes -', end=' ')
    n = validate(0)
    print()

    # crear y cargar los arreglos paralelos...
    destinos = n * [0]
    formas = n * [0]
    read(destinos, formas)
    print()

    # mostrar todos los clientes...
    display(destinos, formas)
    print()

    # contar por destino y forma de pago...
    conteo = count(destinos, formas)

    # mostrar por pantalla el listado...
    display_count(conteo)

# script principal...
if __name__ == '__main__':
    test()

```

## 5.] Diagonalización de matrices cuadradas.

Un interesante ejercicio de aplicación sobre matrices cuadradas [\[1\]](#) puede usarse para cerrar esta ficha. Mostramos directamente el enunciado y la solución propuesta.

**Problema 45.)** Cargar por teclado una matriz cuadrada de números enteros, de orden  $n*n$ , y desarrollar las siguientes operaciones en ella:

- Acumular los elementos ubicados en el triángulo superior de la matriz (es decir, los elementos ubicados por encima de la diagonal principal).
- Determinar cuántos elementos de la diagonal principal valen 0(cero).
- Determinar cuántos elementos ubicados en el triángulo inferior de la matriz (o sea, los elementos ubicados debajo de la diagonal principal) son pares.
- Mostrar la matriz.

**Discusión y solución:** En una matriz cuadrada pueden definirse tres zonas a partir de la diagonal principal (que en el gráfico siguiente se marca resaltada en gris):

Figura 2: La diagonal principal y los triángulos superior e inferior en una matriz cuadrada.

d	s	s	s	s
i	d	s	s	s
i	i	d	s	s
i	i	i	d	s
i	i	i	i	d

- ✓ Los elementos que pertenecen a la *diagonal principal* se marcan aquí con una letra *d*.
- ✓ Los elementos ubicados en el *triángulo superior* se marcan aquí con una letra *s*.
- ✓ Los elementos ubicados en el *triángulo inferior* se marcan aquí con una letra *i*.

El acceso a los elementos de la *diagonal principal* es directo si se observa que todos esos elementos tienen el *índice de fila igual al índice de columna*. La función *diagonal()* tiene el objetivo de recorrer esa diagonal y contar cada uno de los casilleros que contenga un cero. Sólo es necesario *un ciclo*, y *repetir el índice en los dos pares de corchetes* al acceder a la matriz:

```
def diagonal(mat):
    cc, n = 0, len(mat)
    for f in range(1, n):
        if mat[f][f] == 0:
            cc += 1
    return cc
```

El acceso a los elementos del *triángulo superior* es un poco más complicado, pero resulta sencillo si se analiza lo siguiente: hay elementos del triángulo superior en todas las filas, *salvo en la última* (ver gráfico). Es decir que el ciclo que recorra las filas de la matriz no tiene necesidad de llegar hasta la fila  $n-1$  (que es la última). El ciclo para recorrer las filas comenzará colocando la variable *f* en cero, y continuará mientras *f* se mantenga menor que  $n-1$ . Y por otra parte, notemos que en la fila 0 el primer elemento del triángulo superior está en la columna 1. En la fila 2 el primero está en la columna 3... Puede verse fácilmente que en cada fila *f* el primer elemento del triángulo superior se encuentra en la columna *f+1*. Por eso, el ciclo que recorra las columnas (con variable de control *c*) no debe comenzar desde 0, sino desde *f+1*, y llegar en todos los casos hasta la última columna. Si los ciclos se ajustan de esta forma, solo se recorrerán los elementos que pertenecen al *triángulo superior*, sin tocar ni perder tiempo en los elementos que están fuera de él. La función *upper\_triangle()* acumula los elementos del *triángulo superior* aplicando estos principios de recorrido:

```
def upper_triangle(mat):
    ac, n = 0, len(mat)
    for f in range(n-1):
        for c in range(f+1, n):
            ac += mat[f][c]
    return ac
```

Un análisis parecido permite ajustar los ciclos de recorrido para atravesar el *triángulo inferior*: todas las filas tienen elementos del *triángulo inferior*, salvo la *primera*. El ciclo para barrido de filas debe comenzar entonces con la variable de control *f* valiendo 1. Por otra parte, en la fila 1 el último elemento del triángulo inferior está en la columna 0. En la fila 2 el último está en la columna 1, y así sucesivamente: el ciclo que recorra las columnas (con variable de control *c*), debe comenzar entonces en 1 y proseguir mientras *c* se mantenga menor a *f*. La función *lower\_triangle()* aplica estos principios para recorrer el *triángulo inferior* y contar los valores que sean pares del triángulo inferior:

```
def lower_triangle(mat):
    cp, n = 0, len(mat)
    for f in range(1, n):
        for c in range(0, f):
            if mat[f][c] % 2 == 0:
                cp += 1
    return cp
```

La visualización por pantalla de la matriz completa, pero de forma que cada fila aparezca a renglón seguido de la fila anterior, es realizada por la función *write()*, cuya estructura es muy simple:

```
def write(mat):
    n = len(mat)
    for f in range(n):
        print(mat[f])
```

El ciclo *for* de la función recorre todas las filas de la matriz *mat*, y en cada giro de ese ciclo sólo es necesario mostrar con *print()* la fila *f* completa (*print(mat[f])*), sin tener que usar otro ciclo para recorrerla: la fila *mat[f]* es ella misma una variable de tipo *list*, y la función *print()* ya está diseñada para mostrar correctamente toda una lista. El programa completo se ve a continuación (modelo *test04.py* en el proyecto [F16] Arreglos Bidimensionales que acompaña a esta ficha):

```
def validate(inf):
    t = inf
    while t <= inf:
        t = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
        if t <= inf:
            print('Error: se pidió > a', inf, '... cargue de nuevo...')
    return t

def read(n):
    # crear y cargar por teclado una matriz cuadrada...
    mat = [[0] * n for f in range(n)]
    for f in range(n):
        for c in range(n):
            mat[f][c] = int(input('Valor [' + str(f) + '][' + str(c) + ']: '))
    return mat

def write(mat):
    n = len(mat)
```

```

for f in range(n):
    print(mat[f])

def upper_triangle(mat):
    ac, n = 0, len(mat)
    for f in range(n-1):
        for c in range(f+1, n):
            ac += mat[f][c]
    return ac

def lower_triangle(mat):
    cp, n = 0, len(mat)
    for f in range(1, n):
        for c in range(0, f):
            if mat[f][c] % 2 == 0:
                cp += 1
    return cp

def diagonal(mat):
    cc, n = 0, len(mat)
    for f in range(1, n):
        if mat[f][f] == 0:
            cc += 1
    return cc

def test():
    print('Orden de la matriz cuadrada...')
    n = validate(0)
    print()
    print('Cargue la matriz...')
    mat = read(n)
    print()
    print('Contenido de la matriz:')
    write(mat)
    r1 = upper_triangle(mat)
    r2 = diagonal(mat)
    r3 = lower_triangle(mat)
    print('Acumulación del triángulo superior:', r1)
    print('Cantidad de ceros en la diagonal:', r2)
    print('Cantidad de pares en el triángulo inferior:', r3)

if __name__ == '__main__':
    test()

```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 17

## Ordenamiento

### 1.] Algoritmos de ordenamiento. Clasificación tradicional.

Es claro que pueden existir muchos algoritmos diferentes para resolver el mismo problema, y el problema de la *ordenación de un arreglo* es quizás el caso emblemático de esa situación. Podemos afirmar que existen varias docenas de algoritmos diferentes para lograr que el contenido de un arreglo se modifique para dejarlo ordenado de menor a mayor, o de mayor a menor. Muchos de esos algoritmos están basados en ideas intuitivas muy simples, y otros se fundamentan en estrategias lógicas muy sutiles y no tan obvias a primera vista. En general, se suele llamar *algoritmo simples* o *algoritmos directos* a los del primer grupo, y *algoritmos compuestos* o *algoritmos mejorados* a los del segundo, aunque la diferencia real entre ambos no es sólo conceptual, sino que efectivamente existe una diferencia de rendimiento muy marcada en cuanto al tiempo que los algoritmos de cada grupo demoran en terminar de ordenar el arreglo [1]. Más adelante profundizaremos la cuestión del *análisis comparativo* del rendimiento de algoritmos, y justificaremos formalmente la diferencia entre los dos grupos, pero por ahora nos concentraremos sólo en la clasificación de los algoritmos y el funcionamiento de aquellos que estén al alcance de este curso.

Tradicionalmente, como dijimos, los algoritmos de ordenamiento se suelen clasificar en los dos grupos ya citados, y en cada grupo se encuentran los siguientes (aunque entienda: la tabla siguiente es sólo una forma clásica de presentar a los algoritmos más conocidos, pero estos no son todos los algoritmos que existen... que son varias docenas...) [1]:

**Figura 1:** Clasificación tradicional de los algoritmos de ordenamiento más comunes.

Algoritmos Simples o Directos	Algoritmos Compuestos o Mejorados
Intercambio Directo (Burbuja)	Método de Ordenamiento Rápido ( <i>Quicksort</i> ) [C. Hoare - 1960]
Selección Directa	Ordenamiento de Montículo ( <i>Heapsort</i> ) [J. Williams – 1964]
Inserción Directa	Ordenamiento por Incrementos Decrecientes ( <i>Shellsort</i> ) [D. Shell – 1959]

En principio, los algoritmos clasificados como *Simples* o *Directos* son algoritmos sencillos e intuitivos, pero de mal rendimiento si la cantidad  $n$  de elementos del arreglo es grande o muy grande, o incluso si lo que se desea es ordenar un archivo en disco. Aquí, *mal rendimiento* por ahora significa "demasiada demora esperada". En cambio, los métodos presentados como *Compuestos* o *Mejorados* tienen un muy buen rendimiento en comparación con los simples, y se aconsejan toda vez que el arreglo sea muy grande o se quiera ordenar un conjunto en memoria externa. Si el tamaño  $n$  del arreglo es pequeño (por ejemplo, no más de 100 o 150 elementos), los métodos simples siguen siendo una buena elección por cuanto su escasa eficiencia no es notable con conjuntos pequeños. Note que la idea final, es que cada algoritmo compuesto mostrado en esta tabla representa en realidad una mejora en el planteo del algoritmo simple de la misma fila, y por ello se los llama "algoritmos mejorados". Así, el algoritmo *Quicksort* es un replanteo del algoritmo de *intercambio directo* (más conocido como *ordenamiento de burbuja*) para eliminar los

elementos que lo hacen ineficiente. Paradójicamente, el *ordenamiento de burbuja* o *bubblesort* es en general el de *peor rendimiento* para ordenar un arreglo, pero ha dado lugar al *Quicksort*, que en general es el de *mejor rendimiento promedio* conocido.

## 2.] Funcionamiento de los Algoritmos de Ordenamiento Simples o Directos.

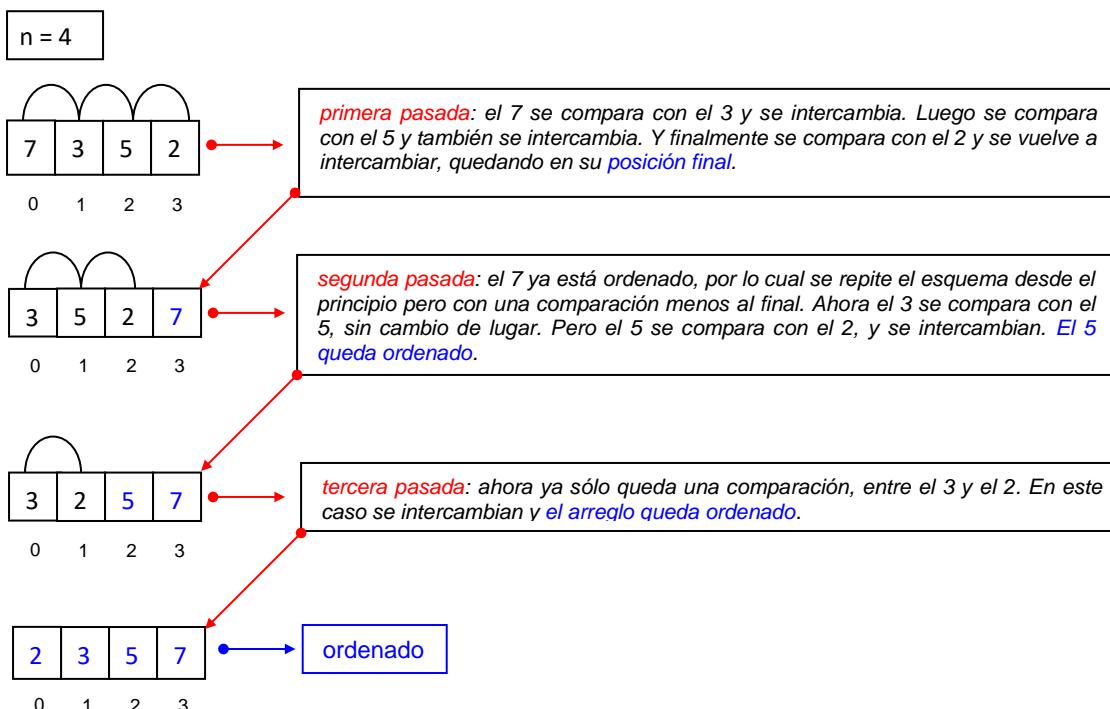
Haremos aquí una breve descripción operativa de las estrategias de funcionamiento de los algoritmos directos conocidos como *Intercambio Directo* (o *Burbuja* o *Bubblesort*) e *Inserción Directa*. Para una descripción del algoritmo de *Selección Directa*, revise la Ficha 14. Más adelante, en una Ficha posterior, mostraremos un análisis de rendimiento formal basado en calcular la cantidad de comparaciones que estos algoritmos realizan.

En todos los casos, suponemos un arreglo  $v$  de  $n$  elementos, y también suponemos que se pretende ordenar de menor a mayor. Hemos incluido un modelo llamado *test03.py* en el proyecto [F17] *Ordenamiento*, que acompaña a esta Ficha. En el mismo proyecto se incluye el módulo *ordenamiento.py*, que contiene todas las funciones que implementan estos algoritmos.

### a.) Ordenamiento por Intercambio Directo (o Bubblesort):

La idea esencial del algoritmo es que cada elemento en cada casilla  $v[i]$  se compara con el elemento en  $v[i+1]$ . Si este último es menor, se intercambian los contenidos. Se usan dos ciclos anidados para conseguir que incluso los elementos pequeños ubicados muy atrás, puedan en algún momento llegar a sus posiciones al frente del arreglo. Gráficamente, supongamos que el tamaño del arreglo es  $n = 4$ . El algoritmo procede como se muestra en la Figura 2.

Figura 2: Funcionamiento general del *Ordenamiento por Intercambio Directo (Bubblesort)*.



Puede verse que si el arreglo tiene  $n$  elementos, serán necesarias a lo sumo  $n-1$  pasadas para terminar de ordenarlo en el peor caso, y que en cada pasada se hace

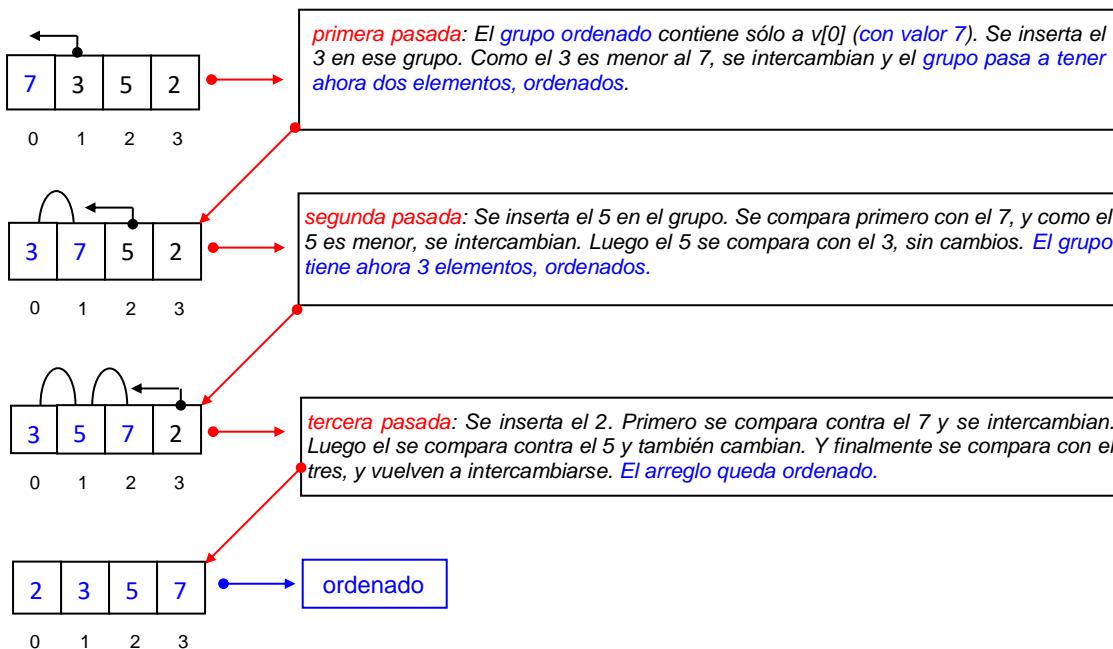
cada vez una comparación menos que en la anterior. Otro hecho notable es que el arreglo podría quedar ordenado *antes de la última pasada*. Por ejemplo, si el arreglo original hubiese sido [2 – 3 – 7 – 5], entonces en la primera pasada el 7 cambiaría con el 5 y dejaría al arreglo ordenado. Para detectar ese tipo de situaciones, en el algoritmo se agrega una variable a modo de **bandera de corte**: si se detecta que en una pasada no hubo ningún intercambio, el ciclo que controla la cantidad de pasadas se interrumpe antes de llegar a la pasada  $n-1$  y el ordenamiento se da por concluido. Se ha denominado *ordenamiento de burbuja* porque los elementos parecen *burbujejar* en el arreglo a medida que se ordena... (⊗)... La siguiente función del módulo *ordenamiento.py* (proyecto [F17] *Ordenamiento*) lo implementa:

```
def bubble_sort(v):
    n = len(v)
    for i in range(n-1):
        ordenado = True
        for j in range(n-i-1):
            if v[j] > v[j+1]:
                ordenado = False
                v[j], v[j+1] = v[j+1], v[j]
        if ordenado:
            break
```

#### b.) Ordenamiento por *Inserción Directa* (o *Inserción Simple*):

La idea es ahora distinta y más original. Se comienza suponiendo que el valor en la casilla  $v[0]$  conforma un subconjunto. Y como tiene un solo elemento, está ordenado. A ese subconjunto lo llamamos un *grupo ordenado* dentro del arreglo. Se toma  $v[1]$  y se trata de insertar su valor en el grupo ordenado. Si es menor que  $v[0]$  se intercambian, y si no, se dejan como estaban. En ambos casos, el grupo tiene ahora dos elementos y sigue ordenado. Se toma  $v[2]$  y se procede igual, comenzando la comparación contra  $v[1]$  (que es el mayor del grupo). Así, también hacen falta  $n-1$  pasadas, de forma que en cada pasada se inserta un nuevo valor al grupo. El algoritmo procede como se muestra en la *Figura 3*:

Figura 3: Funcionamiento general del Algoritmo de *Ordenamiento por Inserción Directa*.



La función que sigue (ver módulo *ordenamiento.py* del proyecto [F17] *Ordenamiento*) lo implementa:

```
def insertion_sort(v):
    n = len(v)
    for j in range(1, n):
        y = v[j]
        k = j - 1
        while k >= 0 and y < v[k]:
            v[k+1] = v[k]
            k -= 1
        v[k+1] = y
```

### 3.] Algoritmos de Ordenamiento Compuestos o Mejorados: El Algoritmo Shellsort.

Como dijimos, los algoritmos designados como *Compuestos* o *Mejorados* están basados en la idea de mejorar algunos aspectos que hacen que los algoritmos directos no tengan buen rendimiento cuando el tamaño del arreglo es grande o muy grande. De los tres algoritmos que hemos mencionado en la tabla de la *Figura 1* (*Quicksort*, *Heapsort* y *Shellsort*), analizaremos en esta sección sólo el mecanismo de funcionamiento de trabajo del *Shellsort*, que es el más simple en cuanto a su estructura y fundamentos.

El algoritmo *Quicksort* aplica una estrategia recursiva conocida como *Divide y Vencerás* y será analizado con detalle cuando se exponga esa estrategia en una Ficha posterior.

Finalmente, el algoritmo *Heapsort* se basa en el uso de una estructura de datos conocida como *Heap* (o *Grupo de Ordenamiento*) que escapa a los contenidos mínimos de este curso. No obstante, por si algún estudiante siente curiosidad por la forma general de funcionamiento de este algoritmo, lo hemos incluido en esta Ficha en forma de tema **totalmente opcional** (no será evaluado ni exigido en ninguna de las evaluaciones de ningún tipo previstas para esta asignatura, aunque alguna pregunta podría surgir en el Cuestionario asociado a esta Ficha).

De nuevo, suponemos en todos los casos un arreglo *v* de *n* elementos, y ordenamiento de menor a mayor.

#### a.) *Ordenamiento de Shell (Shellsort u Ordenamiento por Incrementos Decrecientes):*

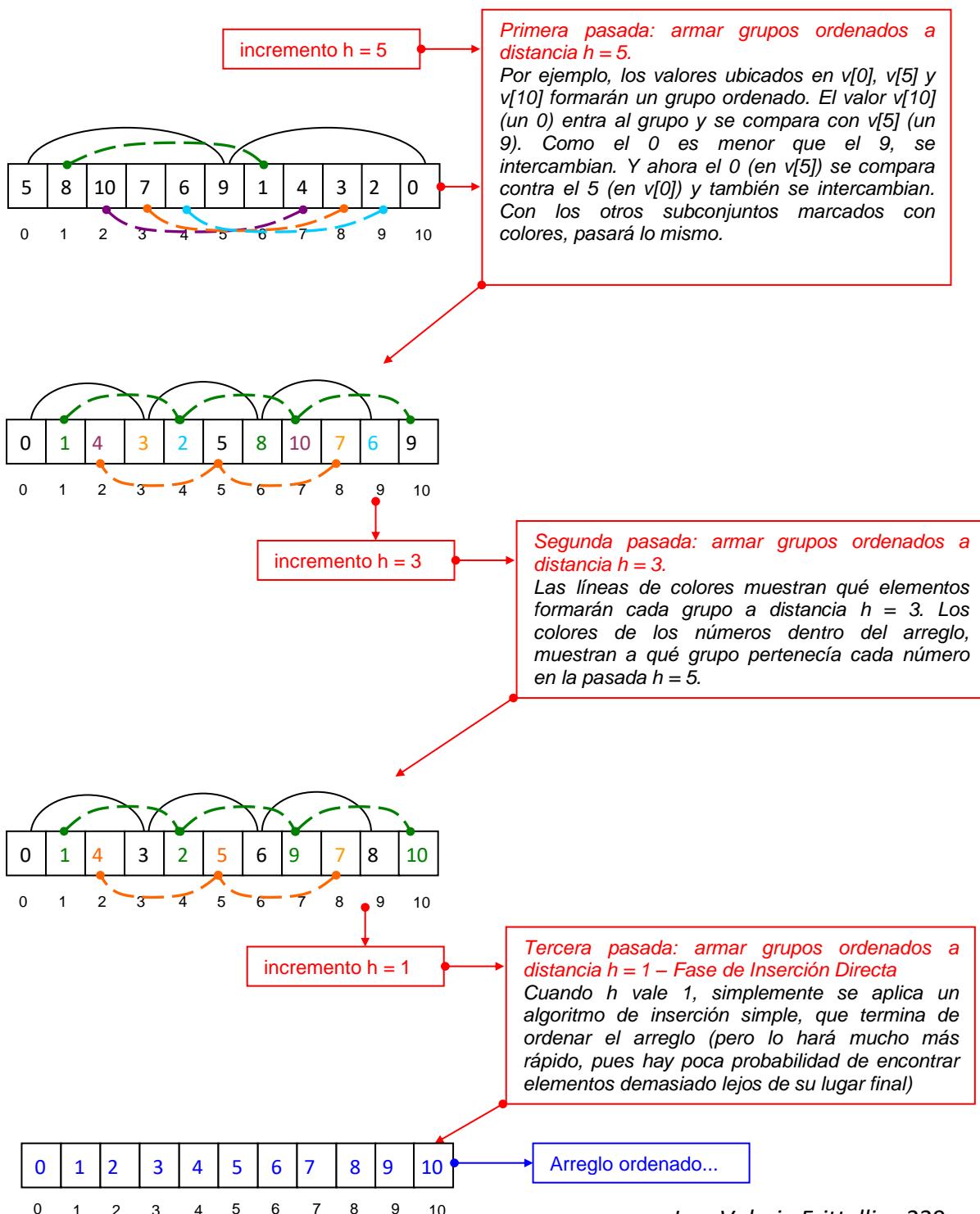
El algoritmo de ordenamiento por *Inserción Directa* es el más rápido de los métodos simples, pues aprovecha que un subconjunto del arreglo está ya ordenado y simplemente inserta nuevos valores en ese conjunto de forma que el subconjunto siga ordenado. El problema es que si llega a aparecer un elemento muy pequeño en el extremo derecho del arreglo, cuando el grupo ordenado de la izquierda ya contiene a casi todo el vector, la inserción de ese valor pequeño demorará demasiado, pues tendrá que compararse con casi todo el arreglo para llegar a su lugar final.

En 1959 un técnico de la *General Electric Company* llamado *Donald Shell*, publicó un algoritmo que mejoraba al de inserción directa y lo llamó *High-Speed Sorting Procedure*, aunque en honor a su creador se lo terminó llamando *Shellsort*. La idea es lanzar un proceso de ordenamiento por inserción, pero en lugar de hacer que cada valor que entra al grupo se compare con su vecino inmediato a la izquierda, se comience haciendo primero un reacomodamiento de forma que cada elemento del arreglo se compare contra elementos ubicados más lejos, a distancias mayores que

uno, y se intercambien elementos a esas distancias [2]. Luego, en pasadas sucesivas, las *distancias de comparación* se van acortando y repitiendo el proceso con elementos cada vez más cercanos unos a otros. De esta forma, se van armando grupos ordenados pero no a distancia uno, sino a distancia  $h$  tal que  $h > 1$ .

Finalmente, se termina tomando una distancia de comparación igual a uno, y en ese momento el algoritmo se convierte lisa y llanamente en una *Inserción Directa* para terminar de ordenar el arreglo. Las distancias de comparación se denominan en general *incrementos decrecientes*, y de allí el nombre con que también se conoce al método [2]. En la Figura 4 mostramos la idea con un pequeño arreglo, suponiendo incrementos decrecientes de la forma [5 – 3 – 1].

Figura 4: Esquema general de funcionamiento del Algoritmo Shellsort.



No es simple elegir los valores de los incrementos decrecientes, y de esa elección depende muy fuertemente el rendimiento del algoritmo. En general, digamos que no es necesario que esos incrementos sean demasiados: suele bastar con una cantidad de distancias igual o menor al 10% del tamaño del arreglo, pero *debe asegurarse siempre que la última sea igual uno*, pues de lo contrario no hay garantía que el arreglo quede ordenado. Por otra parte, es de esperar que los valores elegidos como distancias de comparación *no sean todos múltiplos entre ellos*, pues si así fuera se estarían comparando siempre las mismas subsecuencias de elementos, sin mezclar nunca esas subsecuencias. Sin embargo, no es necesario que los valores de los incrementos decrecientes sean todos necesariamente primos. Es suficiente con garantizar valores que no sean todos múltiplos entre sí.

La función que sigue implementa el *algoritmo de Shell* [2]. Está incluida en el módulo *ordenamiento.py* del proyecto [F17] *Ordenamiento*.

```
def shell_sort(v):
    n = len(v)
    h = 1
    while h <= n // 9:
        h = 3*h + 1

    while h > 0:
        for j in range(h, n):
            y = v[j]
            k = j - h
            while k >= 0 and y < v[k]:
                v[k+h] = v[k]
                k -= h
            v[k+h] = y
        h //= 3
```

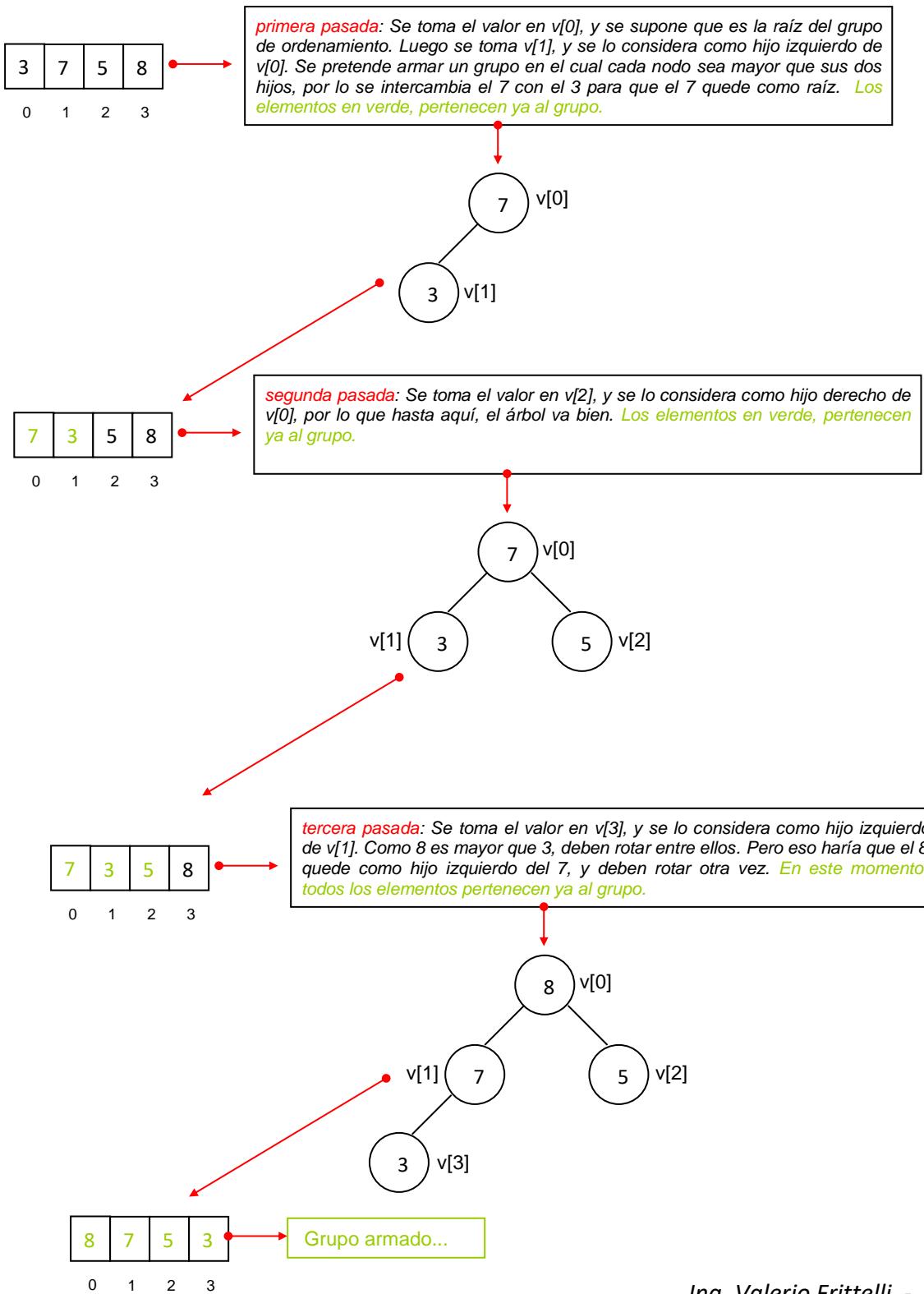
#### b.) El Algoritmo Heapsort.

El método de ordenamiento por Selección Directa realiza  $n-1$  pasadas, y el objetivo de cada una es seleccionar el menor de los elementos que sigan sin ordenar en el arreglo y trasladarlo a la casilla designada como pivot. El problema es que la búsqueda del menor en cada pasada se basa en un proceso secuencial, y exige demasiadas comparaciones. En 1964, un estudiante de Ciencias de la Computación, llamado J. Williams, publicó una mejora para el algoritmo de Selección Directa en *Communications of the ACM*, y llamó al mismo *Ordenamiento de Montículos* o *Heapsort* (en realidad, debería traducirse como Ordenamiento de Grupos de Ordenamiento, pero queda redundante...) Ese algoritmo reduce de manera drástica el tiempo de búsqueda o selección del menor en cada pasada, usando una estructura de datos conocida como *grupo de ordenamiento* (que no es otra cosa que una *cola de prioridades*, pero optimizada en velocidad: los elementos se insertan rápidamente, ordenados de alguna forma, pero cuando se extrae alguno, se extrae el menor [o el mayor, según las necesidades]) Igual que antes, al seleccionar el menor (o el mayor) se lo lleva a su lugar final del arreglo. Luego se extrae el siguiente menor (o mayor), se lo reubica, y así se prosigue hasta terminar de ordenar [1].

En esencia, un *grupo de ordenamiento* es un árbol *binario casi completo* (todos los nodos hasta el anteúltimo nivel tienen dos hijos, a excepción de los nodos más a la derecha en ese nivel que podrían no tener los dos hijos) en el cual el valor de cada nodo es mayor que el valor de sus dos hijos. La idea es comenzar con todos los

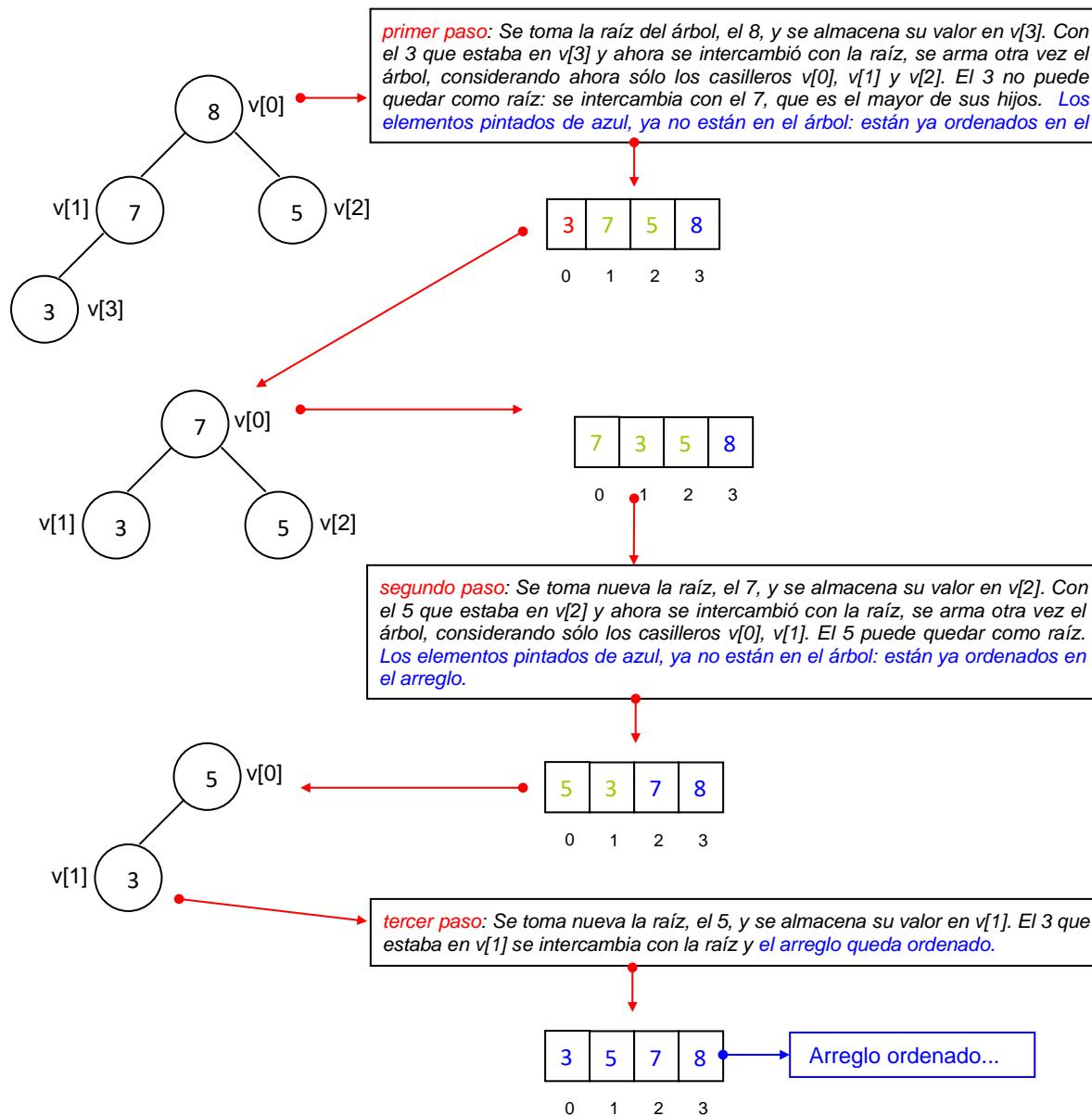
elementos del arreglo, y formar un árbol de este tipo, pero dentro del mismo arreglo (de otro modo, se usaría demasiado espacio adicional para ordenar...). Es simple implementar un árbol completo o casi completo en un arreglo: se ubica la raíz en el casillero  $v[0]$ , y luego se hace que para nodo en el casillero  $v[i]$ , su hijo izquierdo vaya en  $v[2*i + 1]$  y su hijo derecho  $v[2*i + 2]$ . Así, el algoritmo Heapsort primero lanza una fase de armado del grupo, en la cual los elementos del arreglo se reubican para simular el árbol. La Figura 5 muestra un esquema general del proceso de armado del grupo inicial para un pequeño arreglo de cuatro elementos.

Figura 5: Funcionamiento general del Algoritmo Heapsort - Fase 1: Armado del Grupo Inicial.



Una vez armado el grupo inicial<sup>1</sup>, comienza la segunda fase del algoritmo: Se toma la raíz del árbol (que es el mayor del grupo), y se lleva al último casillero del arreglo (donde quedará ya ordenado). El valor que estaba en el último casillero se reinserta en el árbol (que ahora tiene un elemento menos), y se repite este mecanismo, llevando el nuevo mayor al anteúltimo casillero. Así se continúa, hasta que el árbol quede con sólo un elemento, que ya estará ordenado en su posición correcta del arreglo. Gráficamente, la segunda fase se desarrolla como se muestra en la Figura 6.

Figura 6: Funcionamiento general del Algoritmo Heapsort - Fase 2: Ordenamiento Final.



<sup>1</sup> El Heapsort usa un "grupo de datos" para favorecer el proceso de ordenamiento... Y eso nos refiere a la importancia del trabajo en grupos o en equipos para tareas de otra índole. En 1998, la película *Saving Private Ryan* (o *Rescatando al Soldado Ryan*) dirigida por Steven Spielberg y protagonizada por Tom Hanks, cuenta la dramática historia de un pelotón de ocho soldados que tenían la misión de entrar en territorio enemigo y rescatar a un único soldado (el tal Ryan...) en el contexto de la más feroz guerra que haya experimentado la humanidad: la Segunda Guerra Mundial. La película profundiza en la muy delgada línea que lleva a las personas a tomar decisiones aparentemente ridículas, pero fundamentadas en razones de ética, honor, compañerismo y respeto: ¿Qué tanto se justifica arriesgar las vidas de ocho hombres para salvar la de uno solo?

La función que implementa el algoritmo *Heapsort* es la siguiente, y está incluida en el módulo ordenamiento.py del proyecto [F17] *Ordenamiento* que acompaña a esta Ficha:

```
def heap_sort(v):
    # ordenamiento Heap Sort
    n = len(v)

    # Primera fase: crear el grupo inicial...
    for i in range(n):
        e = v[i]
        s = i
        f = (s - 1) // 2
        while s > 0 and v[f] < e:
            v[s] = v[f]
            s = f
            f = (s - 1) // 2
        v[s] = e

    # Segunda fase: Extraer la raíz, y reordenar el vector y el grupo...
    for i in range(n-1, 0, -1):
        valori = v[i]
        v[i] = v[0]
        f = 0
        if i == 1:
            s = -1
        else:
            s = 1
        if i > 2 and v[2] > v[1]:
            s = 2
        while s >= 0 and valori < v[s]:
            v[f] = v[s]
            f = s
            s = 2*f + 1
            if s + 1 <= i - 1 and v[s] < v[s+1]:
                s += 1
            if s > i - 1:
                s = -1
        v[f] = valori
```

A modo de cierre, y para tener una visión completa, compacta y comparativa de estos algoritmos, el proyecto [F17] *Ordenamiento* que acompaña a esta Ficha, contiene un modelo *test03.py* en el cual se puede seleccionar por medio un menú la técnica de ordenamiento a emplear para ordenar un vector. En ese programa hemos aplicado los seis algoritmos citados en la tabla de la *Figura 1*, incluyendo el *Heapsort* (que como se dijo, se agrega en esta Ficha a modo de tema opcional) y el *Quicksort* (que será presentado en la Ficha 27 cuando se presente el tema de la *Recursividad* y la estrategia *Divide y Vencerás*). El programa además, calcula el tiempo que cada algoritmo demora en terminar de ordenar el vector. Pruebe con valores grandes de *n*... y saque sus propias conclusiones. El programa completo es el siguiente (sin validar en cada opción si el arreglo está creado o no... dejamos ese control para los estudiantes):

```
import time
import ordenamiento

__author__ = 'Cátedra de AED'
```

```

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n
def test():
    op = 0
    while op != 9:
        print('1. Generar el vector')
        print('2. Verificar orden')
        print('3. Ordenamiento por Selección Directa')
        print('4. Ordenamiento por Intercambio Directo (Burbuja)')
        print('5. Ordenamiento por Inserción Directa')
        print('6. Ordenamiento Heapsort')
        print('7. Ordenamiento Quicksort')
        print('8. Ordenamiento Shellsort')
        print('9. Salir')
        op = int(input('\t\tIngresé opción: '))
        if op == 1:
            print()
            n = validate(0)
            v = n * [0]
            ordenamiento.generate_random(v)
            print('Hecho... arreglo creado...')
            print()

        elif op == 2:
            print()
            if ordenamiento.check(v):
                print('Está ordenado...')
            else:
                print('No está ordenado...')
            print()

        elif op == 3:
            print()
            print('Ordenamiento: Selección Directa.')
            t1 = time.perf_counter()
            ordenamiento.selection_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 4:
            print()
            print('Ordenamiento: Intercambio Directo (Burbuja).')
            t1 = time.perf_counter()
            ordenamiento.bubble_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 5:
            print()
            print('Ordenamiento: Inserción Directa.')
            t1 = time.perf_counter()
            ordenamiento.insertion_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 6:
            print()

```

```
print('Ordenamiento: Heap Sort.')
t1 = time.perf_counter()
ordenamiento.heap_sort(v)
t2 = time.perf_counter()
tt = t2 - t1
print('Hecho... Tiempo total insumido:', tt, 'segundos')
print()
elif op == 7:
    print()
    print('Ordenamiento: Quick Sort.')
    t1 = time.perf_counter()
    ordenamiento.quick_sort(v)
    t2 = time.perf_counter()
    tt = t2 - t1
    print('Hecho... Tiempo total insumido:', tt, 'segundos')
    print()

elif op == 8:
    print()
    print('Ordenamiento: Shell Sort.')
    t1 = time.perf_counter()
    ordenamiento.shell_sort(v)
    t2 = time.perf_counter()
    tt = t2 - t1
    print('Hecho... Tiempo total insumido:', tt, 'segundos')
    print()

# script principal...
if __name__ == '__main__':
    test()
```

---

## Bibliografía

- [1] Y. Langsam, M. Augenstein and A. Tenenbaum, Estructura de Datos con C y C++, México: Prentice Hall, 1997.
- [2] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [3] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [5] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 18

## Registros

### 1.] Introducción.

Hemos analizado en fichas anteriores el concepto de *estructura de datos* que hemos definido como una variable capaz de contener varios valores al mismo tiempo. Sabemos que Python provee diversas formas de manejo de estructuras de datos y hemos usado algunas como las tuplas, las cadenas de caracteres y los rangos. Por razones relacionadas con el contexto y el enunciado de los problemas que hasta aquí se propusieron, en esas estructuras de datos que usamos hemos almacenado siempre elementos del mismo tipo: o bien sólo números enteros, o bien sólo cadenas de caracteres, o bien elementos de cualquier otro tipo que el programador hubiese necesitado.

Pero el hecho es que en algún momento el programador necesitará almacenar valores de *tipos diferentes* en una misma estructura de datos, para describir algún elemento o entidad del dominio del problema. Este tipo de situaciones existen y son (como siempre...) muy comunes.

A modo de ejemplo, supongamos que se requiere representar y almacenar datos de las distintas asignaturas de un plan de estudios en un programa simple. Supongamos que por cada asignatura se dispone de un código numérico de identificación y de una cadena de caracteres para su nombre.

Una primera idea simple (aunque no sea la idea final a emplear) es recordar que en Python una *tupla* es una secuencia de datos que pueden ser de tipos diferentes, de forma que se puede acceder a cada uno de esos datos por *vía de un índice*. En ese sentido, queda claro que una *tupla* podría usarse en forma *muy elemental* para representar una *asignatura*, ya que se podría emplear una tupla con dos elementos (uno para el código y otro para el nombre) por cada asignatura a describir, como se ve en el ejemplo que sigue [\[1\]](#) [\[2\]](#):

```
a1 = 1, 'AED'  
a2 = 2, 'PPR'  
a3 = 3, 'TSB'
```

Las tres variables *a1*, *a2* y *a3* son *tuplas* que contienen (cada una de ellas) los datos de una asignatura, y *cada uno de esos datos puede accederse mediante su índice*. Por ejemplo, el siguiente script muestra en consola los datos de las tres asignaturas:

```
print('Asignatura 1 - Código:', a1[0], 'Nombre:', a1[1])  
print('Asignatura 2 - Código:', a2[0], 'Nombre:', a2[1])  
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])
```

Sin embargo, el enfoque anterior tiene dos problemas: el primero es que desde el punto de vista de la *claridad del código fuente* el programador debe recordar en qué posición de cada tupla está cada valor de la asignatura y acceder a él mediante un índice numérico, que no es

tan descriptivo ni simple de recordar. El programador debe saber que el código de la materia lleva el índice 0 dentro de la tupla, y el nombre de la materia lleva el índice 1. El código fuente debería complementarse con innumerables comentarios aclaratorios que finalmente significan más trabajo y más cuidado para no cometer un error.

Y en este caso, el segundo problema es bastante más grave: en Python, una *tupla* es una *secuencia immutable*... lo cual quiere decir que *una vez que se creó la tupla no pueden modificarse los valores almacenados en ella* [1]. Esto llevaría a situaciones claramente inaceptables: si el programador quisiese, por ejemplo, cambiar el nombre de una de las asignaturas, se produciría un error de intérprete y el programa se interrumpiría:

```
a3 = 3, 'TSB'  
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])  
  
a3[1] = 'DLC' # Error aquí!!!  
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])
```

El script anterior lanzaría un error en la tercera línea, similar al que mostramos aquí:

```
Traceback (most recent call last):  
  File "C:/Ejemplo/registros.py", line 7, in <module>  
    a3[1] = 'DLC'  
TypeError: 'tuple' object does not support item assignment
```

Por supuesto, Python provee otros tipos de secuencias y estructuras de datos *mutables* (como las *listas*) que podrían usarse en lugar de las *tuplas* para evitar el problema anterior, pero seguiríamos teniendo el primer inconveniente: el programador tendrá que recordar en qué casillero exacto de cada estructura está almacenado cada valor para deducir el índice de acceso.

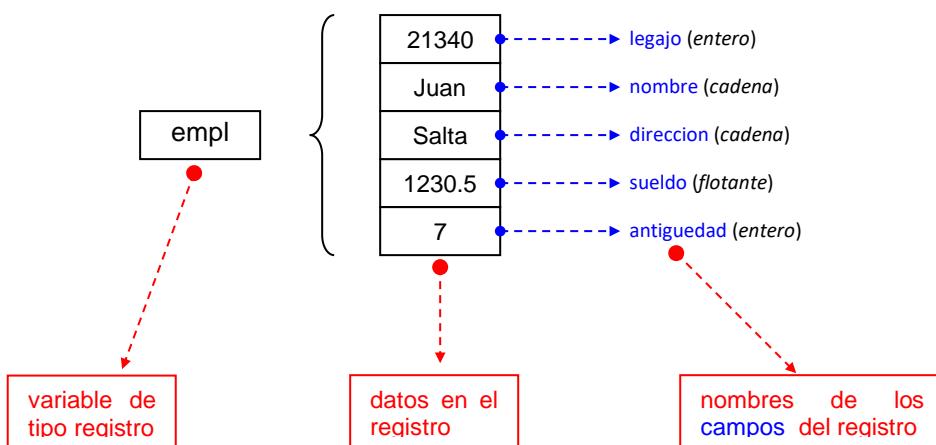
Claramente se requiere poder almacenar valores de tipos diferentes en una misma estructura de datos, pero de forma que la sintaxis y forma de gestión sea clara. Como vimos algunos de los enfoques que Python provee para lograr esto terminan siendo poco claros y por lo tanto, necesitamos una nueva estructura. Esta estructura es la que estudiaremos en esta Ficha y se designa con el nombre genérico de *registro*.

## 2.] Registros en Python.

Un *registro* es un conjunto *mutable* de valores que *pueden ser de distintos tipos*. Cada componente de un registro se denomina *campo* (o también *atributo*, dependiendo del contexto). Los *registros* son útiles para representar en forma clara a cualquier elemento (o *entidad* u *objeto*) del dominio o enunciado del problema que el programador necesite manejar y cuya descripción pudiera contener datos de tipos diferentes.

Por ejemplo, si se desea en un programa representar la información de un *empleado* de una empresa, puede usarse una variable *empl* de tipo *registro* con *campos* para el *legajo* (que sería un valor de *tipo entero*), el nombre (una *cadena de caracteres*), la dirección (otra *cadena de caracteres*), el *suelo básico* (un valor de *coma flotante*) y la *antigüedad* de ese empleado en la empresa (que sería otro valor *entero*). El siguiente gráfico muestra una representación de la forma que podría tener tal variable [3]:

Figura 1: Esquema de una variable de tipo registro.



En el esquema anterior se supone el uso de una *variable registro* llamada *empl*, la cual consta de cinco *campos*, llamados *legajo*, *nombre*, *direccion*, *sueldo* y *antiguedad*. La importancia de los registros como estructuras de datos es evidente en el ejemplo anterior: sin registros en algunos lenguajes no sería posible almacenar en una misma variable todos los datos del empleado supuesto, ya que esos datos son todos de tipos diferentes, y en otros lenguajes se produciría mucha confusión en el acceso a cada campo, ya que el programador debería usar índices y saber de antemano en qué casilla quedó cada valor.

Note que no hay ningún problema si un programador quiere definir un registro que contenga *campos del mismo tipo*: un *registro* es una colección de variables llamadas *campos*, y esos campos *pueden* ser de tipos diferentes (pero ***no deben ser obligatoriamente*** de tipos distintos)

Para definir variables de tipo registro en un programa, lo común es proceder primero a declarar un nuevo nombre o identificador de *tipo de dato* para el registro. Con este nuevo tipo, se definen luego las variables. La declaración de un *tipo registro* se puede hacer en Python con la palabra reservada *class*, básicamente en la forma que se indica en el siguiente esquema (ver modelo *test01.py* - proyecto [F18] *Registros* que acompaña a esta ficha) [1]:

```
# declaración de un tipo Empleado como registro vacío
class Empleado:
    pass

# creación de variables de tipo Empleado
e1 = Empleado()
e1.legajo = 1
e1.nombre = 'Juan'
e1.direccion = 'Calle 1'
e1.sueldo = 10000
e1.antiguedad = 10

e2 = Empleado()
e2.legajo = 2
e2.nombre = 'Luis'
e2.direccion = 'Calle 2'
e2.sueldo = 20000
e2.antiguedad = 15

e3 = Empleado()
e3.legajo = 3
e3.nombre = 'Pedro'
```

```
e3.direccion = 'Calle 3'
e3.sueldo = 25000
e3.antiguedad = 20
```

En la forma que se trabajó en el esquema anterior, el identificador *Empleado* constituye un nuevo *tipo de dato* para el programa y en base a él se pueden definir variables que tendrán la estructura de campos o atributos que el programador necesite.

Las variables *e1*, *e2*, y *e3* del ejemplo anterior se crean como *registros* de tipo *Empleado* (también se dice que se crean como *instancias* de ese tipo) a través de la función constructora *Empleado()*, que está automáticamente disponible para tipos definidos mediante la palabra reservada *class*. Cada vez que se invoca a esa función, Python crea un registro vacío, sin ningún campo, y *retorna la dirección de memoria* donde ese registro vacío quedó alojado:

```
e1 = Empleado()
```

En el ejemplo anterior, la dirección retornada por la *función constructora* se asigna en la variable *e1*, que de allí en adelante se usa para acceder y gestionar el registro (se dice que *e1* está *apuntando al registro* o que está *referenciando al registro*).

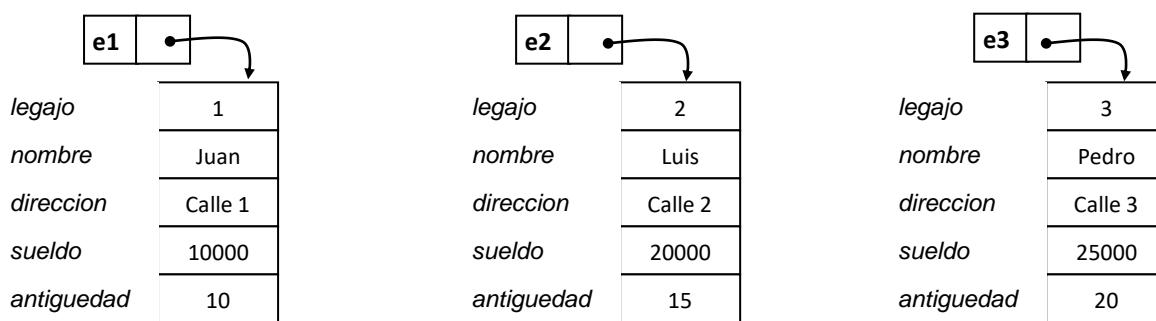
Como cada variable se crea inicialmente como un *registro vacío*, entonces luego el programador debe ir agregando campos (o atributos) a esas variables según lo necesite, usando el *operador punto (.)* para hacer el acceso:

```
e1 = Empleado()
e1.legajo = 1
e1.nombre = 'Juan'
e1.direccion = 'Calle 1'
e1.sueldo = 10000
e1.antiguedad = 10
```

Como se dijo, el acceso a los campos individuales de una variable *registro* se hace con el *operador punto (.)* en forma muy sencilla: se escribe primero el *nombre de la variable* registro (que contiene la dirección de memoria de ese registro), luego un *punto*, y finalmente el *nombre del campo* que se quiere agregar o acceder. Con esto se forma lo que se conoce como el *identificador del campo*, y a partir de aquí se opera con ese campo en forma normal, como se haría con cualquier variable común. En este caso, se están agregando cinco campos a la variable *e1*, llamados *legajo*, *nombre*, *direccion*, *sueldo* y *antiguedad*, cada uno asignado con el valor inicial que se requiera.

Luego de crearse las tres variables *e1*, *e2* y *e3* y luego de agregarse a cada una los campos citados, la memoria asignada a ellas podría verse como en el esquema de la *Figura 2*.

**Figura 2: Esquema de memoria para tres registros de tipo *Empleado*.**



Como se puede ver, cada variable apunta a un registro diferente de forma que cada uno contiene su propia copia de cada uno de los cinco campos, y cada campo en cada registro tiene su propio valor, que es independiente del valor que ese mismo campo tenga en los otros registros. Las siguientes instrucciones muestran en consola estándar el *legajo* y el *nombre* de cada uno de los tres empleados:

```
# mostrar legajo y nombre de cada empleado...
print('Empleado 1 - Legajo:', e1.legajo, '- Nombre:', e1.nombre)
print('Empleado 2 - Legajo:', e2.legajo, '- Nombre:', e2.nombre)
print('Empleado 3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre)
```

Además, como un registro es una estructura *mutable*, el valor de cualquier campo puede ser modificado cuando se necesite, recordando siempre que el acceso a un campo se realiza mediante el *operador punto*. Mostramos algunos ejemplos de operaciones válidas para los campos de los registros *e1*, *e2* y *e3* que ya hemos definido en los ejemplos anteriores [3]:

```
# algunas operaciones válidas...

# cambiar el legajo de e1...
e1.legajo = 4

# hacer que la antiguedad de e3 sea igual que la de e2...
e3.antiguedad = e2.antiguedad

# cargar por teclado el sueldo de e2...
e2.sueldo = float(input('Ingrese el nuevo sueldo del empleado 2: '))

# mostrar la dirección del empleado e1...
print('Direccion del empleado 1:', e1.direccion)

# sumar un año a la antiguedad de e3...
e3.antiguedad += 1
```

También recuerde que en Python las variables no quedan atadas a un tipo fijo y estático, y de hecho, nada impediría entonces que se cree una cuarta variable de tipo *Empleado* con una *estructura de campos diferente* a las tres que ya existen, como se ve en la variable *e4* que se muestra a continuación, en la cual *se ha agregado un campo extra para indicar el cargo del empleado* (ver modelo *test02.py* en el proyecto [F18] *Registros* que acompaña a esta Ficha) [1] [2]:

```
# creación de una variable Empleado con un campo adicional...
e4 = Empleado()
e4.legajo = 10
e4.nombre = 'Luis'
e4.direccion = 'Calle 4'
e4.sueldo = 50000
e4.antiguedad = 30
e4.cargo = 'Gerente'
```

En el ejemplo anterior, la variable *e4* se crea también como un registro de tipo *Empleado*, y se agregan en ella los mismos cinco campos que se habían incluido en las otras tres. Pero en la última instrucción se agrega para *e4* el campo *cargo*, que las otras tres no tenían: La variable *e4* (y sólo la variable *e4*) contendrá el campo *cargo* para asignar en él el nombre del puesto o cargo ocupado por el empleado. La siguiente secuencia de instrucciones muestra los legajos y los nombres de los cuatro empleados, más el cargo del cuarto:

```
# mostrar datos básicos...
print('E1 - Legajo:', e1.legajo, '- Nombre:', e1.nombre)
print('E2 - Legajo:', e2.legajo, '- Nombre:', e2.nombre)
print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre)
print('E4 - Legajo:', e4.legajo, '- Nombre:', e4.nombre, '- Cargo:', e4.cargo)
```

Como las variables *e1*, *e2* y *e3* no contienen el campo *cargo*, se produciría un error de intérprete al intentar accederlo para consultararlo:

```
print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre, '- Cargo:', e3.cargo)
```

La instrucción anterior produciría un error como el siguiente:

```
File "C:/Ejemplo/test02.py", line 42, in <module>
    print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre, '- Cargo:', e3.cargo)
AttributeError: 'Empleado' object has no attribute 'cargo'
```

En general, la inicialización de un registro (así como cualquier otra operación) puede canalizarse por medio de funciones: no hay motivo para escribir largas secuencias de código fuente que hagan la misma tarea. En el siguiente modelo simple (*test03.py* – proyecto [F18] *Registros*) se crean los mismos tres registros de tipo *Empleado* que ya hemos visto, pero cada uno de ellos es inicializado mediante una función que hemos llamado *init()*, y luego cada uno de ellos es visualizado mediante otra función que hemos llamado *write()*:

```
# declaración de un tipo registro vacío
class Empleado:
    pass

# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant

# una función para mostrar un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')

# una función de prueba
def test():
    # creación de variables vacías de tipo Empleado...
    e1 = Empleado()
    e2 = Empleado()
    e3 = Empleado()

    # inicialización de campos de las tres variables...
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
    init(e2, 2, 'Luis', 'Calle 2', 20000, 15)
    init(e3, 3, 'Pedro', 'Calle 3', 25000, 20)
```

```

# visualizacion de los valores de los tres registros...
write(e1)
write(e2)
write(e3)

# script principal...
if __name__ == '__main__':
    test()

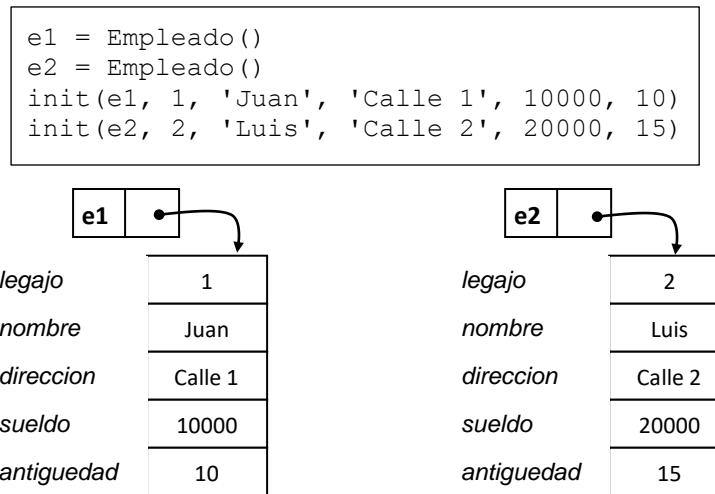
```

Note que en este ejemplo simple, es la propia función `init()` la que finalmente "define" la estructura de campos del registro que entra como primer parámetro.

Un detalle que no debe pasarse por alto, es que como ya dijimos, en Python una variable de tipo *registro* contiene en realidad la *dirección de memoria* del registro asociado a ella (se dice la variable es *una referencia a ese registro*). Por lo tanto, si se asignan dos variables de tipo *registro* entre ellas (por ejemplo, dos registros de tipo *Empleado*) **no se estará haciendo una copia** del registro original, **sino una copia de las direcciones de memoria**, haciendo que ambas variables queden apuntando al mismo registro.

A partir de aquí, entonces, el programador debe tener cuidado de entender bien lo que hace cuando opera con variables que son *referencias*: supongamos que las dos variables *e1* y *e2* apuntan a dos registros diferentes, ambos de tipo *Empleado*. El gráfico de memoria se vería como en la gráfica que sigue:

Figura 3: Dos referencias apuntando a registros diferentes.

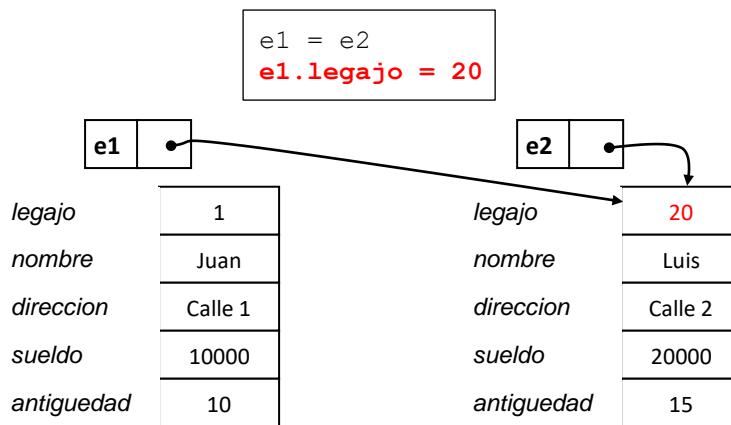


Si luego se hace una asignación como la siguiente:

```
e1 = e2
```

entonces *ambas* variables pasarán a apuntar o referenciar al *mismo* registro (el que originalmente era apuntado por *e2*). Por lo tanto, si se modifica el valor de algún campo para *e1*, se estará modificando *también* el registro referido por *e2*. La asignación *e1 = e2* **no copia** los contenidos del registro apuntado por *e2* hacia el registro apuntado por *e1*. Lo que copia es la *dirección contenida en e2* en la variable *e1*:

Figura 4: Dos referencias apuntando al mismo registro.



El registro apuntado originalmente por *e1* queda *des-referenciado*: esto significa que el programa ha perdido la dirección de memoria de ese registro y ya no hay forma de volver a utilizarlo. En algunos lenguajes (como C++), esta situación implica un error por parte del programador, ya que el registro des-referenciado sigue ocupando memoria aunque ya no pueda ser accedido.

Pero en Python (como en Java y otros lenguajes) eso no es necesariamente un error, pues en Python existe un sistema de *recolección de residuos* de memoria que se encarga de chequear la memoria en busca de variables "perdidas" en tiempo de ejecución. Cuando un programa se ejecuta, *en otro hilo de ejecución paralelo* corre también un proceso llamado *garbage collector*, que se encarga de los elementos des-referenciados, sin que el programador deba preocuparse por ellos.

El hecho ya citado de que una variable registro mantiene una referencia al registro, también hace que al enviar como parámetro una variable registro a una función, la misma pueda usarse para modificar el **contenido** del registro (aunque no puede cambiarse la dirección contenida en el parámetro). La función *init()* que ya hemos visto, hace justamente eso: toma como parámetro una variable *empleado* apuntando a un registro *ya creado*, y procede a modificar *su contenido* para inicializarlo:

```
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
```

Si la función *init()* intentase cambiar el valor de la variable *empleado* como se muestra en el modelo siguiente:

```
def init(empleado, leg, nom, direc, suel, ant):
    empleado = Empleado()
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
```

entonces dentro de la función la variable *empleado* estaría apuntando a un registro que efectivamente sería inicializado con los cinco campos asignados. Pero al terminar de ejecutarse la función, la variable local *empleado* se destruye, y con eso el registro que se había creado queda des-referenciado. La variable registro que hubiese sido enviada como parámetro actual quedará sin cambios y seguirá apuntando al registro que apuntaba originalmente. Con estas consideraciones, el siguiente programa (modelo *test04.py* en el proyecto *F[18] Registros*) produce un error al ejecutarse:

```
# declaración de un tipo registro vacío
class Empleado:
    pass

# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):

    # cuidado con esta línea...
    empleado = Empleado()

    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant

# una función para mostrar por consola un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')

# una función de prueba
def test():
    e1 = Empleado()
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
    write(e1)

# script principal...
if __name__ == '__main__':
    test()
```

El error se produce cuando al regresar de la función *init()* se invoca a la función *write()* para mostrar el registro. La función *write()* intentará mostrar los campos de la variable *e1*, pero debido al mecanismo antes descripto, *esa variable no tiene ningún campo*: fue creada en la función *test()* como registro vacío, luego se envió como parámetro a *init()*, la cual intentó cambiar la dirección que contenía por otra. Como eso no es posible debido al mecanismo de parametrización por valor, la variable *e1* volvió a *test()* exactamente con el mismo valor que

tenía antes de invocar a *init()*: la dirección de un registro vacío. Y eso provoca el fallo en *write()*...

```
Traceback (most recent call last):
  File "C:/Ejemplo/test04.py", line 37, in <module>
    test()
  File "C:/Ejemplo/Fuentes/[F14] Registros/test04.py", line 32, in test
    write(e1)
  File "Ejemplo/test04.py", line 21, in write
    print("\nLegajo:", empleado.legajo, end=' ')
AttributeError: 'Empleado' object has no attribute 'legajo'
```

Obviamente, los problemas anteriores se resuelven de inmediato en forma simple, haciendo que nuestra función *init()* *cree el registro con la función constructora, lo inicialice y retorne el registro* que acaba de crear e inicializar, en forma similar a lo siguiente:

```
class Empleado:
    pass

    def init(leg, nom, direc, suel, ant):
        empleado = Empleado()
        empleado.legajo = leg
        empleado.nombre = nom
        empleado.direccion = direc
        empleado.sueldo = suel
        empleado.antiguedad = ant
        return empleado
```

Si la función *init()* está definida así, entonces la creación de un objeto puede hacerse de forma más simple y natural, como se ve en la función *test()* que sigue:

```
def test():
    e1 = init(1, 'Juan', 'Calle 1', 10000, 10)
    write(e1)
```

Note que esta nueva versión de nuestra función *init()* ahora no necesita tomar como parámetro el registro cuyos campos se desea crear, ya que la misma función crea ese registro y lo completa campo a campo, *para finalmente retornarlo*.

### 3.] Creación e inicialización de registros mediante constructores.

Como vimos, la palabra reservada *class* permite declarar un nuevo tipo de datos a modo de un conjunto capaz de contener una colección de variables (los *campos*). Pero la novedad es que el conjunto o tipo definido con *class* puede contener *funciones* además de *variables* o *campos*.

Antes de continuar conviene aclarar algunos elementos de terminología: el concepto de *registro* como agrupamiento de variables de tipos que pueden ser diferentes es propio del contexto de la *Programación Estructurada* (que abrevamos como *PE*), y muchos lenguajes que soportan ese paradigma disponen de palabras reservadas específicas para definir *registros* (tales como *record* en Pascal o *struct* en C). Y en el contexto de la *PE* un *registro* es una colección de variables que pueden ser de tipos diferentes, pero *un registro no puede contener funciones*.

Sin embargo, la posterior llegada de la *Programación Orientada a Objetos* (abreviada como *POO*) aportó nuevos conceptos y herramientas, entre las que aparece nítidamente la noción de *clase*. En la *POO*, una *clase* es un tipo de datos muy similar al *registro* de la *PE*, pero con una primera diferencia esencial: **una clase puede contener funciones además de campos**.

Los elementos que en un *registro* se llaman *campos* (o sea, las *variables* definidas dentro del *registro*), reciben el nombre de *atributos* cuando se definen en una *clase*. Y si una *función* se define dentro de una *clase*, pasa a designarse como un *método* (en lugar de una *función*).

Finalmente (y como ya vimos), las variables que se definen en base a un tipo registro se llaman ellas mismas *registros* (o también *instancias*), pero las variables que se definen en base a una clase se suelen designar como *objetos* (aunque aquí también cabe la designación de *instancias*)

En Python, la palabra reservada *class* está claramente pensada para definir *clases* (y no *registros*). Obviamente, un programador que aún no tiene conocimientos de *POO* puede usar la palabra *class* para definir *registros* simplemente incluyendo sólo campos o atributos en la clase pero sin incluir funciones o métodos. Y ese ha sido el criterio que hemos seguido a lo largo de esta Ficha: usar versiones simplificadas de clases (con atributos pero sin métodos) para emular el concepto de registro.

Por supuesto esta técnica para crear e inicializar un registro es válida, pero el hecho es que no hay motivo real para desaprovechar la potencia del uso de clases y los métodos en esas clases si el lenguaje utilizado provee el soporte. Y Python provee clases... no registros. Por lo tanto, veamos cuál es la forma de usar correctamente el recurso.

Recuerde: una clase puede pensarse como un registro que además de contener campos (o atributos), puede contener funciones (o métodos). Y uno de los métodos más importantes que una clase puede contener se designa como *método constructor* (o simplemente, un *constructor*). El objetivo de un constructor, es la creación y la inicialización de una instancia. En Python, un método constructor debe llamarse específicamente *\_\_init\_\_()* y como dijimos, debe definirse dentro del ámbito de la clase. El programa siguiente (*test05.py – Proyecto [F18] Registros*) es el mismo que ya vimos para crear instancias de tipo *Empleado*, pero ahora modificado para que la *clase Empleado* contenga el constructor *\_\_init\_\_()* ya citado:

```
# declaración de una clase con un constructor...
class Empleado:
    # un constructor dentro de la clase Empleado...
    def __init__(self, leg, nom, direc, suel, ant):
        self.legajo = leg
        self.nombre = nom
        self.direccion = direc
        self.sueldo = suel
        self.antiguedad = ant

    # una función para mostrar un objeto o instancia de la clase Empleado
    def write(empleado):
        print("\nLegajo:", empleado.legajo, end=' ')
        print("- Nombre:", empleado.nombre, end=' ')
        print("- Direccion:", empleado.direccion, end=' ')
        print("- Sueldo:", empleado.sueldo, end=' ')
        print("- Antiguedad:", empleado.antiguedad, end=' ')
```

```

# una función de prueba
def test():
    # creación e inicialización de objetos de la clase Empleado...
    e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
    e2 = Empleado(2, 'Luis', 'Calle 2', 20000, 15)
    e3 = Empleado(3, 'Pedro', 'Calle 3', 25000, 20)

    # visualización de los valores de los tres objetos...
    write(e1)
    write(e2)
    write(e3)

# script principal...
if __name__ == '__main__':
    test()

```

Todo método de una clase debe definir un primer parámetro formal llamado *self* que represente al objeto o instancia (o registro) sobre el cual se aplica el método. Si ese método es el constructor (*\_\_init\_\_()*) entonces *self* es el objeto o registro que se está creando e inicializando.

Si la clase *Empleado* tiene definido el constructor que hemos mostrado, entonces la siguiente instrucción crea un objeto o registro *e1* de tipo *Empleado*, y lo inicializa con los valores enviados como parámetros actuales:

```
e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
```

La parte derecha de la asignación anterior es una invocación al constructor. Aun cuando el nombre *\_\_init\_\_()* no está presente en esa invocación, Python interpreta que si el nombre de una clase se usa a la derecha de una expresión de asignación y se le pasan parámetros, entonces se está invocando *implícitamente* al constructor de esa clase. Note además que no es necesario enviar como parámetro al propio objeto *e1* que se quiere crear: Python también interpreta que si se invoca a un constructor, el objeto que se construye es enviado automáticamente como parámetro a ese constructor, y por si fuese poco, el constructor automáticamente retorna la dirección del objeto creado (sin necesidad de incluir un *return*) razón por la cual el retorno del constructor es asignado en la variable *e1*.

En otras palabras: la expresión anterior invoca al constructor *\_\_init\_\_()* de la clase *Empleado*. El parámetro *self* de ese constructor queda automáticamente asignado con la dirección del objeto que se está creando (sin tener que enviar un parámetro actual desde el exterior para que sea asignado en *self*). El resto de los parámetros actuales enviados (los valores 1, 'Juan', 'Calle 1', 10000, y 10) son asignados en forma normal en los parámetros formales *leg*, *nom*, *direc*, *suel* y *ant* y con esos valores se crean los campos o atributos del objeto. El bloque de acciones del constructor tiene las instrucciones:

```

self.legajo = leg
self.nombre = nom
self.direccion = direc
self.sueldo = suel
self.antiguedad = ant

```

lo cual significa que para el registro u objeto *self* se está creando el campo *self.legajo* con el valor del parámetro formal *leg*, y algo similar para los campos o atributos *self.nombre*, *self.direccion*, *self.sueldo* y *self.antiguedad*. La dirección del registro u objeto *self* es la que

automáticamente retorna el constructor `__init__()`, y esa dirección se asigna en la variable `e1` que se mostró en el ejemplo.

Una vez que el registro u objeto ha sido creado e inicializado de esta forma, puede ser utilizado de allí en adelante en la misma forma en que hemos visto en esta Ficha: se usa el **nombre de la variable registro seguido de un punto y luego el nombre del campo** que se quiere acceder, como se ve en la función `write()` del ejemplo:

```
# una función para mostrar un objeto o instancia de la clase Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')
```

Para aplicar lo visto, veamos ahora un caso de estudio muy conocido, como es el de la representación de puntos en un plano. El siguiente enunciado formaliza la idea:

**Problema 46.)** Desarrollar un programa que permita manejar puntos en un plano. Por cada punto se deben indicar sus coordenadas (que pueden ser números en coma flotante) y una cadena de caracteres a modo de descriptor del punto cuando se muestren sus valores en pantalla. Incluir un menú de opciones que permita:

1. Cargar por teclados los datos de un punto y mostrar esos datos en pantalla.
2. Cargar por teclado los datos de un punto, y mostrar la distancia al origen desde ese punto.
3. Cargar por teclado los datos de dos puntos, y mostrar la pendiente de la recta que los une.

**Discusión y solución:** Un punto  $p_1$  en un plano bidimensional gestionado a través de un par de ejes cartesianos, puede ser representado en forma simple a partir de sus dos coordenadas ( $x, y$ ) que dan la distancia horizontal y vertical de ese punto  $p_1$  al origen del sistema<sup>1</sup>, como se ve en la Figura 5 de página 359.

Esto permite rápidamente pensar en que un punto puede representarse en un programa en Python mediante un registro que contenga tres campos: dos valores de tipo `float` para las coordenadas, y una cadena de caracteres para el descriptor (que en el gráfico anterior es "p1"). Dentro del proyecto [F18] *Registros*, el módulo `geometry.py` contiene la declaración del registro o clase `Point` que usaremos para representar puntos, incluyendo su constructor:

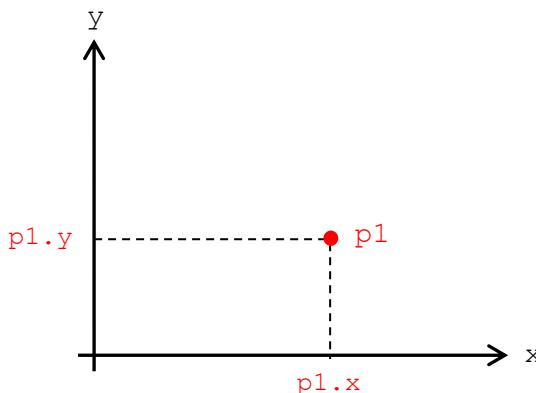
```
class Point:
    def __init__(self, cx, cy, desc='p'):
        self.x = cx
        self.y = cy
        self.descripcion = desc
```

El constructor `__init__(self, cx, cy, desc='p')` crea los campos del punto `self` que toma automáticamente como parámetro. El parámetro formal `desc` tiene por defecto asignada la

<sup>1</sup> El uso de un sistema de ejes cartesianos para representar objetos o propiedades en función de dos o más coordenadas o valores aparece en las más insospechadas disciplinas, y en ese sentido el cine ha mostrado algunas escenas memorables. Un ejemplo concreto y muy recordado es la película *Dead Poets Society (La Sociedad de los Poetas Muertos)* de 1989, dirigida por Peter Weir. El profesor Keating (interpretado por Robin Williams) enseña literatura en un colegio secundario. El libro de texto que usan sus alumnos les explica cómo "medir" el valor de una poesía usando un sistema de ejes cartesianos... y el profesor Keating les ordena arrancar esa página del libro calificándola de basura... ¡Toda una perlita!

cadena 'p', por lo cual si ese parámetro se omite al invocar al constructor, el campo *descripcion* de *point* quedará valiendo 'p'.

**Figura 5:** Representación de un punto en un sistema cartesiano (primer cuadrante).



El mismo módulo *geometry.py* contiene el resto de las funciones pedidas en el enunciado. El cálculo de la distancia de un punto hasta el origen de coordenadas del sistema, se realiza con la función *distance()*, que simplemente aplica el Teorema de Pitágoras:

```
def distance(point):
    # Pitágoras...
    return math.sqrt(pow(point.x, 2) + pow(point.y, 2))
```

El módulo *geometry.py* provee una función *to\_string()* simple pero muy cómoda, que crea una cadena de caracteres a partir de un registro u objeto *point* de tipo *Point*, y retorna esa cadena lista para ser mostrada en consola si fuese necesario:

```
def to_string(point):
    r = str(point.descripcion) + '(' + str(point.x) + ',' + str(point.y) + ')'
    return r
```

El cálculo de la pendiente de la recta que une a dos puntos *p1* y *p2*, se realiza mediante la función *gradient(p1, p2)*. Esa pendiente no es otra cosa que la *tangente trigonométrica* del ángulo  $\alpha$  que forma la recta *p1p2* con la horizontal que pasa por *p1* (ver *Figura 6*, en página 360).

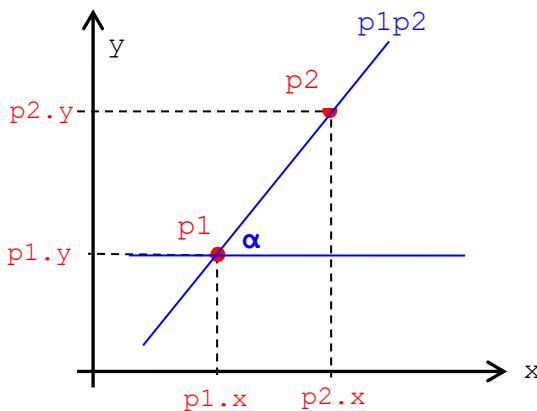
Sólo hay que tomar la precaución de controlar que  $\Delta x$  no sea cero (es decir, controlar que los puntos *p1* y *p2* no formen una recta vertical) pues en ese caso el cociente no puede calcularse (y se dice que la pendiente de una recta vertical es indefinida).

En base a lo expuesto, la función *gradient(p1, p2)* es la siguiente:

```
def gradient(p1, p2):
    # calcular "delta y" y "delta x"
    dy = p2.y - p1.y
    dx = p2.x - p1.x

    # si los puntos no forman una recta vertical,
    # retornar la pendiente...
    if dx != 0:
        return dy / dx

    # de otro modo, la pendiente es indefinida...
    return None
```

Figura 6: Representación geométrica de la pendiente de la recta  $p1p2$ .

$$\begin{aligned} \text{pendiente}(p1p2) &= \operatorname{tg}(\alpha) = \Delta y / \Delta x \\ &= (p2.y - p1.y) / (p2.x - p1.x) \end{aligned}$$

Asumiendo que el módulo *geometry.py* existe y contiene estas declaraciones y funciones (ver proyecto [F18] *Registros*), entonces el programa completo es el siguiente (de nuevo, ver proyecto [F18] *Registros*, modelo *test06.py*):

```
from geometry import *

def opcion1():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = Point(cx, cy)

    print(to_string(p))

def opcion2():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = Point(cx, cy)

    d = distance(p)
    print('Distancia al origen:', d)

def opcion3():
    cx1 = float(input('Punto 1 - Coordenada x: '))
    cy1 = float(input('Punto 1 - Coordenada y: '))
    p1 = Point(cx1, cy1)

    cx2 = float(input('Punto 2 - Coordenada x: '))
    cy2 = float(input('Punto 2 - Coordenada y: '))
    p2 = Point(cx2, cy2)

    pd = gradient(p1, p2)
    print('Pendiente de la recta que los une:', pd)

def menu():
    op = -1
```

```

while op != 4:
    print('1. Cargar y mostrar un punto')
    print('2. Distancia al origen')
    print('3. Pendiente de la recta que une dos puntos')
    print('4. Salir')
    op = int(input('Ingrese opcion: '))

    if op == 1:
        opcion1()
    elif op == 2:
        opcion2()
    elif op == 3:
        opcion3()

if __name__ == '__main__':
    menu()

```

#### 4.] Arreglos de registros (o vectores de registros) en Python.

En muchas aplicaciones será más útil almacenar registros completos en cada casillero de un arreglo unidimensional, en lugar de usar arreglos paralelos como mostramos en la sección anterior. Esto es perfectamente válido en Python: un arreglo en Python puede contener elementos de cualquier tipo incluyendo *elementos que sean referencias a registros*. Esta técnica es efectivamente más cómoda que la de los arreglos paralelos, ya que en lugar de definir varios arreglos separados para almacenar en cada uno una parte de los valores que se necesitan, se define un único arreglo que contiene a todos los datos y con eso no sólo se simplifica el código fuente sino que se ahorra esfuerzo cuando se tiene que enviar ese conjunto de datos como parámetro a una función, ya que en lugar de varios arreglos, se pasa sólo uno [3]. Un arreglo unidimensional de registros también suele designarse como un *vector de registros*.

Supongamos que se tiene definido un registro *Estudiante* y que se desea almacenar varios registros de ese tipo en un arreglo. Suponga que se cuenta con el tipo *Estudiante* ya definido y con una función constructora `__init__()` para crear y asignar los campos a cada registro de ese tipo:

```

class Estudiante:
    def __init__(self, leg=0, nom='', pro=0):
        self.legajo = leg
        self.nombre = nom
        self.promedio = pro

```

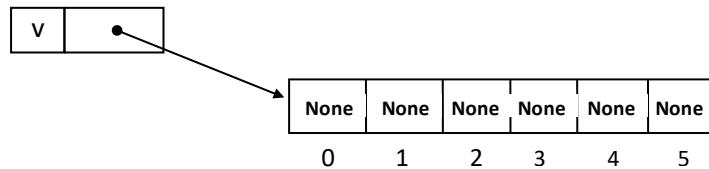
Si se conoce de antemano la cantidad  $n$  de registros a almacenar, una forma de crear el arreglo pedido consiste en declarar primero la referencia a ese arreglo, reservando  $n$  casilleros que pueden comenzar valiendo cualquier valor (por ejemplo, *None*): lo importante no es qué valor comience asignado en cada casillero (puesto que este valor puede cambiarse más tarde incluso por uno de otro tipo), *sino que se tengan efectivamente  $n$  casilleros* [1]:

```

n = 6
v = n * [None]

```

La instrucción anterior crea un arreglo de  $n = 6$  componentes con valor inicial *None*. La idea es que en cada uno se almacene luego una referencia a un registro de tipo *Estudiante*, aunque todavía no se han creado esos registros. El aspecto que tendría este arreglo en este momento en memoria sería como el que se muestra a continuación:

**Figura 7: Un arreglo con  $n = 6$  casilleros con valor inicial *None*.**

A diferencia de un arreglo que contenga valores de tipo simple, un arreglo de referencias todavía no contiene elementos listos para usar (a menos que el programador quiera efectivamente usar los valores `None`...): deben crearse los registros que serán apuntados desde cada casilla del arreglo, y recién entonces comenzar a usar esos objetos. Por ejemplo, en este caso, podría hacerse algo como:

```
for i in range(n):
    v[i] = Estudiante()
```

lo cual haría que cada casilla del arreglo contenga ahora una referencia a un registro de tipo *Estudiante*, pero con todos sus campos valiendo los *valores default que asigna la función constructora* (ya que al invocarla en el momento de crear cada registro, no le hemos enviando ningún valor como parámetro formal). Los valores `None` ya no están en cada casillero, y fueron reemplazados por las direcciones de cada uno de los registros. Una mejor forma de proceder, podría ser que no sólo se creen los registros vacíos, sino que también se proceda a cargar por teclado los datos a almacenar en cada uno, y *luego se creen los registros con sus campos inicializados de forma definitiva*. La secuencia que sigue muestra la forma de hacerlo:

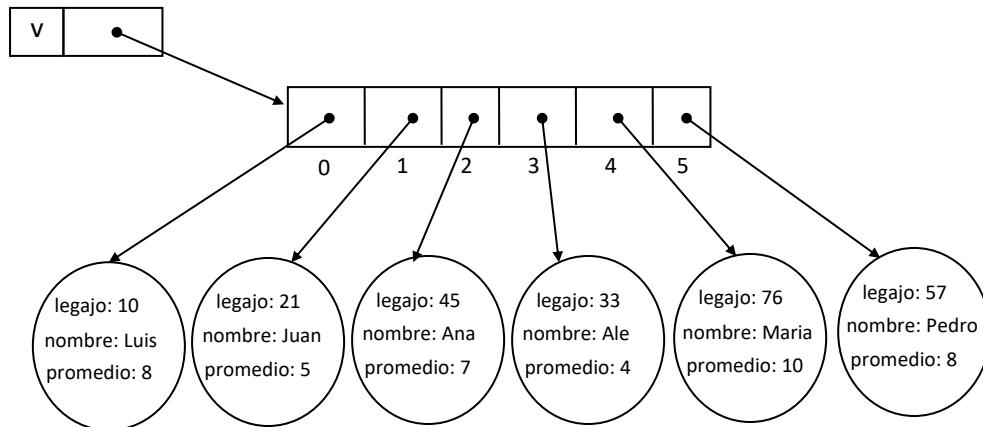
```
for i in range(n):
    leg = int(input('Legajo[' + str(i) + ']: '))
    nom = input('Nombre: ')
    pro = float(input('Promedio: '))
    print()
    v[i] = Estudiante(leg, nom, pro)
```

Está claro que puede lograrse el mismo resultado dejando los valores default de la función constructora `__init__`, y luego *creando y asignando directamente los campos en cada casillero*. Lo anterior es equivalente a:

```
for i in range(n):
    v[i] = Estudiante()
    v[i].legajo = int(input('Legajo[' + str(i) + ']: '))
    v[i].nombre = input('Nombre: ')
    v[i].promedio = float(input('Promedio: '))
    print()
```

Como cada casillero `v[i]` del arreglo es un registro de tipo *Estudiante*, entonces el acceso a cada campo de `v[i]` se hace con el operador punto combinado con el propio identificador `v[i]`: `v[i].legajo` permite acceder al campo *legajo* del registro ubicado en `v[i]`, y en forma similar ocurre para el resto de los campos.

Las dos secuencias anteriores crean diferentes registros y los asignan en distintas casillas del arreglo, que podría verse ahora como sigue (suponga que los datos cargados por teclado fueron efectivamente los que se ven en la figura):

**Figura 8:** Un arreglo con  $n = 6$  referencias a registros ya creados de tipo *Estudiante*.

Sólo cuando cada registro haya sido creado se puede recorrer el arreglo y aplicar a cada uno alguna tarea o proceso. Recuerde que  $v[i]$  es la referencia que apunta al registro en la casilla  $i$ , y por lo tanto algo como  $v[i].legajo$  accede al número de legajo almacenado en el registro  $v[i]$ . Pero esto sólo es válido si  $v[i]$  es efectivamente un registro de tipo *Estudiante*: si se intenta acceder a un campo desde una referencia que no apunta a un registro de ese tipo, el programa se interrumpirá lanzando un mensaje de error por campo inexistente.

Finalmente, el arreglo se usa como se usaría a cualquier arreglo: si se sabe qué clase de registros se almacenaron dentro de él, se podrá cargarlos por teclado, visualizarlos, ordenarlos, efectuar búsquedas, etc. Analice el siguiente ejercicio que completa el ejemplo que hemos venido utilizando:

**Problema 47.)** *Se desea almacenar en un arreglo la información de los  $n$  estudiantes que se registraron para participar de un curso de programación. Por cada estudiante se tiene su número de legajo, su nombre y su promedio en la carrera que cursa. Participarán del curso los estudiantes cuyo promedio sea mayor o igual a  $x$ , siendo  $x$  un valor cargado por teclado. Muestre los datos de los estudiantes que participarán del curso, pero ordenados de menor a mayor por número de legajo.*

**Discusión y solución:** El programa que resuelve este problema es el modelo *test07.py* (incluido en el proyecto [F18] *Registros* que viene con esta Ficha), y es el siguiente:

```

class Estudiante:
    def __init__(self, leg, nom, pro):
        self.legajo = leg
        self.nombre = nom
        self.promedio = pro

    def write(est):
        print('Legajo:', est.legajo, end=' ')
        print('- Nombre:', est.nombre, end=' ')
        print('- Promedio:', est.promedio)

    def validate(inf):
        n = inf
        while n <= inf:
            n = int(input('Cantidad de elementos (mayor a ' + str(inf) + '): '))

```

```

    if n <= inf:
        print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def read(estudiantes):
    n = len(estudiantes)
    for i in range(n):
        leg = int(input('Legajo[' + str(i) + ']: '))
        nom = input('Nombre: ')
        pro = float(input('Promedio: '))
        print()
        estudiantes[i] = Estudiante(leg, nom, pro)

def sort(estudiantes):
    n = len(estudiantes)
    for i in range(n-1):
        for j in range(i+1, n):
            if estudiantes[i].legajo > estudiantes[j].legajo:
                estudiantes[i], estudiantes[j] = estudiantes[j], estudiantes[i]

def display(estudiantes, x):
    n = len(estudiantes)
    print('Estudiantes que harán el curso (tienen promedio >=', x, ')')
    for i in range(n):
        if estudiantes[i].promedio >= x:
            write(estudiantes[i])

def test():
    # cargar cantidad de estudiantes...
    n = validate(0)

    # crear un arreglo con n casilleros de valor indefinido...
    # ... se usará para almacenar luego las referencias a los Estudiantes...
    estudiantes = n * [None]

    # cargar el arreglo por teclado...
    print('\nCargue los datos de los estudiantes:')
    read(estudiantes)
    print()

    x = float(input('Promedio mínimo para poder hacer el curso: '))
    print()

    # ordenar alfabéticamente el arreglo...
    sort(estudiantes)

    # mostrar por pantalla el listado...
    display(estudiantes, x)

# script principal...
if __name__ == '__main__':
    test()

```

La función `test()` define un arreglo referenciado por la variable `estudiantes`, que contendrá referencias a registros de tipo `Estudiante` (tipo que fue definido al inicio del módulo). Este arreglo se usará para almacenar los datos de todos los estudiantes que se hayan inscripto para hacer el curso. La misma función `test()` carga por teclado la cantidad total de alumnos `n`, y en base a ese valor crea el arreglo con `n` casilleros valiendo `None`. Recuerde que no se

debería comenzar a usar el arreglo hasta crear o asignar registros de tipo *Estudiante* en cada componente (cosa que se hace en este caso la función *read()*).

La función *sort()* ordena el arreglo de acuerdo a los legajos de los estudiantes, de menor a mayor. Lo más relevante de esta función *es la condición para determinar si debe hacerse un intercambio o no*:

```
if estudiantes[i].legajo > estudiantes[i].legajo:  
    estudiantes[i], estudiantes[j] = estudiantes[j], estudiantes[i]
```

Puede verse que en la condición se accede al campo *legajo* de cada registro, y se comparan sus valores: ahora cada casillero contiene un registro y *no* un valor *int*, por lo que comparar directamente *estudiantes[i]* con *estudiantes[j]* carece de sentido (iy provocaría un error!). Además, si la condición es verdadera puede verse que se procede a intercambiar directamente las referencias a los registros, *sin tener que hacer ese intercambio campo por campo*.

Finalmente, la función *display()* recorre el contenido del arreglo mostrando por consola los datos de los alumnos cuyo promedio sea mayor o igual al valor *x* que se tomó como parámetro. Como el arreglo está ordenado por legajo, los datos de los estudiantes que se muestren saldrán a su vez ordenados por legajo.

Analicemos ahora otro problema en el que se requiere gestionar un vector de registros. El enunciado es el siguiente:

**Problema 48.)** Una empresa dedicada a la producción de piezas de repuestos automotrices está desarrollando un programa para gestionar un arreglo con datos de los *n* insumos que componen a una pieza en particular (cargar *n* por teclado). Todo el vector se usa para representar una misma y única pieza, y todos los insumos registrados en el vector son parte de esa única pieza.

Por cada insumo se tiene como dato su código (es un valor numérico que puede tomar valores de 1 a 20 ambos incluidos), nombre (una cadena de caracteres), valor (es el precio del insumo) y cantidad (es un número que indica la cantidad utilizada del insumo para fabricar la pieza).

Desarrollar un programa en Python controlado por un menú de opciones. Ese menú debe permitir gestionar las siguientes tareas:

- a) Cargar por teclado el vector pedido, validando que el código de insumo esté efectivamente entre 1 y 20.
- b) Mostrar el valor total de la pieza representada por el vector. Para esto debe acumular (sumar) el valor obtenido de multiplicar el precio por la cantidad de cada insumo del arreglo.
- c) Mostrar sólo los insumos cuya cantidad sea menor a un valor *x* cargado por teclado. Este listado debe mostrarse ordenado de menor a mayor de acuerdo al código de los insumos.
- d) Cargar por teclado un valor *c*, y para todo insumo cuya cantidad sea igual a 0 cambiar esa cantidad por el valor *c*. Mostrar los datos de los insumos que hayan sido modificados.

e) Desarrollar una función que reciba como parámetro un código de insumo y muestre por pantalla todos los datos del mismo si se encuentra en el vector. Informe con un mensaje si ese insumo no existe.

**Discusión y solución:** El programa completo puede verse en el proyecto [F18] *Registros* que acompaña a esta Ficha. Dentro de ese proyecto, el módulo *insumos.py* define la clase *Insumo* para manejar registros que representen insumos. La clase incluye su función constructora *\_\_init\_\_()* con un parámetro por cada campo previsto en el registro, y una función adicional *to\_string()* que retorna una cadena con el contenido del registro tomado como parámetro, de forma que esa cadena ya tiene el formato adecuado para ser visualizada en pantalla si fuese el caso:

```
class Insumo:
    def __init__(self, cod=1, nom='', pre=0.0, cant=0):
        self.codigo = cod
        self.nombre = nom
        self.valor = pre
        self.cantidad = cant

    def to_string(insumo):
        r = ''
        r += '{:<15}'.format('Codigo: ' + str(insumo.codigo))
        r += '{:<30}'.format('Nombre: ' + insumo.nombre)
        r += '{:<18}'.format('Precio: ' + str(insumo.valor))
        r += '{:<15}'.format('Cantidad: ' + str(insumo.cantidad))
    return r
```

La función *to\_string()* usa en forma directa los campos que sean de tipo cadena y también convierte cada campo numérico del registro a una cadena de caracteres con la función predefinida *str()* que Python ya provee. Cada cadena así obtenida se une a otra cadena descriptiva con el operador + en forma similar a lo que se muestra a continuación:

```
'Codigo: ' + str(insumo.codigo)
```

En principio, una cadena así formada ya estaría en condiciones de concatenarse a las de los otros campos para finalmente retornar la cadena completa. Sin embargo, para lograr un ajuste más fino en cuanto a la justificación hacia la izquierda y el espaciado entre valores dentro de la cadena final, se está usando aquí el método *format()* incluido dentro de la clase *str* que representa al conjunto de cadenas de caracteres en Python [1].

El método *format()* es invocado por una cadena de caracteres, y retorna esa misma cadena pero ajustada al formato indicado por los parámetros enviados a *format()* y por algunos elementos contenidos en la propia cadena que se designan como *campos de reemplazo*. Los campos de remplazo se componen de algún tipo de indicador de formato *encerrado entre dos llaves*. Veamos un ejemplo sencillo:

```
a, b = 2, 4,
cad = 'La suma de {0} + {1} es {2}'.format(a, b, a+b)
print(cad)
```

En el script anterior la cadena 'La suma de {0} + {1} es {2}' es la cadena mediante la cual se invoca al método *format()*, y es por lo tanto la cadena cuyo formato se quiere ajustar. Dentro de esa cadena se pueden ver tres campos de reemplazo: {0}, {1} y {2}. A su vez, al método *format()* se le están pasando tres parámetros: a, b y a+b. Cuando el método es

invocado, el campo de reemplazo `{0}` en la cadena es reemplazado por el valor (convertido a cadena) del *primer parámetro* que haya recibido (en este caso, el valor de la variable `a`). El segundo campo de reemplazo (o sea, `{1}`) se reemplaza con el valor del segundo parámetro enviado a `format()` (en este caso, el valor de `b`), y así en forma sucesiva hasta agotar todos los campos de reemplazo. De esta forma el `print()` del final del script mostrará en pantalla la siguiente salida:

```
La suma de 2 + 4 es 6
```

Ahora bien: la cadena con la que se invoca al método `format()` puede estar completamente formada por un campo de reemplazo que puede contener no sólo números que refieran a un parámetro, sino otros símbolos (o "comodines") que tienen significados diferentes según como se los agrupe. Veamos el siguiente ejemplo:

```
nom = 'Juan Perez'
edad = 21
cad1 = '{:<30}'.format('Nombre: ' + nom)
cad1 += '{:<10}'.format('Edad: ' + str(edad))
print(cad1)
```

Aquí, la cadena '`{:<30}`' con la que se invoca por primera vez al método está compuesta solamente por un campo de reemplazo, que contiene a su vez la secuencia '`<30`'. Esta secuencia está indicando que la cadena que vaya a reemplazar a ese campo de reemplazo, debe ser alineada o *justificada a la izquierda* (lo que se indica con los caracteres '`:`' y en total, la cadena completa debe ajustarse hasta ocupar 30 caracteres de largo (eso es lo que significa el 30 ubicado al final de la secuencia). Si la cadena original no llegase a completar 30 caracteres, los que falten hasta llegar a 30 hacia la derecha serán llenados con espacios en blanco. Hasta aquí la cadena `cad1` contiene una cadena como 'Nombre: Juan Perez' con 30 caracteres de largo (el resultado del fondo en color celeste es sólo un recurso de texto para reforzar la idea: obviamente, no será mostrado con ese color en la consola de salida) (y note que los últimos 12 caracteres de la derecha se completan con blancos).

Pero en el mismo script se está usando el operador `+=` para concatenar (agregar al final) una segunda cadena a la que ya contenía la variable `cad1`. La cadena que será agregada es de la forma '`{:<10}.format('Edad: ' + str(edad))` y esto indica que la nueva cadena debe también ajustarse a la izquierda, pero completarse hasta llegar a 10 caracteres. Esta segunda cadena contendrá entonces la secuencia 'Edad: 21' (con dos blancos al final). Pero como esta segunda cadena será agregada al final de la primera, y la primera ya estaba ajustada a la izquierda con hasta 30 caracteres, el resultado mostrado por el `print()` final será algo como:

```
'Nombre: Juan Perez'      'Edad: 21'
```

Y puede notarse entonces que en total, la cadena completa que se formó tendrá entonces  $30 + 10 = 40$  caracteres de largo y espacios en blanco correctamente ubicados para facilitar el justificado hacia la izquierda (y por lo tanto el encolumnado) de todos los valores.

Si en lugar de '`{:<30}`' se escribiese '`{:>30}`' (con el signo `>` en lugar de `<`) el efecto sería el mismo, pero con la cadena *ajustada hacia la derecha* y llenando con blancos a la izquierda. Y si se escribiese '`{:^30}`' (con `^` en lugar de `<` o `>`) el efecto sería una *cadena centrada* (con blancos de relleno en ambos márgenes). El siguiente script simple muestra un ejemplo de cada caso:

```
cad = '{:<30}'.format('Hola mundo')
print(cad)
```

```
# muestra: 'Hola mundo'

cad = '{:>30}'.format('Hola mundo')
print(cad)
# muestra: '           Hola mundo'

cad = '{:^30}'.format('Hola mundo')
print(cad)
# muestra: '      Hola mundo'
```

Con todo esto aclarado, es simple ver lo que hace nuestra función *to\_string()* original:

```
def to_string(insumo):
    r = ''
    r += '{:<15}'.format('Codigo: ' + str(insumo.codigo))
    r += '{:<30}'.format('Nombre: ' + insumo.nombre)
    r += '{:<18}'.format('Precio: ' + str(insumo.valor))
    r += '{:<15}'.format('Cantidad: ' + str(insumo.cantidad))
    return r
```

Cada campo del registro *insumo* que se toma como parámetro se convierte a cadena de caracteres (de ser necesario) y se forma por concatenación una sola gran cadena de un total de  $15 + 30 + 18 + 15 = 78$  caracteres de largo, con cada subcadena justificada a la izquierda y rellenada con blancos a la derecha (si fuese requerido). La cadena así formada es retornada por la función. Si los datos contenidos en el registro fuesen:

```
insumo.nombre = 'Puerta derecha'
insumo.codigo = 10
insumo.precio = 1000
insumo.cantidad = 100
```

entonces el siguiente script:

```
print(to_string(insumo))
```

produciría esta salida en la consola:

```
'Codigo: 10      Nombre: Puerta derecha      Precio: 1000      Cantidad: 200 '
```

El módulo *insumos.py* no contiene ya otros elementos. El programa completo para resolver el problema está incluido en el archivo *test08.py* del mismo proyecto, y lo mostramos a continuación:

```
import insumos

def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Error... Se pidió > ' + str(inf) + '... Cargue de nuevo: '))
    return n

def validate_code(mn=1, mx=20):
    cod = int(input('Ingrese código (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Error... era >=' + str(mn) + ' y <=' + str(mx) + '). De nuevo: '))
    return cod
```

```
def read(pieza):
    n = len(pieza)
    for i in range(n):
        nom = input('Nombre[' + str(i) + ']: ')
        cod = validate_code(1, 20)
        val = int(input('Precio: '))
        can = int(input('Cantidad: '))
        pieza[i] = insumos.Insumo(cod, nom, val, can)
    print()

def opcion1(pieza):
    print('Cargue los datos de los insumos de la pieza:')
    read(pieza)

def display_all(pieza):
    for insumo in pieza:
        print(insumos.to_string(insumo))

def opcion2(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Listado completo de insumos para la pieza')
    display_all(pieza)

def total_value(pieza):
    tv = 0
    for insumo in pieza:
        monto = insumo.valor * insumo.cantidad
        tv += monto
    return tv

def opcion3(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Monto total en insumos para la pieza:', total_value(pieza))

def sort(pieza):
    n = len(pieza)
    for i in range(n-1):
        for j in range(i+1, n):
            if pieza[i].codigo > pieza[j].codigo:
                pieza[i], pieza[j] = pieza[j], pieza[i]

def display(pieza, x):
    print('Insumos con menos de', x, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad < x:
            print(insumos.to_string(insumo))

def opcion4(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    x = int(input('Cantidad x de unidades a controlar: '))
```

```
sort(pieza)
display(pieza, x)

def change_quantity(pieza, c):
    exists = False
    print('Insumos que pasaron de 0 unidades a', c, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad == 0:
            exists = True
            insumo.cantidad = c
            print(insumos.to_string(insumo))

    if not exists:
        print('No hay insumos con 0 unidades en esta pieza')

def opcion5(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese la nueva cantidad c para los insumos sin unidades...')
    c = validate(0)
    change_quantity(pieza, c)

def search(pieza, cod):
    for insumo in pieza:
        if insumo.codigo == cod:
            return insumo
    return None

def opcion6(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese el codigo del insumo a buscar...')
    cod = validate_code(1, 20)
    ins = search(pieza, cod)
    if ins is not None:
        print(insumos.to_string(ins))
    else:
        print('No existe un insumo con ese codigo en la pieza')

def test():
    # cargar cantidad de insumos...
    print('Ingrese la cantidad de insumos en la pieza...')
    n = validate(0)

    # crear el arreglo (inicialmente vacio)...
    pieza = n * [None]

    opc = 0
    while opc != 7:
        print('\nMenu de opciones:')
        print('1. Cargar insumos de la pieza')
        print('2. Mostrar todos los insumos de la pieza')
        print('3. Mostrar el valor total de la pieza')
        print('4. Mostrar insumos con menos de x unidades')
        print('5. Cambiar cantidad en insumos con 0 unidades')
        print('6. Buscar un insumo por su codigo')
        print('7. Salir')

        opc = int(input('Ingrese su eleccion: '))
```

```

if opc == 1:
    opcion1(pieza)

elif opc == 2:
    opcion2(pieza)

elif opc == 3:
    opcion3(pieza)

elif opc == 4:
    opcion4(pieza)

elif opc == 5:
    opcion5(pieza)

elif opc == 6:
    opcion6(pieza)

elif opc == 7:
    print("--- Programa finalizado ---")

# script principal...
if __name__ == '__main__':
    test()

```

El archivo en su primera línea contiene la inclusión del módulo *insumos.py* con la correspondiente instrucción *import*:

```
import insumos
```

La función *test()* del programa (ubicada casi al final del mismo) sirve como *punto de entrada*: es la función que se ejecuta desde el script principal, y contiene la **creación inicial del arreglo con *n* componentes vacíos (o sea, *None*)** y la gestión del menú principal:

```

def test():
    # cargar cantidad de insumos...
    print('Ingrese la cantidad de insumos en la pieza...')
    n = validate(0)

    # crear el arreglo (initialmente vacio)...
    pieza = n * [None]

    opc = 0
    while opc != 7:
        print('\nMenu de opciones:')
        print('1. Cargar insumos de la pieza')
        print('2. Mostrar todos los insumos de la pieza')
        print('3. Mostrar el valor total de la pieza')
        print('4. Mostrar insumos con menos de x unidades')
        print('5. Cambiar cantidad en insumos con 0 unidades')
        print('6. Buscar un insumo por su código')
        print('7. Salir')

        opc = int(input('Ingrese su elección: '))

        if opc == 1:
            opcion1(pieza)

        elif opc == 2:
            opcion2(pieza)

        elif opc == 3:
            opcion3(pieza)

```

```

        elif opc == 4:
            opcion4(pieza)

        elif opc == 5:
            opcion5(pieza)

        elif opc == 6:
            opcion6(pieza)

        elif opc == 7:
            print("--- Programa finalizado ---")

# script principal...
if __name__ == '__main__':
    test()

```

La función `opción1()` activa la carga del arreglo desde el teclado, lo cual se hace con la función `read()`, que a su vez es auxiliada por las funciones `validate()` y `validate_code()` para validar la carga de los campos numéricos. No hay demasiado misterio en este grupo de funciones que replicamos aquí:

```

def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Error... Se pido > ' + str(inf) + '... Cargue de nuevo: '))
    return n

def validate_code(mn=1, mx=20):
    cod = int(input('Ingrese código (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Error... era >=' + str(mn) + ' y <=' + str(mx) + '). De nuevo: '))
    return cod

def read(pieza):
    n = len(pieza)
    for i in range(n):
        nom = input('Nombre[' + str(i) + ']: ')
        cod = validate_code(1, 20)
        val = int(input('Precio: '))
        can = int(input('Cantidad: '))
        pieza[i] = insumos.Insumo(cod, nom, val, can)
    print()

def opción1(pieza):
    print('Cargue los datos de los insumos de la pieza:')
    read(pieza)

```

La función `opción2()` permite la visualización del contenido completo del arreglo en la pantalla. Lo primero que hace la función es comprobar si el arreglo efectivamente contiene algún dato o no, lo cual sale en forma directa comprobando el contenido del casillero 0: si el mismo es `None`, es porque el arreglo nunca fue cargado desde el teclado, y en ese caso la función termina avisando al usuario que el arreglo está vacío. Sólo si el arreglo estuviese cargado, se invoca a la función `display_all()` y esta recorre y muestra el contenido del arreglo recurriendo en cada vuelta del ciclo iterador a un `print()` que muestra la conversión a cadena de cada registro con nuestra ya analizada función `to_string()` (tomada desde el módulo `insumos.py`):

```

def display_all(pieza):
    for insumo in pieza:
        print(insumos.to_string(insumo))

def opcion2(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Listado completo de insumos para la pieza')
    display_all(pieza)

```

La función *opción3()* activa el proceso de cálculo del monto total acumulado para la pieza completa representada por el arreglo. La función comienza también comprobando si el arreglo efectivamente contiene algún registro (cosa que de aquí en adelante harán las funciones que restan para cumplir con cada opción) y luego invoca a la función *total\_value()* para hacer la acumulación, que es muy simple [3]: se recorre el arreglo con un *for iterador*, y en cada vuelta de ese ciclo se calcula el monto de cada insumo multiplicando el precio (o valor) de ese insumo por la cantidad del mismo que haya usado, acumulando ese resultado. El valor final del acumulador es el monto total y se retorna al final de la función *total\_value()*:

```

def total_value(pieza):
    tv = 0
    for insumo in pieza:
        monto = insumo.valor * insumo.cantidad
        tv += monto
    return tv

def opcion3(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Monto total en insumos para la pieza:', total_value(pieza))

```

La función *opción4()* se invoca para mostrar en pantalla un listado con los insumos cuya cantidad usada sea menor a un valor *x* que la propia función carga por teclado. Como el listado debe mostrarse ordenado de acuerdo al código de cada insumo, lo primero que se hace es entonces ordenar el arreglo completo mediante la función *sort()*, y luego se invoca a la función *display()* para mostrar sólo el subconjunto de registros que tengan un valor menor a *x* en el campo *cantidad* [3]:

```

def sort(pieza):
    n = len(pieza)
    for i in range(n-1):
        for j in range(i+1, n):
            if pieza[i].codigo > pieza[j].codigo:
                pieza[i], pieza[j] = pieza[j], pieza[i]

def display(pieza, x):
    print('Insumos con menos de', x, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad < x:
            print(insumos.to_string(insumo))

def opcion4(pieza):

```

```

if pieza[0] is None:
    print('No hay datos cargados en el arreglo...')
    return

x = int(input('Cantidad x de unidades a controlar: '))
sort(pieza)
display(pieza, x)

```

La función *opción5()* tiene el objetivo de lanzar el proceso para cambiar el valor del campo *cantidad* en todos los insumos cuya cantidad original sea cero. El nuevo valor a asignar en ese campo se carga por teclado en la variable *c*, y luego se invoca a la función *change\_quantity()* para recorrer el arreglo y producir los cambios. Esta última función recorre el arreglo con *for* iterador, y si el registro analizado en la vuelta actual tiene su campo *cantidad* valiendo cero, simplemente lo reemplaza por el valor *c*. Se usa una bandera llamada *exists* para saber (al final del ciclo) si efectivamente hubo registros modificados o no, y poder de esta forma, si fuese necesario, mostrar un mensaje avisando que no hubo cambios. Otra vez, para mostrar el listado de insumos se recurre a nuestra función *insumos.to\_string()*:

```

def change_quantity(pieza, c):
    exists = False
    print('Insumos que pasaron de 0 unidades a', c, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad == 0:
            exists = True
            insumo.cantidad = c
            print(insumos.to_string(insumo))

    if not exists:
        print('No hay insumos con 0 unidades en esta pieza')

def opcion5(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese la nueva cantidad c para los insumos sin unidades...')
    c = validate(0)
    change_quantity(pieza, c)

```

La función *opción6()* (la última) inicia el mecanismo de búsqueda de un insumo cuyo código coincida con el valor *cod* cargado por teclado. La búsqueda se hace mediante la función *search()*, que simplemente implementa un proceso de *búsqueda secuencial* [3]: si el registro buscado existe, se lo retorna completo. Y si no existe, la función *search()* retorna *None*. La función *opcion6()* chequea el valor returned por *search()* y muestra el registro en caso de haberlo hallado, o un mensaje informando que ese insumo no existía (si ese fuese el caso). Para la visualización del registro, nuevamente se usa un *print()* combinado con la función *insumos.to\_string()*:

```

def search(pieza, cod):
    for insumo in pieza:
        if insumo.codigo == cod:
            return insumo
    return None

def opcion6(pieza):
    if pieza[0] is None:

```

```

        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese el código del insumo a buscar...')
    cod = validate_code(1, 20)
    ins = search(pieza, cod)
    if ins is not None:
        print(insumos.to_string(ins))
    else:
        print('No existe un insumo con ese código en la pieza')

```

Con esto concluye el análisis de la solución para este problema. Presentamos ahora otro problema similar, para desplegar y aplicar otras técnicas elementales de procesamiento de arreglos de registros:

**Problema 49.)** Una asociación deportiva desea almacenar la información referida a sus *n* socios en un arreglo de registros (cargar *n* por teclado). Por cada socio, se pide guardar su número de identificación, su nombre, el arancel que paga cada mes y un código entre 0 y 9 para indicar el deporte que cada socio practica (suponiendo que por ejemplo, el 0 puede ser fútbol, el 1 básquet, y así hasta el código 9).

Se pide desarrollar un programa en Python controlado por un menú de opciones. Ese menú debe permitir gestionar las siguientes tareas a partir del arreglo pedido en el párrafo anterior:

- 1- Cargar el arreglo pedido con los datos de los *n* socios. Sólo valide el código del deporte practicado, para asegurar que esté entre 0 y 9.
- 2- Mostrar los datos de todos los socios que paguen un arancel mayor a *p*, siendo *p* un valor que se carga por teclado.
- 3- Determinar y mostrar cuántos socios practican cada tipo de deporte posible (un contador para contar cuántos socios practican el deporte 0, otro para los que practican el deporte 1, etc.).
- 4- Mostrar todos los datos, ordenados de menor a mayor por número de identificación.
- 5- Determinar si existe algún socio cuyo nombre sea igual a *x*, siendo *x* una cadena que se carga por teclado. Si existe, cambiar el valor del campo arancel de forma de sumarle un valor fijo de 100 pesos, y mostrar todos los datos de ese socio por pantalla. Si no existe, informar con un mensaje.

**Discusión y solución:** El programa que resuelve este problema es el modelo *test09.py* (incluido en el proyecto [F18] Registros que viene con esta Ficha). Incluye el módulo *socios.py* con la definición de la clase *Socio*, su función constructora *\_\_init\_\_()* y la ya típica función *to\_string()*:

```

class Socio:
    def __init__(self, num, nom='', ara=0.0, cod=0):
        self.numero = num
        self.nombre = nom
        self.arancel = ara
        self.codigo = cod

    def to_string(socio):
        r = ''
        r += '{:<15}'.format('Número: ' + str(socio.numero))
        r += '{:<30}'.format('Nombre: ' + socio.nombre)
        r += '{:<18}'.format('Arancel: ' + str(socio.arancel))

```

```
r += '{:<15}'.format('Codigo: ' + str(socio.codigo))
return r
```

El programa completo para resolver el problema está incluido en el archivo *test09.py* del mismo proyecto, y se transcribe a continuación:

```
import socios

def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Se pidio mayor a ' + str(inf) + '... Cargue de nuevo: '))
    return n

def validate_code(mn=0, mx=9):
    cod = int(input('Ingrese codigo (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Se pidio >=' + str(mn) + ' y <=' + str(mx) + '). De nuevo: '))
    return cod

def read(club):
    n = len(club)
    for i in range(n):
        nom = input('Nombre[' + str(i) + ']: ')
        print('Ingrese numero de socio...')
        num = validate(0)

        print('Ingrese arancel...')
        ara = validate(0)

        print('Ingrese código de deporte...')
        cod = validate_code(0, 9)

        club[i] = socios.Socio(num, nom, ara, cod)
    print()

def opcion1(club):
    print('Cargue los datos de los socios del club:')
    read(club)

def display(club, p):
    print('Listado de socios que pagan arancel mayor a', p, ':')
    for socio in club:
        if socio.arancel > p:
            print(socios.to_string(socio))

def opcion2(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese arancel para controlar...')
    p = validate(0)
    display(club, p)

def count(club):
    vc = 10 * [0]
    for socio in club:
        d = socio.codigo
        vc[d] += 1
```

```
print('Cantidad de socios en cada deporte deporte disponible:')
for i in range(10):
    if vc[i] != 0:
        print('Codigo de deporte:', i, 'Cantidad de socios:', vc[i])

def opcion3(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
    return

count(club)

def sort(club):
    n = len(club)
    for i in range(n-1):
        for j in range(i+1, n):
            if club[i].numero > club[j].numero:
                club[i], club[j] = club[j], club[i]

def display_all(club):
    print('Listado completo de socios del club:')
    for socio in club:
        print(socios.to_string(socio))

def opcion4(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
    return

sort(club)
display_all(club)

def search(club, nom):
    for socio in club:
        if socio.nombre == nom:
            return socio
    return None

def opcion5(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
    return

nom = input('Ingrese el nombre del socio a buscar: ')
socio = search(club, nom)
if socio is not None:
    socio.arancel += 100
    print('Socio encontrado... se incremento en $100 su arancel...')
    print('Datos modificados del socio:')
    print(socios.to_string(socio))
else:
    print('No existe un socio registrado con ese nombre')

def test():
    # cargar cantidad de socios...
    print('Ingrese la cantidad de socios del club...')
    n = validate(0)

    # crear el arreglo (inicialmente vacio)...
    club = n * [None]
```

```

opc = 0
while opc != 6:
    print('\nMenu de opciones:')
    print('1. Cargar socios')
    print('2. Mostrar socios que pagan arancel mayor a p')
    print('3. Conteo de socios por cada deporte')
    print('4. Listado ordenado de socios')
    print('5. Buscar socio y ajustar su arancel')
    print('6. Salir')

    opc = int(input('Ingrese su elección: '))

    if opc == 1:
        opcion1(club)

    elif opc == 2:
        opcion2(club)

    elif opc == 3:
        opcion3(club)

    elif opc == 4:
        opcion4(club)

    elif opc == 5:
        opcion5(club)

    elif opc == 6:
        print("--- Programa finalizado ---")

# script principal...
if __name__ == '__main__':
    test()

```

Las mayor parte de las tareas que deben llevarse a cabo para resolver este problema son similares a las que se expusieron en para el problema anterior, por lo que confiamos en que el alumno podrá realizar al análisis por su propia cuenta. Sin embargo, veremos con algún detalle el proceso lanzado por la función *opcion3()* para contar cuántos socios están registrados en cada uno de los 10 tipos de deportes que se ofrecen en el club.

Como esos deportes son 10 y cada uno está identificado con un número del 0 al 9, se puede aplicar la ya conocida técnica de *arreglo o vector de conteo* [3]. La función *count()* crea primero un arreglo *vc* de 10 elementos iniciados en cero, de forma que cada uno de esos elementos se use luego como un contador para cada deporte: la casilla *vc[0]* se usará para contar cuántos socios se registraron en el deporte 0, la casilla *vc[1]* se usará para contar los que se registraron en el deporte 1, y así con todas las demás. De esta forma, el conteo (que no es otra cosa que la determinación de la distribución de frecuencias de los socios por cada deporte), se realiza en forma directa: un ciclo iterador va tomando uno por uno los registros de los socios del arreglo, y se usa el valor del campo *codigo* (que contiene el número que identifica al deporte elegido por cada socio) para acceder en forma directa al casillero del arreglo *vc* donde se debe contar. Así, si un socio tiene el campo *codigo* con el valor 4, el siguiente segmento hace que se acceda a la casilla 4 del vector *vc* para incrementar en uno su valor:

```

d = socio.codigo
vc[d] += 1

```

Cuando todos los socios han sido contados de esta forma, se muestra convenientemente el vector `vc` en pantalla para producir el listado final. Para simplificar la salida, sólo se muestran los deportes que efectivamente hayan tenido al menos un socio registrado:

```
def count(club):
    vc = 10 * [0]
    for socio in club:
        d = socio.codigo
        vc[d] += 1

    print('Cantidad de socios en cada deporte disponible:')
    for i in range(10):
        if vc[i] != 0:
            print('Codigo de deporte:', i, 'Cantidad de socios:', vc[i])

def opcion3(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
    return

count(club)
```

Se deja el resto de los procesos y funciones de este programa para ser analizados por el estudiante.

## 5.] Matrices de registros en Python.

Del mismo modo que lo hecho para arreglos unidimensionales, se pueden crear arreglos bidimensionales que contengan referencias a registros si fuese necesario. La idea es esencialmente la misma que para crear una matriz de valores simples. Supongamos que el tipo registro *Estudiante* está ya definido y listo para usar en base al esquema que sigue:

```
class Estudiante:
    def __init__(self, leg, nom, prom):
        self.legajo = leg
        self.nombre = nom
        self.promedio = prom
```

Suponga también que se quiere crear una matriz `est` en la que cada fila `f` contenga los datos de los estudiantes que cursan en el año o nivel `f`, y cada columna `c` se usa para distinguir el número de orden de cada estudiante en ese nivel. Entonces la creación de esa matriz `est` de `fils` filas y `cols` columnas en la que cada casilla contenga una referencia a un registro de tipo *Estudiante* puede comenzar definiendo la matriz mediante *creación por comprensión*, en la forma siguiente [1]:

```
est = [[None] * cols for f in range(fils)]
```

La instrucción anterior crea una variable `est` de tipo *list*, que contendrá tantos elementos (a modo de `fils`) como indique la variable `fils`. Cada uno de esos elementos será a su vez una lista con `cols` elementos (a modo de columnas) valiendo `None` (relea la Sección *¡Error! No se encuentra el origen de la referencia.*, página *¡Error! Marcador no definido.* y siguientes en esta misma Ficha).

Como cada casillero de la matriz en este momento vale `None` y no hay realmente ningún registro de tipo *Estudiante*, lo siguiente *es crear esos registros y asignarlos en cada casilla*.

Un esquema de ciclos anidados como el que se muestra en la siguiente función permite lograrlo, realizando también la carga por teclado de los datos de cada registro:

```
def read(fils, cols):
    est = [[None] * cols for f in range(fils)]
    print('Ingrese los datos de cada estudiante...')
    for f in range(fils):
        print('Nivel', f, ':')
        for c in range(cols):
            print('\tEstudiante número', c)
            leg = int(input('\t\tLegajo: '))
            nom = input('\t\tNombre: ')
            pro = float(input('\t\tPromedio: '))

            est[f][c] = Estudiante(leg, nom, pro)
            print()
    return est
```

Y el resto es simple cuestión de extrapolar las técnicas de recorrido ya conocidas para procesar el contenido de la matriz, recordando que ahora cada casillero contiene un registro. Por lo tanto, y a modo de ejemplo, el acceso al campo *legajo* del registro ubicado en la casilla *est[f][c]* se realiza con el identificador *est[f][c].legajo* y luego asignando en ese campo un valor o usando su valor como parte de una expresión cualquiera. El siguiente problema o caso de análisis permite ilustrar todo lo visto, aclarar las ideas y poner en práctica los mecanismos explicados:

**Problema 50.)** *El responsable del consorcio de un pequeño barrio cerrado desea obtener una estadística de los gastos incurridos en el barrio a lo largo de un trimestre por las n propiedades que hay en el barrio. Para ello, se solicita crear tipo de registro designado como Consumo, con los campos que describan el consumo realizado en un mes dado por una propiedad: en uno de los campos almacenar el importe pagado por electricidad en ese mes, en otro el importe gastado en gas, y en otro el importe gastado en servicio medido de agua. Se debe crear una matriz cons de referencias a registros de tipo Consumo, en la que cada columna represente uno de los tres meses del trimestre analizado (numerados de 0 a 2) y en la que cada fila represente una de las n propiedades (numeradas de 0 a n-1). El registro de tipo Consumo almacenado en la casilla cons[i][j], representará entonces los gastos realizados por la propiedad i en el mes j. Se pide un que permita:*

- Cargar los datos de la matriz.
- Mostrar adecuadamente los datos de toda la matriz.
- Mostrar por pantalla un listado general en el cual se indique el gasto acumulado por cada propiedad en el trimestre (es decir, una totalización por filas de la matriz).
- Mostrar por pantalla un listado general en el cual se indique el gasto acumulado por todas las propiedades en cada mes del trimestre (es decir, una totalización por columnas de la matriz).

**Discusión y solución:** El proyecto F[18] Registros que acompaña a esta Ficha contiene un modelo *test10.py* con el programa completo que resuelve este problema.

```
class Consumo:
    def __init__(self, el, gs, ag):
        self.electricidad = el
        self.gas = gs
```

```

    self.agua = ag

def to_string(cons):
    r = ''
    r += '{:<35}'.format('Gasto de electricidad: ' + str(cons.electricidad))
    r += '{:<17}'.format('Gas: ' + str(cons.gas))
    r += '{:<17}'.format('Agua: ' + str(cons.agua))
    return r

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    if n <= inf:
        print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def read(fils, cols):
    cons = [[None] * cols for f in range(fils)]
    print('Ingrese los montos consumidos por cada rubro...')
    for f in range(fils):
        print('Propiedad', f, ':')
        for c in range(cols):
            print('\t\tMes', c)
            el = float(input('\t\tGasto en electricidad: '))
            gs = float(input('\t\tGasto en gas: '))
            ag = float(input('\t\tGasto en agua: '))
            cons[f][c] = Consumo(el, gs, ag)
            print()
    return cons

def display(cons):
    filas, columnas = len(cons), len(cons[0])
    print('Planilla de gastos mensuales por propiedad...')
    for f in range(filas):
        print('Propiedad', f)
        for c in range(columnas):
            print(to_string(cons[f][c]))
    print()

def total_per_property(cons):
    filas, columnas = len(cons), len(cons[0])
    print('Gastos por propiedad en cada trimestre')
    for f in range(filas):
        ac = 0
        for c in range(columnas):
            t = cons[f][c].electricidad + cons[f][c].gas + cons[f][c].agua
            ac += t
        print('Propiedad:', f, '\tGasto total:', ac)

def total_per_month(cons):
    filas, columnas = len(cons), len(cons[0])
    print('Gastos por mes entre todas las propiedades')
    for c in range(columnas):
        ac = 0
        for f in range(filas):
            t = cons[f][c].electricidad + cons[f][c].gas + cons[f][c].agua
            ac += t
        print('Mes:', c, '\tGasto total:', ac)

def test():

```

```

# cargar cantidad de propiedades...
print('Cantidad de propiedades - ', end=' ')
fils = validate(0)
print()

# crear y cargar la matriz de consumos...
cols = 3
cons = read(fils, cols)
print()

# mostrar todos los datos cargados...
display(cons)
print()

# acumulación por filas...
total_per_property(cons)
print()

# acumulación por columnas...
total_per_month(cons)

# script principal...
if __name__ == '__main__':
    test()

```

## 6.] Generación de un arreglo de registros con contenido aleatorio ("generación automática").

En los dos problemas que hemos presentado en lo que va de esta Ficha, se ha usado un arreglo de registros cuyo contenido inicial siempre era cargado desde el teclado. Pero en algunas ocasiones esa tarea es muy incómoda y engorrosa, ya que la carga de un conjunto de varios registros con muchos campos lleva tiempo, y cuando se están haciendo pruebas sobre el funcionamiento del programa esa pérdida de tiempo es molesta.

Por ese motivo es interesante explorar alguna forma de generar el contenido de cada campo de los registros del arreglo en forma automática, recurriendo a procesos basados en generación de valores aleatorios (que ya hemos presentado en fichas anteriores). El siguiente problema sigue en la misma línea general que los ya mostrados en esta Ficha, pero se agrega una opción para permitir esa generación automática del contenido:

**Problema 51.)** *Una empresa concesionaria de peajes está desarrollando un programa que le permitirá registrar en un arreglo de registros los datos de los n vehículos que pasan por sus cabinas (cargar n por teclado). Por cada vehículo que pasa por una cabina se toman los siguientes datos: patente del vehículo (una cadena de caracteres), tipo de vehículo (un número entero), número de cabina (es un valor numérico entre 0 y 14 que indica por cual cabina ha pasado el vehículo) y el importe pagado.*

*Desarrollar un programa en Python controlado por un menú de opciones. Ese menú debe permitir gestionar las siguientes tareas:*

- 1- *Cargar por teclado el vector validando que el número de cabina sea efectivamente un número entre 0 y 14.*
- 2- *Generar en forma automática (con valores obtenidos en forma aleatoria) los contenidos de cada registro del arreglo.*
- 3- *Mostrar todos los datos ordenados por patente de menor a mayor.*

- 4- Mostrar todos los datos de los vehículos que hayan pasado por la cabina x sin pagar peaje (es decir, los datos de los vehículos que pasaron por esa cabina y tienen registrado un importe o pago igual a 0). El valor x debe ser cargado por teclado.
- 5- Determine el importe acumulado que se registró por cada cabina (el importe acumulado por vehículos que pasaron por la cabina 0, lo mismo para la cabina 1, y así con las 15 cabinas). También determine la cantidad de vehículos que pasaron por cada cabina (un total de 15 contadores).
- 6- Cargue por teclado una patente p y determine cuántas veces la misma está registrada en el vector. Muestre todos sus datos cada vez que la encuentre, e indique al final cuántas veces aparece. Si esa patente no existe, informe con un mensaje.

**Discusión y solución:** La solución está incluida en el mismo proyecto [F18] *Registros* que acompaña a esta Ficha. Como de costumbre, la clase *Vehiculo*, su función *\_\_init\_\_()* y la función *to\_string()* se incorporan en un módulo aparte, que se llama *vehiculos.py* en el mismo proyecto:

```
class Vehiculo:
    def __init__(self, pat, tip, cab, imp=0.0):
        self.patente = pat
        self.tipo = tip
        self.cabina = cab
        self.importe = imp

    def to_string(vehiculo):
        r = ''
        r += '{:<20}'.format('Patente: ' + vehiculo.patente)
        r += '{:<20}'.format('Tipo: ' + str(vehiculo.tipo))
        r += '{:<20}'.format('Cabina: ' + str(vehiculo.cabina))
        r += '{:<20}'.format('Importe: ' + str(vehiculo.importe))
        return r
```

El programa completo está desarrollado en el archivo *test11.py* del mismo proyecto, y lo mostramos a continuación:

```
import random
import vehiculos

def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Error... Se pidió > ' + str(inf) + '... Cargue de nuevo: '))
    return n

def validate_code(mn=0, mx=14):
    cod = int(input('Ingrese código (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Se pidió > ' + str(mn) + ' y <= ' + str(mx) + '... De nuevo: '))
    return cod

def read(peaje):
    n = len(peaje)
    for i in range(n):
        pat = input('Patente[' + str(i) + ']: ')
        print('Ingrese tipo de vehículo...')
        tip = validate(0)
```

```

print('Ingrese numero de cabina...')
cab = validate_code(0, 14)

print('Ingrese importe pagado...')
imp = validate(-1)

peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)
print()

def opcion1(peaje):
    print('Cargue los datos de los vehiculos:')
    read(peaje)

def generate(peaje):
    letras = ('ABC', 'BCD', 'CDE', 'EFG')

    n = len(peaje)
    for i in range(n):
        p1 = random.choice(letras)
        p2 = str(random.randint(100, 1000))
        pat = p1 + p2

        tip = random.randint(1, 20)
        cab = random.randint(0, 14)
        imp = random.randint(0, 50)

        peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)

    print('Hecho... el arreglo ha sido generado...')

def opcion2(peaje):
    print('Se precede a la generacion automatica del arreglo... pulse <Enter>...')
    input()
    generate(peaje)

def sort(peaje):
    n = len(peaje)
    for i in range(n-1):
        for j in range(i+1, n):
            if peaje[i].patente > peaje[j].patente:
                peaje[i], peaje[j] = peaje[j], peaje[i]

def display_all(peaje):
    print('Listado completo de socios del peaje:')
    for v in peaje:
        print(vehiculos.to_string(v))

def opcion3(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    sort(peaje)
    display_all(peaje)

def display(peaje, x):
    exists = False
    print('Listado de vehiculos que pasaron por la cabina', x, 'sin pagar peaje:')
    for v in peaje:
        if v.importe == 0 and v.cabina == x:

```

```

exists = True
print(vehiculos.to_string(v))

if not exists:
    print('No hay vehiculos que hayan pasado sin pagar por esa cabina')

def opcion4(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese numero de cabina para controlar...')
    x = validate_code(0, 14)
    display(peaje, x)

def count(peaje):
    vc = 15 * [0]
    va = 15 * [0]
    for v in peaje:
        d = v.cabina
        vc[d] += 1
        va[d] += v.importe

    print('Cantidad de vehiculos e importe acumulado por cada cabina:')
    for i in range(15):
        if vc[i] != 0:
            print('Cabina:', '{:<4}'.format(str(i)), end='')
            print('Cantidad de vehiculos:', '{:<6}'.format(str(vc[i])), end='')
            print('Total recaudado:', '{:<10}'.format(str(va[i])))

def opcion5(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    count(peaje)

def search(peaje, pat):
    c = 0
    for v in peaje:
        if v.patente == pat:
            c += 1
            print(vehiculos.to_string(v))

    if c != 0:
        print('Cantidad de pasos registrados para ese vehiculo:', c)
    else:
        print('No esta registrado ese vehiculo')

def opcion6(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    pat = input('Ingrese la patente a buscar: ')
    search(peaje, pat)

def test():
    print('Ingrese la cantidad de vehiculos a cargar...')
    n = validate(0)

    peaje = n * [None]

```

```

opc = 0
while opc != 7:
    print('\nMenu de opciones:')
    print('1. Cargar vehiculos en forma manual')
    print('2. Cargar vehiculos en forma automática')
    print('3. Listado de vehiculos ordenado por patente')
    print('4. Vehiculos que pasaron por cabina x sin pagar peaje')
    print('5. Conteo de vehiculos e importe acumulado (por cabina)')
    print('6. Listado de todos los pasos de un vehiculo')
    print('7. Salir')

    opc = int(input('Ingrese su eleccion: '))

    if opc == 1:
        opcion1(peaje)

    elif opc == 2:
        opcion2(peaje)

    elif opc == 3:
        opcion3(peaje)

    elif opc == 4:
        opcion4(peaje)

    elif opc == 5:
        opcion5(peaje)

    elif opc == 6:
        opcion6(peaje)

    elif opc == 7:
        print("--- Programa finalizado ---")

# script principal...
if __name__ == '__main__':
    test()

```

El programa incluye dos instrucciones *import*: una para habilitar el acceso al módulo predefinido *random* que contiene las funciones para manejo de números aleatorios [1] [2], y el segundo para habilitar el modulo *vehiculos.py* con la definición de la clase del registro pedido por el enunciado. En general, todo el trabajo que se implementa en cada función puede ser perfectamente estudiado y entendido por los estudiantes, por lo que solamente nos concentraremos en la opción 2 del menú: la generación automática (en base a valores aleatorios) del arreglo de registros.

La función *opcion2()* es invocada desde el ciclo de control del menú principal en la función *test()*. La función simplemente pone un mensaje aclaratorio en pantalla e invoca a la función *generate(peaje)* para que sea esta la que finalmente haga el trabajo de generar el contenido del arreglo *peaje*.

```

def generate(peaje):
    letras = ('ABC', 'BCD', 'CDE', 'EFG')

    n = len(peaje)
    for i in range(n):
        p1 = random.choice(letras)
        p2 = str(random.randint(100, 999))
        pat = p1 + p2

```

```

tip = random.randint(1, 20)
cab = random.randint(0, 14)
imp = random.randint(10, 50)

peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)

print('Hecho... el arreglo ha sido generado...')

def opcion2(peaje):
    print('Generacion automatica del arreglo... pulse <Enter>...')
    input()
    generate(peaje)

```

La generación de valores aleatorios para los campos de tipo numérico no es un problema: sabemos que la función predefinida `random.randint(a, b)` retorna un número entero aleatoriamente elegido en el intervalo  $[a, b]$ , con lo que es suficiente con invocarla tres veces en nuestro caso (una vez por cada uno de los campos *tipo* (un entero mayor a cero, y que para simplificar suponemos no mayor a 20 ), *cabina* (un entero entre 0 y 14) e *importe* (también para simplificar, lo suponemos entero entre 10 y 50)). La secuencia que sigue genera esos valores convenientemente, y los almacena en tres variables locales *tip*, *cab* e *imp*:

```

tip = random.randint(1, 20)
cab = random.randint(0, 14)
imp = random.randint(10, 50)

```

Cuando el campo cuyo valor se quiere generar en forma aleatoria es una cadena de caracteres, se pueden pensar muchísimas técnicas ingeniosas para hacerlo, y la elección final dependerá de las restricciones del enunciado del problema, la forma esperada de las cadenas a crear, y el gusto y la necesidad del programador. En este modelo mostramos una forma simple, que esperamos pueda servir como disparador de creatividad para que los estudiantes piensen e implementen sus propias técnicas.

Una cadena que represente una patente argentina (según el estándar de 1994) consta de tres letras mayúsculas y luego tres dígitos. Una forma de sencilla de empezar, consiste en armar una *tupla* con algunas cadenas de tres letras que puedan ser usadas como base para generar la primera parte de la patente, y luego seleccionar alguna de esas cadenas con la función `random.choice()`:

```

letras = ('ABC', 'BCD', 'CDE', 'EFG')
p1 = random.choice(letras)

```

El pequeño script anterior selecciona en forma aleatoria una de las cuatro cadenas contenidas en la tupla *letras*, y almacena la cadena elegida en la variable *p1*. Está claro que de esta forma todas las patentes generadas comenzarán con alguna de estas cuatro cadenas, pero se puede ampliar el rango simplemente aumentando el número de cadenas base en la tupla, o crear una tupla solo con letras y luego seleccionar aleatoriamente tres de esas letras para unirlas concatenadas en un sola variable (dejamos estas ideas para ser exploradas por los estudiantes).

La generación de la parte numérica de la patente puede hacerse nuevamente con `random.randint()`, y para asegurarnos que siempre vuelva un número con exactamente tres

dígitos, el intervalo dentro del cual se debe seleccionar podría ser [100, 999], convirtiendo luego a cadena de caracteres ese número obtenido:

```
p2 = str(random.randint(100, 999))
```

La instrucción anterior deja en la variable *p2* una cadena aleatoriamente creada, compuesta por exactamente 3 dígitos. Finalmente, si la variable *p1* contiene la cadena base de tres letras y *p2* contiene la cadena de 3 dígitos, una instrucción como la que sigue permite obtener la cadena completa representando a la patente esperada, dejándola asignada en la variable *pat*:

```
pat = p1 + p2
```

Por lo tanto, la función *generate(peaje)* efectivamente llena el arreglo *peaje* con valores aleatorios en rangos correctos, incluidas la patentes:

```
def generate(peaje):
    letras = ('ABC', 'BCD', 'CDE', 'EFG')

    n = len(peaje)
    for i in range(n):
        p1 = random.choice(letras)
        p2 = str(random.randint(100, 999))
        pat = p1 + p2

        tip = random.randint(1, 20)
        cab = random.randint(0, 14)
        imp = random.randint(10, 50)

        peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)

    print('Hecho... el arreglo ha sido generado...')
```

Se deja el resto del programa para ser analizado por los estudiantes.

---

## Bibliografía

- [1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [3] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

# Ficha 19

## Pilas y Colas

### 1.] Introducción.

En distintas Fichas de estudio anteriores hemos introducido el concepto de *estructura de datos* como una colección de muchos valores almacenados al mismo tiempo en una misma variable. Hemos visto que los distintos lenguajes de programación proveen ya listos para usar distintos tipos de estructuras de datos, y que en Python algunos de esos tipos son las tuplas, los rangos, las cadenas de caracteres, las listas (a través de las que se representan arreglos) y los registros. Todos estos tipos compuestos permiten organizar datos y resultados en formas diversas, brindando además distintas maneras de acceder a los datos individuales contenidos en cada estructura creada [1].

Sin embargo, en muchos problemas un programador se enfrenta a situaciones en las que percibe que necesita usar o aplicar formas diferentes de organizar y/o acceder a sus datos. Un ejemplo que hemos analizado nos llevó a la necesidad de los arreglos bidimensionales: si cada dato del dominio del problema se identifica o define con dos números (tipo de gasto y mes, número de estudiante y número de prueba, y en definitiva, un número de fila y otro de columna) puede seguir empleando un arreglo unidimensional... pero está claro que un arreglo bidimensional permitirá aprovechar mejor la forma de identificación natural de esos datos. Si el programador desconocía la forma de trabajar con arreglos bidimensionales, una situación práctica como esta lo llevaría a explorar el lenguaje en busca de variantes y más temprano que tarde encontraría la forma de usar arreglos con un índice adicional (o más).

Si el programador descubre que necesita nuevos tipos de estructuras de datos y el lenguaje que aplica dispone de esos tipos ya listos para usar, estamos en presencia de *estructuras de datos nativas* del lenguaje. Está claro que en Python las *secuencias* de cualquier tipo (estructuras de datos subindicadas, mutables o inmutables, como las *tuplas*, los *rangos*, las *cadenas* y las *listas*), así como los *registros* y los *diccionarios*, representan *estructuras de datos nativas* [1].

Pero en muchos casos, la estructura de datos que requeriría el programador no está disponible directamente en su lenguaje de trabajo, y el programador debe entonces *implementarla* combinando elementos que sí estén disponibles en su lenguaje. Un ejemplo: los registros como concepto general de variable que puede contener campos de tipos diferentes son estructuras nativas en Python, pero si el programador necesita un registro que *específicamente* permita representar libros en un sistema de información para una biblioteca, deberá producir ese nuevo tipo *Libro*, en base al uso combinado del tipo registro nativo (empleando la construcción *class*) y declarando campos con identificadores que estrictamente describan características de un libro en ese contexto. El nuevo tipo *Libro* no existía en Python: fue introducido por el programador combinando tipos y mecanismos que *sí* existían.

Las estructuras de datos que el programador necesita pero que el lenguaje no provee en forma nativa, se designan en general como *estructuras o tipos abstractos*. En el ejemplo anterior, *los registros como estructuras generales son estructuras nativas de Python*, pero el tipo *Libro* desarrollado por el programador para un requerimiento específico representa una *estructura abstracta*. El proceso llevado a cabo por el programador para crear el nuevo tipo abstracto (*Libro* en este ejemplo), se conoce con el nombre general de *implementación* del tipo abstracto [1].

El proceso de *implementación* de tipos o estructuras abstractas es una tarea muy común en programación, ya que es también muy común que se detecte la necesidad de emplear nuevos tipos que reflejen mejor el dominio de datos de un problema. Una vez que el programador descubre que debe implementar un nuevo tipo abstracto, el paso inicial es aplicar un mecanismo conceptual designado como *mecanismo de abstracción*: es el proceso mediante el cual se intenta captar y aislar la *información* y el *comportamiento esencial* de una entidad, elemento u objeto del dominio de un problema.

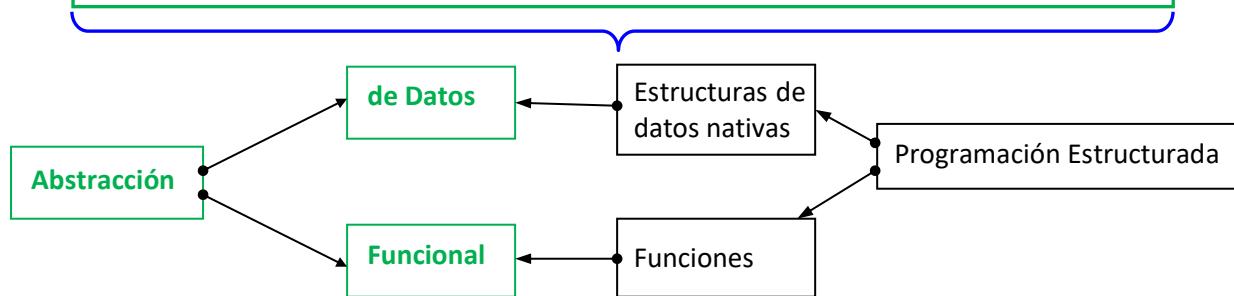
La información que se usa en un programa de computadora es una *selección simplificada de datos de la realidad* de forma tal que ese conjunto de datos se considera relevante para el problema estudiado. Por lo tanto, una *abstracción* es también una simplificación de la realidad.

A su vez, el *mecanismo de abstracción* se descompone en dos aspectos: captar y aislar los *datos relevantes* de una entidad del dominio se conoce como *abstracción de datos*, mientras que el mecanismo de identificar *procesos relevantes* se suele designar como *abstracción funcional*. Cuando aquí se habla de *datos relevantes* y de *procesos relevantes* se quiere expresar que el programador debe hacer un análisis detallado de cuáles son los datos que *realmente* necesita en su programa para representar a una entidad del enunciado o dominio del problema, y cuales son los procesos o algoritmos que *realmente* necesita implementar en su programa para procesar esos datos. La gráfica siguiente muestra un ejemplo de un esquema de abstracción aplicado al concepto de *libro* en un programa de gestión de datos para una biblioteca:

Figura 1: Esquema de un proceso de abstracción.

Ejemplo: Programa para control de préstamos de libros en una biblioteca...

- Libro  $\Rightarrow \{\text{ISBN, título, autor, copias disponibles}\}$   $\Rightarrow$  **abstracción de datos adecuada.**
- Libro  $\Rightarrow \{\text{color de tapa, peso, largo, ancho, tipo de letra}\}$   $\Rightarrow$  **abstracción de datos inadecuada.**
- Libro  $\Rightarrow \{\text{buscar(libro), prestar(libro), recibir(libro)}\}$   $\Rightarrow$  **abstracción funcional adecuada.**
- Libro  $\Rightarrow \{\text{calcular_peso(libro), superficie_tapa(libro)}\}$   $\Rightarrow$  **abstracción funcional inadecuada.**



En general dentro del paradigma de *programación estructurada* la **abstracción de datos** se realiza mediante la combinación de *estructuras de datos nativas* (comúnmente un registro agrupando distintos campos, aunque no es obligatorio el uso de registros) y la **abstracción funcional** se realiza mediante *funciones que luego tomen como parámetro a variables de los nuevos tipos*, o bien mediante *funciones que creen y retornen variables de esos tipos*.

En el ejemplo de la *Figura 1*, la entidad del dominio del problema que se quiere representar en el programa es el *libro*. Para ello, se supone que el programador definirá un registro llamado *Libro* y agregará campos al mismo para representar un libro (*abstracción de datos*). Sin embargo, no cualquier conjunto de campos será útil o relevante para su programa (y esto se determina en función del enunciado o requerimiento del problema): si el programa se usará para gestión de préstamos de libros, entonces un conjunto de campos de la forma {color de tapa, peso, largo, ancho, tipo de letra} no tendría ningún sentido (*abstracción de datos inadecuada*), pero el conjunto {ISBN, título, autor, cantidad de copias disponibles} sería relevante (*abstracción de datos adecuada*). Lo mismo pasa con el conjunto de posibles funciones a programar en ese proyecto (*abstracción funcional*) y en ese contexto: funciones para calcular el peso del libro o la superficie de la tapa del libro no serían en absoluto aplicables al contexto del problema (*abstracción funcional inadecuada*), pero funciones para buscar un libro, gestionar el préstamo, o recibir el libro en devolución serían no sólo adecuadas sino exigibles (*abstracción funcional adecuada*).

Como se dijo, una parte importante del trabajo de un programador es la implementación de estructuras abstractas. Esta Ficha está enfocada en el tratamiento e implementación de dos tipos abstractos: las *pilas* y las *colas*, que son quizás las estructuras de datos abstractas más sencillas de conceptualizar e implementar, sin que eso signifique que sean triviales sus aplicaciones.

Existen numerosas situaciones y problemas que requieren que los datos sean tratados en *orden inverso al de su entrada*, y en estos casos son especialmente útiles las *pilas*. Otros problemas requieren que los datos sean procesados en el *mismo orden en que fueron cargados*, y en casos así las *colas* son estructuras de datos muy recomendables. Ambas estructuras pueden ser implementadas con relativamente poco trabajo, y esto es particularmente cierto en Python, como veremos en las secciones que siguen.

## 2.] Pilas.

Una *pila* es una *estructura lineal* (cada elemento tiene un único sucesor y un único antecesor) en la cual los elementos se organizan de forma tal que uno de ellos se ubica *al principio* (o *al frente* o *en el tope*) de la *pila* y los demás se *enciman* o *apilan* uno sobre el otro<sup>1</sup> a partir del primero en ser insertado (y este queda ubicado al fondo de la pila) [1].

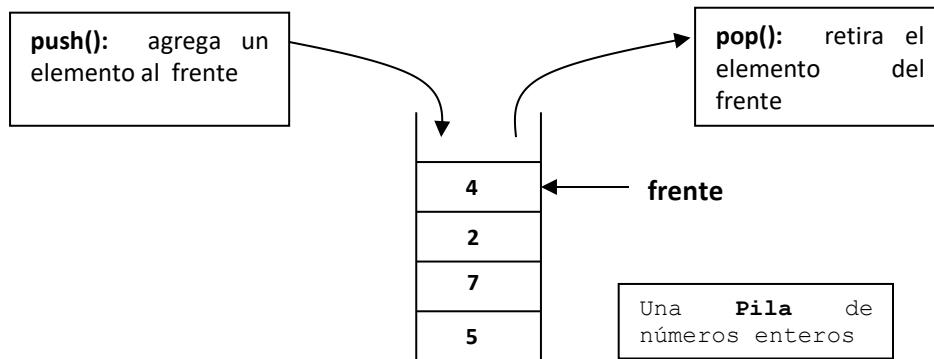
---

<sup>1</sup> Una curiosa (y morbosa...) situación de *apilamiento de cosas* se puede ver en la película *World War Z* (o *Guerra Mundial Z*) de 2013, dirigida por *Marc Forster* y protagonizada por *Brad Pitt*: en cierto momento, miles y miles de *zombies* (sí... *zombies*... como los de *The Walking Dead*) inician un ataque sobre las murallas de la ciudad de Jerusalén, y para poder escalar esas murallas simplemente se trepan unos sobre otros, formando descomunales *pilas de zombies*, hasta que logran llegar a la parte alta de la muralla y pasar al otro lado para atacar a los desprevenidos habitantes que su vez serán convertidos en *zombies*. No se puede negar que se trata de una espeluznante y original aplicación práctica para las pilas... ☺.

Si un elemento se inserta en una pila, lo hace de forma tal que queda ubicado como el elemento del frente. El único elemento que puede retirarse directamente con una sola operación, es el elemento del frente (ver *Figura 2*). Como la operación de inserción puede entenderse como una operación de *empujar hacia abajo los datos* y dejar encima al último que ingresa, se suele designar como *push()* (en inglés: *empujar*) a la función que la implementa. Y como la operación de extraer un elemento se puede entender como la *expulsión hacia arriba* del elemento que está en el tope, esa función suele designarse como *pop()* (que en inglés hace referencia a la *acción de expulsar hacia arriba*).

Las *pilas* son estructuras de datos muy útiles en situaciones donde se debe invertir una secuencia de entrada: como el último elemento en insertarse se ubica arriba de todos y sólo puede retirarse en forma directa el de arriba, entonces el último en ser insertado es el primero en ser retirado, y por lo tanto el primero en ser procesado. De este modo, si una secuencia de valores es almacenada en una *pila*, quedarán ubicados de forma que cuando comiencen a ser retirados para su procesamiento, *lo harán en orden inverso al que tenían cuando entraron*. Esta forma de procesamiento con inversión de entradas se conoce como *LIFO* (iniciales de *last in – first out*: último en entrar – primero en salir), y por ello mismo las *pilas* suelen designarse como *estructuras tipo LIFO*.

**Figura 2:** Esquema conceptual de una pila de números enteros.



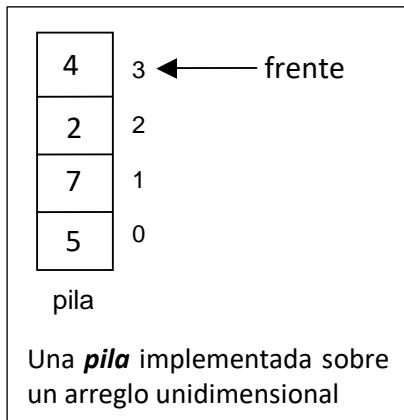
Observemos (por ejemplo) que el segmento de memoria que se conoce como *stack segment* se comporta como una *pila de bloques de memoria* para soportar el esquema de llamadas de funciones que pudiera disparar un programa. A pesar de su apariencia sencilla, como vemos las *pilas* tienen una fuerte participación en el software de base de una computadora. A lo largo de esta ficha, analizaremos algunas aplicaciones sencillas de *pilas* para resolver problemas típicos.

La forma más simple de implementar una *pila* en un programa, es usar un *arreglo unidimensional como soporte*. La idea es que los elementos de la *pila* se guarden en los casilleros de un arreglo (que por ejemplo puede llamarse *pila*), comenzando desde la posición cero. El primer elemento que se inserta en la *pila*, se guarda en el casillero cero del arreglo, y es hasta allí el elemento del frente de la *pila*. La Figura 3 (página 393) muestra la misma pila de la *Figura 2*, suponiendo que se implementa sobre un arreglo unidimensional en Python.

Si se inserta otro elemento en la *pila* (operación *push()*), se guarda en el casillero número uno del arreglo, y este pasa a ser el del frente. Si se desea obtener y eliminar el elemento del frente (operación *pop()*), sólo se debe remover el elemento que se encuentra en la última posición del arreglo de soporte y retornar ese valor (recuerde que en Python, la última

casilla también se identifica con el índice -1). Todo esto es particularmente simple de hacer en Python, ya que a diferencia de otros lenguajes, los arreglos unidimensionales en Python son de naturaleza dinámica y pueden aumentar o disminuir su tamaño en forma directa, mediante una llamada a *append()* para aumentarlo o al operador *del* para disminuirlo [2] [3]. Si el programador decidiera que el frente de la pila estuviese en la casilla 0 en lugar de en la última, entonces puede agregar un nuevo elemento con un corte de índices (*pila[0:0]*) en lugar de *append()*.

Figura 3: Esquema de una pila implementada sobre un arreglo unidimensional.



Con estas pautas, se puede definir un módulo *stack.py* en Python que contenga la implementación de una *pila* soportada en un arreglo. El módulo ni siquiera necesita declarar un registro para hacer la *abstracción de datos*: los datos de la pila estarán representados completamente por el arreglo que se envíe como parámetro a cada función del módulo, y estas funciones se encargarán de la *abstracción funcional*.

El módulo *stack.py* está incluido en el proyecto [F19] *Pilas y Colas* que acompaña a esta Ficha. Este módulo implementa varias funciones básicas para manejo de una pila representada como un arreglo unidimensional, que luego analizaremos:

```
__author__ = 'Catedra de AED'

def init():
    """Crea y retorna una pila vacía

    :return: la pila vacía creada
    """
    pila = []
    return pila

def is_empty(pila):
    """Chequea si la pila está vacía.

    :param pila: la pila a chequear.
    :return: True si pila está vacía - False en caso contrario.
    """
    n = len(pila)
    return n == 0
```

```

def peek(pila):
    """Retorna el elemento del frente de la pila, sin eliminarlo.

    :param pila: la pila en la cual se consulta.
    :return: el valor que está al frente, o None si la pila estaba vacía.
    """
    x = None
    if not is_empty(pila):
        # si frente está en la casilla 0...
        # frente = 0

        # si frente está en la última casilla...
        frente = -1
        x = pila[frente]
    return x

def push(pila, x):
    """Inserta un elemento x al frente de la pila.

    :param pila: la pila en la cual se hará la inserción.
    :param x: el elemento a insertar.
    """
    # si frente está en la casilla 0...
    # pila[0:0] = [x]

    # si frente está en la última casilla...
    pila.append(x)

def pop(pila):
    """Elimina y retorna el elemento del frente de la pila.

    :param pila: la pila en la cual se remueve.
    :return: el valor que estaba al frente, o None si la pila estaba vacía.
    """
    x = None
    if not is_empty(pila):
        # si frente está en la casilla 0...
        # frente = 0

        # si frente está en la última casilla...
        frente = -1

        x = pila[frente]
        del pila[frente]
    return x

```

La función *init()* realiza una tarea simple y directa: sólo debe crear un arreglo vacío que represente a la pila que se quiere crear, también vacía.

La operación de insertar un elemento en la *pila* es realizada por la función *push()*, la cual también es simple: si se consideró que el frente está en la casilla 0, el nuevo elemento *x* a ser insertado debe ubicarse en la casilla 0 del arreglo de soporte y correr un casillero hacia la derecha a todos los demás. Esto en Python es directo: la operación de corte de índices *pila[0:0] = [x]* hace justamente eso [2]: inserta la lista cuyo único componente es *x*, exactamente en la posición 0 del arreglo *pila*. Y si se consideró que el frente de la pila está en la última casilla, entonces sólo se debe invocar a *append()*. De acuerdo a lo que prefiera, active o desactive las líneas comentarizadas en la función.

La operación de eliminar el elemento del frente (`pop()`) hace exactamente lo contrario de la operación de insertar: elimina el elemento que en ese momento se encuentre en la casilla 0 del arreglo (si se consideró que el *frente* está en la casilla 0), y retorna ese valor para su eventual uso posterior. Simplemente se controla si el arreglo está vacío (en cuyo caso no puede removese nada de él y se retorna `None`), y en caso de contener al menos un elemento, se copia el valor de la casilla 0 en `x` (con la instrucción `x = pila[0]`) y luego se elimina el casillero 0 del arreglo, haciendo que los todos los demás se desplacen a la izquierda una posición (con la instrucción `del pila[0]`) [2]. La eliminación se hace de la misma forma si se consideró que el frente estaba en la última casilla, haciendo exactamente lo mismo pero en la casilla `pila[-1]`. De acuerdo a lo que prefiera, active o desactive las líneas comentarizadas en la función.

Se incluyó además una función `peek()` que retorna el valor del elemento del frente, pero sin removerlo de la *pila*, más una función `is_empty()` para determinar si la *pila* está vacía o no. Ambos son muy sencillos y se deja su análisis para el alumno.

El modelo `test01.py` incluido en el mismo proyecto [F19].Pilas y Colas contiene un pequeño programa para probar todas estas funciones. De nuevo, dejamos al alumno la tarea de analizar su funcionamiento:

```
import stack

__author__ = 'Cátedra de AED'

def test():
    p1 = stack.init()
    stack.push(p1, 5)
    stack.push(p1, 7)
    stack.push(p1, 2)
    stack.push(p1, 4)
    print('Estado actual de la pila:', p1)

    y = stack.pop(p1)
    print('Elemento removido del frente:', y)

    x = stack.peek(p1)
    print('Elemento actual al frente:', x)

    print('Estado actual de la pila:', p1)
    if stack.is_empty(p1):
        print('La pila está vacía...')
    else:
        print('La pila no está vacía')

    while not stack.is_empty(p1):
        print('Elemento removido:', stack.pop(p1))

    print('Estado actual de la pila:', p1)
    if stack.is_empty(p1):
        print('La pila está vacía...')
    else:
        print('La pila no está vacía')

if __name__ == '__main__':
    test()
```

### 3.] Colas.

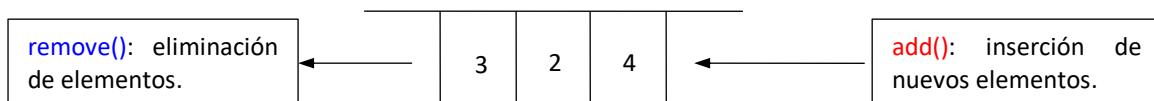
Una *cola* es una estructura lineal (cada elemento tiene un único antecesor y un único sucesor) en la cual los elementos se organizan uno detrás del otro, quedando uno de ellos al principio (o *al frente*) de la *cola* y otro en el último lugar de la misma (o *al fondo*). Para insertar un elemento (operación *add()*) se hace de forma tal que el nuevo componente queda último en la *cola*. Para eliminar un elemento (operación *remove()*) sólo puede eliminarse aquél que está primero (ver Figura 4 más abajo). Entonces, como cada elemento que se inserta queda último en la fila y por ese motivo será también el último en ser retirado, las *colas* suelen designarse también genéricamente como estructuras tipo **FIFO** (por *First In – First Out: primero en entrar, primero en salir*) [1].

Las *colas* son estructuras de datos muy útiles en programas que requieren efectuar lo que se denomina una *simulación de situaciones de espera frente a un puesto de servicio*. En estos programas se supone la existencia de un *proveedor de servicios* (por ejemplo, una ventanilla de un cajero de un banco, un puesto de peaje en una ruta, un programa controlador de entrada a una impresora en un sistema de red, etc.), y se busca simular la llegada de *clientes* para ese servicio, los cuales *forman una cola (o fila)* de espera hasta que les toca ser atendidos.

Los programas de simulación de este tipo buscan estudiar el comportamiento de estas colas en cuanto a diversos resultados. Por ejemplo, puede pedirse determinar cual es el tiempo promedio de espera de un auto en una cola hasta que es atendido en la ventanilla de peaje, o puede pedirse determinar si hace falta abrir más ventanillas de cobro a una hora determinada en un banco de acuerdo a la cantidad de clientes que hay en las colas a esa hora.

La realización de programas de simulación como los descriptos escapa a los alcances de este curso. Nos limitaremos sólo a mostrar la forma básica de implementar una cola en el lenguaje Python, y su aplicación en programas simples.

**Figura 4:** Esquema conceptual de una cola de números enteros.



Al igual que las pilas, una *cola* también puede implementarse usando un arreglo como soporte, y otra vez esto es directo y relativamente sencillo en Python, favorecido por el hecho de gestionar arreglos como variables de tipo *list*, y tener con ellos la propiedad del crecimiento y del decrecimiento en forma dinámica. La idea es exactamente la misma que hemos sugerido para las pilas, pero ahora simplemente cambiando la forma de trabajo de la función de inserción de un elemento para que lo haga en el extremo opuesto al que se use para retirar un elemento: lo común es que las inserciones se hagan a partir de la última casilla de la derecha (usando *append()*) y las eliminaciones se hagan a partir de la primera casilla de la izquierda (usando *del*) [2] [3].

El módulo *queue.py* incluido en el proyecto [F19] *Pilas y Colas* implementa todas las funciones necesarias para la gestión de una *cola* con estas ideas. El parecido con el módulo *stack.py* salta a la vista, por lo cual dejamos su análisis para el alumno:

```

__author__ = 'Cátedra de AED'

def init():
    """Crea y retorna una cola vacía

    :return: la cola vacía creada
    """
    cola = []
    return cola

def is_empty(cola):
    """Chequea si la cola está vacía.

    :param cola: la cola a chequear.
    :return: True si cola está vacía - False en caso contrario.
    """
    n = len(cola)
    return n == 0

def peek(cola):
    """Retorna el elemento del frente de la cola, sin eliminarlo.

    :param cola: la cola en la cual se consulta.
    :return: el valor que está al frente, o None si la cola estaba vacía.
    """
    x = None
    if not is_empty(cola):
        x = cola[0]
    return x

def add(cola, x):
    """Inserta un elemento x al fondo de la cola.

    :param cola: la cola en la cual se hará la inserción.
    :param x: el elemento a insertar.
    """
    cola.append(x)

def remove(cola):
    """Elimina y retorna el elemento del frente de la cola.

    :param cola: la cola en la cual se remueve.
    :return: el valor que estaba al frente, o None si la cola estaba vacía.
    """
    x = None
    if not is_empty(cola):
        x = cola[0]
        del cola[0]
    return x

```

El modelo *test03.py* incluido en el mismo proyecto, contiene un programa para aplicar cada una de las funciones vistas, pero ahora en una *cola* de números:

```
__author__ = 'Cátedra de AED'
```

```

import queue

def test():
    p1 = queue.init()
    queue.add(p1, 5)
    queue.add(p1, 7)
    queue.add(p1, 2)
    queue.add(p1, 4)
    print('Estado actual de la cola:', p1)

    y = queue.remove(p1)
    print('Elemento removido del frente:', y)

    x = queue.peek(p1)
    print('Elemento actual al frente:', x)

    print('Estado actual de la cola:', p1)
    if queue.is_empty(p1):
        print('La cola está vacía...')
    else:
        print('La cola no está vacía')

    while not queue.is_empty(p1):
        print('Elemento removido:', queue.remove(p1))

    print('Estado actual de la cola:', p1)
    if queue.is_empty(p1):
        print('La cola está vacía...')
    else:
        print('La cola no está vacía')

if __name__ == '__main__':
    test()

```

#### 4.] Uso de pilas en problemas de control de simetría.

Un interesante ejercicio de aplicación de pilas para controlar si una secuencia de datos presenta elementos de simetría, puede usarse para cerrar esta ficha. Mostramos directamente el enunciado y la solución propuesta.

**Problema 52.)** *Desarrollar un programa que cargue por teclado una cadena de caracteres y use una pila para determinar si esa cadena es capicúa: esto es, queremos determinar si una cadena de caracteres que se recibe para analizar puede leerse igual en dirección izquierda-derecha que en dirección derecha-izquierda.*

**Discusión y solución:** El proyecto [F19] Pilas y Colas que acompaña a esta Ficha contiene un modelo *test02.py* con el programa completo que resuelve este problema.

Usando una *pila* la solución es directa: la idea es que se guardan en la *pila* los caracteres de la mitad izquierda de la palabra, uno a uno tomados en el mismo orden en que aparecen en la cadena. Al guardarlas en la pila, el *orden se invierte*: el carácter que entró primero se va al fondo de la *pila*, y el que entró último queda al frente.

Luego se recorre la segunda mitad de la cadena, y por cada carácter que se tenga, se extrae a su vez un carácter de la *pila*. Si la cadena era capicúa, los caracteres extraídos de la *pila*

deberán coincidir siempre con los caracteres que se recorren en la segunda mitad de la cadena. Si algún par de caracteres no coincide, la palabra ***no es capicúa*** y el proceso se detiene. Se deja el estudio de los detalles para el alumno, o para su discusión en clase:

```

__author__ = 'Cátedra de AED'

import stack

def capicua(s):
    n = len(s)
    a = stack.init()

    mitad = n // 2
    if n % 2 == 1:
        d = mitad + 1
    else:
        d = mitad

    # fase 1: almacenar en pila la mitad izquierda de la cadena...
    for i in range(mitad):
        stack.push(a, s[i])

    # fase 2: recorrer la mitad derecha de la cadena y controlar
    # con lo que sale de la pila...
    for i in range(d, n):
        x = stack.pop(a)
        if x != s[i]:
            return False
    return True

def test():
    s = input('Ingrese la cadena a analizar: ')
    res = capicua(s)
    if res:
        print('La cadena es capicúa...')
    else:
        print('La cadena no es capicúa...')

if __name__ == '__main__':
    test()

```

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 20

## Análisis de Algoritmos – Introducción

### 1.] Introducción y conceptos básicos.

A lo largo de estas fichas de clase hemos visto diversos algoritmos para resolver numerosos problemas. Se hizo evidente (y lo será cada vez en mayor medida) que para el mismo problema pueden plantearse diferentes algoritmos. Por ejemplo, el problema de buscar un valor en un arreglo podía resolverse buscando en *forma secuencial* o en *forma binaria*, y también existen muchos algoritmos diferentes para ordenar un arreglo. La cuestión entonces es la siguiente: si se dispone de varios algoritmos para resolver el mismo problema, ¿cómo comparar el rendimiento de cada uno para decidir cuál aplicar en una situación concreta? Dedicaremos el último apartado de este capítulo justamente a presentar conceptos esenciales de *análisis de algoritmos*, que nos permitan poder hacer esas comparaciones [1].

En general, dado un problema y varios algoritmos para resolverlo, se busca comparar el rendimiento de esos algoritmos en cuanto a algún *parámetro de eficiencia*. Normalmente, los dos parámetros más usados son el *tiempo de ejecución esperado* (o sea, qué tan veloz es de esperar que sea cada algoritmo), y el *espacio de memoria empleado* por cada uno. Lo ideal sería lograr algoritmos que usando la menor cantidad posible de espacio de memoria sean a su vez muy veloces, pero en la práctica ambos parámetros suelen estar en relación inversa: si se desea mucha velocidad, el precio suele ser un mayor uso de memoria, y viceversa [2]. Sin embargo esto no es una regla terminante: hemos visto que el algoritmo de búsqueda binaria es mucho más veloz que el de búsqueda secuencial, y sin embargo ambos algoritmos usan casi la misma cantidad de memoria (el vector en el cual se procede a buscar, y un pequeño número de variables locales auxiliares).

En la práctica, el parámetro de eficiencia más analizado es el del *tiempo de ejecución*, simplemente porque se supone que el espacio de memoria será aproximadamente el mismo en todos los algoritmos diseñados para ese problema, o que las diferencias serán apenas irrelevantes. Por otra parte, el análisis comparativo suele centrarse en dos situaciones: el rendimiento del algoritmo en el *caso promedio*, y el rendimiento del mismo en el *peor caso*. Ambas situaciones se refieren respectivamente al comportamiento del algoritmo evaluado cuando se presenta *la configuración de datos más común* (o *caso promedio*, que suele ser aquella que surge de tomar los datos en orden estrictamente aleatorio) o bien cuando se presenta *la configuración de datos más desfavorable* (o *peor caso*) [2].

Por ejemplo, es claro que para la búsqueda secuencial el *peor caso* se da cuando el valor a buscar no está en el arreglo, o se encuentra al final o muy cerca del final del mismo, porque en ese caso se deben comprobar los  $n$  elementos (o casi todos ellos) para terminar la búsqueda. En muchas ocasiones el análisis de algoritmos se centra sólo en el *peor caso*, simplemente porque el análisis en el *caso promedio* suele ser muy complejo, o bien porque se adopta un criterio "pesimista" (o sea, suponer que el peor caso se presentará con mucha

frecuencia, y en todo caso, el algoritmo debe plantearse para funcionar incluso en ese **peor caso...**)

En todas las situaciones, lo que se busca es comparar los rendimientos relativos entre los diferentes algoritmos y no tanto poder medir en forma numérica y minuciosa el rendimiento de cada uno. Es decir, nos interesa poder *demostrar* que el algoritmo de búsqueda secuencial será menos veloz que el de búsqueda binaria en ciertas condiciones, sin tener que tomar un cronómetro (o usar funciones de medición de tiempo del lenguaje usado) y medir los tiempos de ejecución cada vez que queramos estar seguros de lo mismo. Por ese motivo, se busca poder deducir alguna **fórmula o expresión matemática** que permita modelar el comportamiento del algoritmo en cuanto al tiempo o el espacio empleado, y luego, sabiendo qué formulas describen mejor a cada algoritmo, se comparan los comportamientos de la funciones o relaciones representadas por esas fórmulas.

Para la deducción de esas fórmulas, se parte del hecho que cada algoritmo tiene lo que podría llamarse un *tamaño* o *volumen* natural. Ese tamaño suele venir dado por el número de datos que debe procesar. Por ejemplo, si pretendemos ordenar un arreglo, el tamaño de ese problema es obviamente el valor  $n$  que indica cuántos elementos tiene el arreglo. Lo mismo vale para el problema de la búsqueda en un arreglo. En ciertos problemas, pueden usarse otros elementos para dar el tamaño del problema pero en lo que sigue supondremos que viene dado por la cantidad  $n$  de datos a procesar [2].

Sabiendo el tamaño  $n$  del lote de datos, se intenta deducir qué fórmula se adapta mejor a las variaciones del tiempo (o el espacio usado) cuando varía  $n$ , suponiendo los datos configurados en el *caso promedio* o en el **peor caso**. Limitaremos todo nuestro estudio siguiente al análisis del **peor caso**, por ser más simple de plantear.

Veámoslo a partir de un ejemplo: El algoritmo de *búsqueda secuencial* en un arreglo de tamaño  $n$ , tiene su **peor caso** cuando el valor a buscar está muy al final (o no está). Podemos ver que el *tiempo total* que insumirá el algoritmo en ese caso, es aproximadamente el que corresponda a efectuar exactamente **todas** las comparaciones posibles, **o sea,  $n$  comparaciones**. Se dice entonces que el tiempo esperado para el algoritmo de búsqueda secuencial en el **peor caso**, **está en el orden de  $n$**  (o sea, nunca demorará más de lo que demore en hacer  $n$  comparaciones). Esto suele denotarse simbólicamente, expresando que el tiempo esperado para el algoritmo es  **$O(n)$**  (líase: *orden n* u *orden de n*). El  **$O(n)$**  se conoce también como *orden lineal* [2].

En definitiva: **no esperamos que nos den el número medido en milésimas de segundo, sino una expresión que nos muestre una cota superior para el tiempo de ejecución** (en realidad, la *mejor* cota superior). Esta forma de expresar el rendimiento de un algoritmo (ya sea para el tiempo, para el espacio, o para el parámetro que se use), se designa como **notación O** (se lee como "notación O mayúscula" o también como "notación Big O") [2] [3].

Un análisis similar puede hacerse para la búsqueda binaria. Este algoritmo partirá en dos al arreglo tantas veces como sea necesario, quedándose con un segmento e ignorando al otro, hasta dar con el valor buscado. En el **peor caso**, ese valor no estará en el arreglo y deberán hacerse todas las particiones posibles, efectuando una comparación en cada partición que no sea desecharla. Entonces, la cantidad de comparaciones en el peor caso es aproximadamente igual al número de veces que se divide por dos al vector.

Puede probarse que dado un número  $n > 1$ , la cantidad de veces que podemos dividir por dos hasta obtener un cociente de 1 es igual al logaritmo de  $n$  en base dos (o sea,  $\log_2(n)$  o bien,  $\log(n)$ ) asumiendo que cuando la base del logaritmo no se escribe es entonces igual a 2). De allí que la búsqueda binaria a lo sumo realizará  $\log(n)$  particiones, y por lo tanto el tiempo de ejecución de ese algoritmo en el peor caso es  $O(\log(n))$ . También se dice que ese algoritmo tiene tiempo de ejecución de *orden logarítmico*. Un análisis más detallado de la búsqueda binaria revela que la misma en realidad hace  $\log(n) + 1$  comparaciones, por lo que también podría decirse que el tiempo de ejecución es realmente  $O(\log(n) + 1)$ . Sin embargo, es común en notación  $O$  que las constantes se supriman (en un volumen muy grande de datos las constantes pueden despreciarse), se termina diciendo que la búsqueda binaria es  $O(\log(n))$  [2] [3].

En general, todo algoritmo que aplique el criterio de dividir sucesivamente por dos al lote de datos, quedándose con la mitad y desecharando la otra en cada partición, tendrá *orden logarítmico*. En informática cada vez que un logaritmo aparece se supone que la base del mismo es dos, pero también puede probarse que en notación  $O$  la base del logaritmo carece de importancia.

Y bien: la búsqueda secuencial es  $O(n)$  en el peor caso, mientras que la búsqueda binaria es  $O(\log(n))$  también en el peor caso. ¿Qué significa esto? Es simple: si el arreglo tiene  $n = 1000$  elementos, la búsqueda secuencial insumirá 1000 comparaciones en el peor caso (con el tiempo que sea que eso implique en la máquina donde corra), mientras que la búsqueda binaria no hará más de 9 o 10 comparaciones en el mismo peor caso. Y se pone mejor: a medida que  $n$  crece, el logaritmo también crece, pero lo hace a un ritmo de crecimiento muy suave... Si  $n = 100000$  (cien mil), esa misma cantidad de comparaciones insumirá la búsqueda secuencial, pero la binaria hará a lo sumo 16... Quiere decir que si pueden diseñarse algoritmos cuyo comportamiento sea logarítmico, se podrá estar seguro que esos algoritmos serán básicamente eficientes en cuanto al tiempo, y muy estables a medida que el número de datos crece.

Por supuesto, existen muchas funciones de orden posibles aunque algunas son muy típicas y frecuentes. Por ejemplo, analicemos intuitivamente el caso del *ordenamiento por Selección Directa* visto en una ficha anterior. Prescindiendo de constantes, básicamente se trata de dos ciclos *for* anidados, de forma que el primero de ellos hace aproximadamente  $n$  vueltas, y el segundo hace aproximadamente otras  $n$  por cada una que da el primero. Claramente, esto lleva a un esquema de  $n * n$  repeticiones, de forma que en cada una de ellas se hace una comparación. Esto sugiere que el total de comparaciones estará en el *orden  $n$  al cuadrado*, con lo que el tiempo de ejecución también será  $O(n^2)$ . En general, cada vez que se presenten dos ciclos anidados con aproximadamente  $n$  repeticiones cada uno, tendremos *orden cuadrático*.

Evidentemente, lo ideal sería que un algoritmo o acción demore siempre lo mismo para procesar un lote de datos, sin importar si aumenta el valor del tamaño  $n$  de ese lote. Los algoritmos que hemos visto en las primera fichas (antes de llegar a estudiar el uso de ciclos) son de este tipo: se cargaba siempre la misma cantidad de datos (sin posibilidad de alterar esa cantidad), y por lo tanto la demora era siempre la misma.

Sin embargo, esos ejemplos no son muy significativos pues la cantidad de datos era constante. ¿Qué algoritmos o procesos admitirán que  $n$  crezca de una corrida a la otra sin alterar su tiempo de ejecución? Por ahora, el único caso que conocemos fue analizado

también fichas anteriores: el acceso a un componente individual de un arreglo. Como sabemos, si queremos acceder al valor en la componente  $i$  de un arreglo  $v$ , sólo debemos escribir  $v[i]$  y con esa expresión se accederá al componente en el mismo orden de tiempo cada vez, sin importar si el arreglo tiene 2, 3 o 1000 elementos. Cuando un algoritmo o proceso se comporta de esta forma, denotamos su tiempo como  $O(1)$  (léase: *orden uno* u *orden proporcional a uno*). También se dice que dicho algoritmo tiene tiempo de ejecución de *orden constante*.

Para terminar esta somera introducción al tema del análisis de algoritmos, exponemos una clasificación de las principales y más elementales funciones de orden que suelen aparecer, sin que esto signifique que sean las únicas (más adelante, en otra sección de esta misma ficha, analizaremos con más detalle estas mismas funciones típicas). La última columna de la tabla indica algunos algoritmos o casos que responden a cada función de orden citada [2]:

**Tabla 1: Funciones típicas en el análisis de algoritmos.**

Función	Significado (cuando mide tiempo de ejecución)	Casos típicos
$O(1)$	Orden constante. El tiempo de ejecución es constante, sin importar si crece el volumen de datos.	Acceso directo a un componente de un arreglo.
$O(\log(n))$	Orden logarítmico. Surge típicamente en algoritmos que dividen sucesivamente por dos un lote de datos, desechar una parte y procesando la otra.	Búsqueda binaria.
$O(n)$	Orden lineal. Se da cuando cada uno de los datos debe ser procesado una vez.	Búsqueda secuencial. Recorrido completo de un arreglo.
$O(n \cdot \log(n))$	Surge típicamente en algoritmos que dividen el lote de datos, procesando cada partición sin desechar ninguna, y combinando los resultados al final. No hemos analizado aún algoritmos que respondan a este orden.	Ordenamiento Rápido (Quick Sort).
$O(n^2)$	Orden cuadrático. Típico de algoritmos que combinan dos ciclos de $n$ vueltas cada uno.	Ordenamiento por Selección Directa.
$O(n^3)$	Orden cúbico. Típico de algoritmos que combinan tres ciclos de $n$ repeticiones cada uno. No hemos analizado aún algoritmos que respondan a ese orden.	Multiplicación de matrices.
$O(2^n)$	Orden exponencial. Algoritmos que deben explorar una por una todas las posibles combinaciones de soluciones cuando el número de soluciones crece en forma exponencial.	Problema del viajante. Solución recursiva de la Sucesión de Fibonacci.

## 2.] Noción intuitiva de la notación Big O.

Hemos dicho que una de las motivaciones del *análisis de algoritmos* es la posibilidad realizar *comparaciones de rendimiento* entre distintos algoritmos planteados para resolver el mismo problema. Pero incluso si no se busca en forma inmediata esa comparación, el hecho es que contar con un elemento formal de medición que indique qué tan eficiente es un algoritmo respecto de cierto factor (como el tiempo de ejecución o el consumo de memoria) prepara al programador para tomar decisiones a futuro, cuando efectivamente deba seleccionar el mejor algoritmo para el problema que enfrente, o para estimar en forma correcta los parámetros de uso de ese algoritmo (por caso, para saber si podrá ejecutar ese algoritmo en

una computadora con determinada cantidad de memoria, sabiendo el consumo de memoria que el algoritmo reclama).

Como sabemos, en general los factores de medición de eficiencia empleados para el análisis de un algoritmo son el *tiempo de ejecución* esperado y el *consumo de memoria*, aunque en ocasiones también influye un tercer factor, como es la *complejidad aparente del código fuente* (intuitivamente, si dos algoritmos para resolver un problema tienen tiempos de ejecución y consumo de memoria similares, entonces posiblemente se elegirá el más compacto, claro y sencillo en cuanto a código fuente) [2].

También dijimos que en la práctica, el parámetro de eficiencia más analizado es el del *tiempo de ejecución*, y que el análisis comparativo suele centrarse en dos situaciones: el rendimiento del algoritmo en el *caso promedio*, y su rendimiento en el *peor caso*. El primero se refiere al comportamiento del algoritmo cuando se presenta *la configuración de datos más común* (o *caso promedio*, que suele ser aquella que surge de tomar los datos en orden estrictamente aleatorio) y el segundo cuando se presenta *la configuración de datos más desfavorable* (o *peor caso*) [2].

El análisis procede (en la medida de lo posible) intentando deducir alguna *fórmula* o *expresión matemática* que permita modelar formalmente el comportamiento del algoritmo en cuanto al tiempo o el espacio empleado. Para la deducción de esas fórmulas, se parte del *tamaño* o *volumen* natural del problema, que suele venir dado por el número de datos que se deben procesar (por ejemplo, el tamaño  $n$  de un arreglo), y se intenta deducir qué fórmula se adapta mejor a las variaciones del tiempo (o el espacio usado) cuando varía  $n$ , suponiendo el *caso promedio* o el *peor caso*. En general, limitaremos todo nuestro estudio siguiente al análisis del *peor caso*, por ser más simple de plantear.

Vimos el ejemplo del algoritmo de *búsqueda secuencial* en un arreglo de tamaño  $n$ : su peor caso se da cuando el valor a buscar está muy al final (o no está), ya que entonces el *tiempo total* que insumirá el algoritmo es aproximadamente el que corresponda a efectuar exactamente *todas* las  $n$  comparaciones posibles con lo que entonces el tiempo esperado para el algoritmo de búsqueda secuencial en el *peor caso*, *está en el orden de  $n$* . Y vimos que esto suele denotarse simbólicamente, expresando que el tiempo esperado para el algoritmo es  $O(n)$  (*orden  $n$* , *orden de  $n$* , u *orden lineal*).

Esta forma genérica de expresar el rendimiento de un algoritmo (ya sea para el tiempo o para el espacio), se designa como *notación O* (se lee como *notación O mayúscula* o también como *notación Big O*) [2] [3]. Vimos en esta misma ficha que el algoritmo de búsqueda binaria tiene un tiempo de ejecución del orden del *logaritmo de  $n$*  ( $O(\log(n))$ ) para el peor caso o que el algoritmo de ordenamiento por selección directa ejecuta en un tiempo  $O(n^2)$ , y que un algoritmo o proceso cuyo tiempo de ejecución es siempre el mismo, sin importar el tamaño del problema (por ejemplo, el acceso a una casilla individual de un arreglo o la ejecución de una asignación simple), tiene tiempo de ejecución constante [1] y se denota como  $O(1)$ .

Muchos de los algoritmos que aparecen con más frecuencia en el estudio de las estructuras de datos tienen tiempos de ejecución que para el peor caso se comportan en *el orden de funciones* muy comunes y conocidas. Como vimos, los más típicos de esos órdenes son los siguientes (aunque no los únicos: cualquier otra función podría aparecer) [2]:

- **$O(1)$ :** Un algoritmo con tiempo de ejecución *en el orden de uno* (o *proporcional a 1*), tiene un *tiempo de ejecución constante*, *sin importar el número de datos n*. Esto es claramente lo ideal al diseñar un algoritmo, pues no importa lo que crezca  $n$ , el tiempo de ejecución será siempre el mismo. Un ejemplo claro de una operación que es  $O(1)$  en la práctica, es el acceso directo a un componente de un arreglo: no importa cuántos elementos tenga el arreglo (o sea, no importa el valor de  $n$ ), el tiempo para acceder en forma directa a un componente es siempre el mismo.
- **$O(\log(n))$ :** Un algoritmo cuyo tiempo de ejecución sea del *orden del logaritmo de n*, será *ligeramente más lento a medida que n sea mayor*. Es un orden muy satisfactorio en la práctica, si puede lograrse. Los algoritmos que generalmente tienen tiempos de *orden logarítmico* son los que resuelven un problema de gran tamaño procediendo a transformarlo en uno más pequeño, dividiéndolo por alguna fracción constante. Por ejemplo, el *algoritmo de búsqueda binaria* en un arreglo ordenado tiene  $O(\log(n))$  en el peor caso.
- **$O(n)$ :** Un algoritmo cuyo *tiempo de ejecución es del orden de n* (también se dice que su tiempo de ejecución es *lineal*), es aquél que para cada elemento de entrada realiza una pequeña cantidad de procesos iguales. Un caso típico de un algoritmo cuyo orden es *lineal*, es el de la *búsqueda secuencial en un arreglo*.
- **$O(n * \log(n))$ :** Se da en *algoritmos que resuelven un problema dividiéndolo en pequeños subproblemas, resolviéndolos en forma independiente y combinando después las soluciones*. Esta estrategia se conoce como *divide y vencerás*, y será analizada con detalle en lo que resta del curso. El algoritmo de ordenamiento *Quicksort* (en el caso promedio) tiene este rendimiento y está basado en la estrategia *divide y vencerás*.
- **$O(n^2)$ :** Un algoritmo con *tiempo de ejecución de orden cuadrático*, sólo tiene utilidad práctica en problemas relativamente pequeños (o sea, con  $n$  pequeño). El tiempo de ejecución del orden de  $n^2$  suele aparecer en *algoritmos que procesan pares de elementos de datos, y la forma típica de estos algoritmos incluye un par de ciclos anidados*. Se mencionó ya que todos los métodos de ordenamiento directos tienen ese tiempo de ejecución en el peor caso.
- **$O(n^3)$ :** Un algoritmo de tiempo de ejecución de *orden cúbico* tampoco es muy práctico, salvo en *casos de problemas muy pequeños*. La forma típica de estos algoritmos incluye *tres ciclos anidados*. A modo de ejemplo, es de orden cúbico el tiempo de ejecución del popular algoritmo que permite multiplicar dos matrices entre sí.
- **$O(2^n)$ :** Los algoritmos con *orden de ejecución exponencial* son muy poco útiles en la práctica. Sin embargo, aparecen con frecuencia en casos de algoritmos del tipo de *fuerza bruta*, en los que todas las soluciones posibles son investigadas una por una. Suelen aparecer tiempos de este orden en problemas de optimización de soluciones, y en esos casos se considera todo un éxito el poder replantear un algoritmo de modo de lograr tiempos cuadráticos o cúbicos...

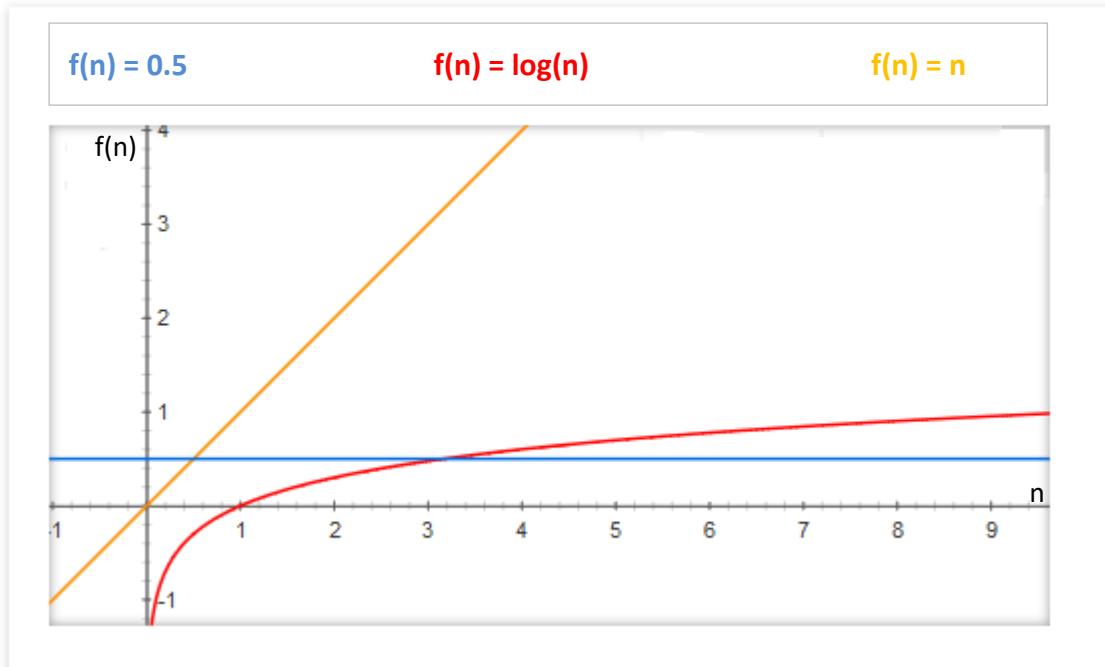
### 3.] Forma de crecimiento de las funciones clásicas de orden de complejidad.

Las funciones típicas que hemos mostrado en las secciones anteriores se han listado en orden de menor a mayor de acuerdo a la *tasa de variación* de cada una cuando  $n$  se hace grande o muy grande (que es cuando realmente tiene valor el análisis del comportamiento de un algoritmo). Es decir que para  $n$  grande o muy grande, se tiene que:

$$O(1) < O(\log(n)) < O(n) < O(n * \log(n)) < O(n^2) < O(n^3) < O(2^n)$$

La gráfica siguiente (desarrollada con el *graficador de funciones de Google*<sup>1</sup>) muestra el comportamiento de las tres primeras. Para la función constante hemos seleccionado graficar  $f(n) = 0.5$  (en este contexto, tiempo de ejecución constante u  $O(1)$  significa que el algoritmo siempre tendrá el mismo tiempo de ejecución, y no es relevante cuál sea el valor real de esa constante siempre y cuando se trate de un valor razonable para un computador):

Figura 1: Gráfica general de las funciones  $f(n) = 0.5$  -  $f(n) = \log(n)$  -  $f(n) = n$



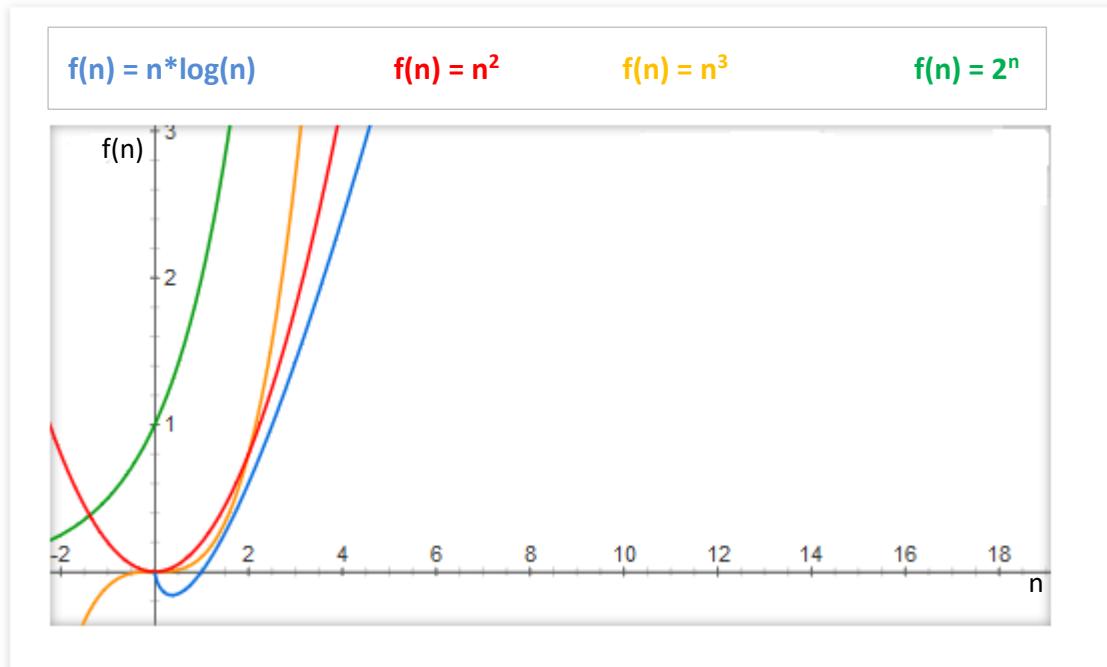
La gráfica anterior muestra que para valores bajos de  $n$ , las tres curvas se comportan en forma aceptable (si se están representando tiempos de ejecución) y que incluso la *curva del logaritmo* (en color rojo) parece mejor que las otras dos. Pero lo realmente importante es el comportamiento de las tres cuando  $n$  es grande (en la gráfica es suficiente con mirar lo que ocurre para  $n > 3$ ): en ese caso, la curva de  $f(n) = n$  (en naranja) muestra claramente tiempos mucho mayores a los de las otras dos; y la función  $f(n) = \log(n)$  se vuelve mayor que  $f(n) = 0.5$  (en celeste) y permanecerá mayor (ya que la función  $f(n) = \log(n)$  es monótona creciente).

A su vez, la gráfica que sigue (otra vez: desarrollada con el *graficador de funciones de Google* ya citado) muestra el comportamiento de las últimas cuatro funciones de nuestra tabla. De nuevo, para valores pequeños de  $n$  podrían parecer aceptables los tiempos de respuesta, e incluso pudiera parecer mejor la función *cúbica* (en color naranja) que la *cuadrática* (en color rojo). Pero cuando  $n$  crece las cosas se ponen en su lugar: claramente la función *exponencial* (en color verde) se hace ridículamente grande para valores muy pequeños de  $n$  ( $n \leq 35$ , por

<sup>1</sup> Google incluye una serie aplicaciones online para este tipo de cálculos que puede accederse desde la dirección url: <https://support.google.com/websearch/answer/3284611?hl=es-US#plotting>.

ejemplo) y se vuelve la mayor de todas, mientras que la **cúbica** supera a la **cuadrática** y todas ellas son mayores que  $f(n) = n * \log(n)$  (en celeste)<sup>2</sup>:

Figura 2: Gráfica general de las funciones  $f(n) = n * \log(n)$  -  $f(n) = n^2$  -  $f(n) = n^3$  -  $f(n) = 2^n$



Lo anterior está mostrando un hecho muy importante de destacar: la *función exponencial* tiene un ritmo o tasa de crecimiento muy violento: para muy pequeños aumentos en el valor de  $n$  (en el dominio de la función), se obtienen valores de respuesta (en su imagen) muy pero muy grandes. Por lo tanto, si se sabe que un algoritmo tiene rendimiento exponencial en cuanto al tiempo de ejecución, entonces ese algoritmo en la práctica es muy poco aplicable: para valores muy pequeños de  $n$ , se obtienen tiempos de respuesta asombrosamente altos y ni siquiera una computadora moderna y potente podría llegar a un resultado en un tiempo aceptable (sin exagerar, incluso una computadora muy potente necesitaría *miles de años* para terminar de ejecutar un programa que incluya un número exponencial de pasos...)

Sobre el final de este curso, veremos algunos elementos de la *Teoría de la Complejidad* dentro de la cual los algoritmos con tiempo de ejecución exponencial tienen una importancia fundamental. Por ahora, baste con saber que si para un problema dado *sólo se conocen algoritmos de tiempo de ejecución exponencial*, entonces esos problemas se

<sup>2</sup> Confesamos que hemos hecho una pequeña trampa: la gráfica de la **función cúbica** que se muestra, corresponde en realidad a la función  $f(n) = 0.1 * n^3$ , mientras que la gráfica de la curva **cuadrática** corresponde a  $f(n) = 0.2 * n^2$ . El motivo fue permitir que la gráfica conjunta de todas las curvas muestre rápidamente la relación de orden para  $n$  grande, que de otro modo exigiría un gráfico mucho mayor.

designan como *problemas intratables* y son objeto de profundos estudios en el campo de las ciencias de la computación<sup>3</sup>.

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [3] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.
- [4] Coursera, "Take the world's best courses, online, for free," 2012. [Online]. Available: <https://www.coursera.org/>. [Accessed 27 March 2013].

---

<sup>3</sup> Los usuarios de computadoras poco experimentados tienden a sobreestimar la capacidad de una computadora moderna para llevar a cabo cualquier proceso en forma veloz. Sin embargo, son muy numerosas las situaciones reales y concretas de problemas que necesitan muchísimo tiempo de procesamiento incluso para una super computadora. Una muy buena película de suspenso y espionaje de 1987 llamada *No Way Out* (o *Sin Salida*) [dirigida por *Roger Donaldson* y protagonizada por *Kevin Costner*] utiliza en forma convincente este hecho: un oficial naval estadounidense debe encontrar al asesino de una mujer, pero una fotografía instantánea arruinada (una "polaroid") que ella tenía de este investigador, incriminaría falsamente al oficial. La fotografía es entregada a un experto en computación que aplica sobre ella un *lentísimo proceso* de reconstrucción de imagen, que *llevará muchas horas*, mientras el oficial busca al asesino. Y debe encontrarlo antes que la computadora reconstruya su propia foto...

# Ficha 21

## Análisis de Algoritmos – Formalización

### 1.] Formalización de la notación Big O.

La notación *Big O* se usa para indicar un *límite superior* para el comportamiento esperado de un algoritmo en cuanto al tiempo de ejecución o el espacio ocupado o algún otro parámetro<sup>1</sup>. Si se dice que un algoritmo de ordenamiento tiene un tiempo de ejecución en el peor caso de  $O(n^2)$ , de alguna forma se está diciendo que ese algoritmo *no se comportará peor* que  $n^2$  en cuanto al tiempo de ejecución. Puede decirse que la función  $f$  que calcula el tiempo de acuerdo al valor de  $n$ , siempre se mantendrá menor o igual que  $n^2$  multiplicada por alguna constante  $c$  [1] [2].

Formalmente, decir que una función  $f$  está en el orden de otra función  $g$ , implica afirmar que eventualmente, para cualquier valor suficientemente grande de  $n$ , la función  $f$  siempre será *menor o igual* que la función  $g$  multiplicada por alguna constante  $c$  mayor a cero. En símbolos:

$$\text{Si } f(n) \text{ es } O(g(n)) \Rightarrow f(n) \leq c * g(n)$$

(para todo valor  $n$  suficientemente grande y algún  $c > 0$ )

Para verlo mejor, analicemos la gráfica de la *Figura 1* (en página 410) [2]. En ella se supone que la función  $f$  es orden de  $g$ . A los efectos del ejemplo, no tiene importancia cuáles sean estrictamente las funciones  $f$  y  $g$ , sino sólo analizar lo que implica la relación  $f(n) = O(g(n))$ .

Si  $f$  es orden  $g$ , entonces podemos encontrar al menos una constante  $c > 0$  (que en la gráfica vale 2) tal que a partir de cierto número  $n_0$  los valores de  $g$  multiplicados por  $c$  serán siempre mayores o iguales a los valores de  $f$ . Eso equivale a decir que la nueva función  $c * g(n)$  será mayor a  $f(n)$  para todo  $n > n_0$ . En el esquema gráfico, puede verse que a partir del valor  $n_0$  la curva  $2 * g(n)$  se vuelve siempre mayor que la curva  $f(n)$ , con lo cual  $f(n) = O(g(n))$ .

Note que la relación  $f(n) = O(g(n))$  implica que  $f$  será menor o igual a  $c * g$  a partir de cierto valor  $n_0$ , y no necesariamente para valores pequeños de  $n$  (o sea, para valores de  $n < n_0$ ). Es decir, lo que importa es lo que pasará para valores grandes o muy grandes de  $n$ , que es lo que se conoce como el *comportamiento asintótico de la función*.

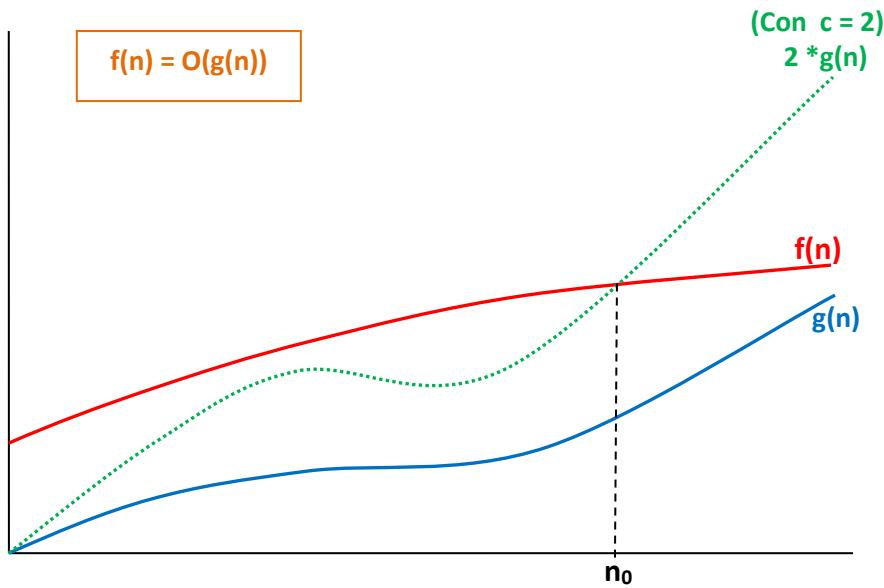
También note que la constante  $c > 0$  puede ser cualquiera y solo basta con encontrar una. Si puede encontrarse al *menos una constante*  $c > 0$  para la cual se pueda probar que  $f(n) \leq c$

<sup>1</sup> Parte del desarrollo de esta sección se basa en: Weiss, M. A. (2000). "Estructuras de Datos en Java". Madrid: Addison Wesley. ISBN: 84-7829-035-4 (página 103 y siguientes). También se han seguido ideas (sobre todo en el planteo de las gráficas) del material del curso "Design and Analysis of Algorithms I" – Stanford University (a cargo de Tim Roughgarden, Associate Professor): <https://www.coursera.org/courses>.

$*g(n)$  desde cierto punto  $n_0$  en adelante, entonces  $f$  es orden de  $g$ . En la gráfica anterior, también hubiese sido válido suponer  $c = 3$  o  $c = 4$ , o cualquier valor de  $c > 0$  que cumpla la relación.

Si se observa con atención, lo que se está haciendo es intentar determinar la forma de variación de la función  $f$  de acuerdo a la forma en que varía otra función  $g$  ya conocida. Pero como en rigor no se usa necesariamente la propia  $g$ , sino alguna función múltiplo de  $g$  (porque se multiplica a  $g$  por la constante  $c$ ), entonces la expresión  $O(g(n))$  en realidad está implicando *una familia de funciones* cuyo comportamiento es caracterizado por el comportamiento de  $g$ .

Figura 1: Idea general del significado de  $f(n) = O(g(n))$ .



En ese sentido, la expresión  $O(g(n))$  es lo que se conoce como un *orden de complejidad*: un conjunto o familia de funciones que se comportan asintóticamente de la misma forma. La función más característica de ese conjunto (la más simple, sin constantes ni términos independientes adicionales) es la que normalmente se usa para expresar la relación de orden, y se suele designar como *función representante* del conjunto (o *función característica* o también *función dominante*). Así, cuando decimos que la *búsqueda secuencial* tiene un tiempo de ejecución  $t(n) = O(n)$ , estamos diciendo que el tiempo  $t$  tiene la misma forma de variación asintótica que el conjunto de funciones representadas por  $g(n) = n$  (que es en este caso es la *función característica* de ese orden de complejidad). Como todas en  $O(n)$  se comportan de la misma forma que  $g(n) = n$ , no tiene relevancia entonces incluir constantes y términos independientes en la expresión de orden. No es necesario que el analista agregue detalles: decir (por ejemplo)  $t(n) = O(2n + 5)$  es asintóticamente lo mismo que  $t(n) = O(n)$ .

Y otro detalle a considerar: al expresar una relación de orden en notación *Big O*, se está indicando una *cota superior* para el comportamiento de una función  $f$  en términos del comportamiento de la familia de funciones  $O(g)$ , pero en general el analista buscará la *menor cota superior para  $f$* . Está claro que si el tiempo de ejecución  $t$  de la búsqueda secuencial es  $t(n) = O(n)$ , es también cierto que  $t(n) = O(n^2)$  (ya que como vimos  $O(n) < O(n^2)$ ) y si se prosigue con ese argumento, resulta que prácticamente todos los algoritmos

conocidos son  $O(2^n)$  (ya que la función exponencial o algún múltiplo de ella siempre será mayor que las demás para  $n$  grande). Pero al expresar una función  $f$  en notación Big O, lo que se quiere obtener es la familia de funciones que sirva como *mejor cota superior* para el comportamiento de  $f$ , y no cualquier familia que sea mayor o igual que  $f$  (ya que en este caso el análisis sería demasiado amplio y vago).

Todo lo anterior ayuda en el análisis de algoritmos: muchas veces un programador tiene un conocimiento estimado (o incluso intuitivo) del comportamiento de la función  $f$  que predice (por ejemplo) el tiempo de ejecución de un algoritmo en el peor caso, pero desconoce la forma analítica precisa de esa función (o a los efectos del análisis del peor caso no requiere de esa forma precisa)<sup>2</sup>. Si el programador puede probar que su función desconocida  $f$  es del orden de otra función conocida (y posiblemente más simple)  $g$ , entonces tendrá una *cota superior* para su función  $f$ : Sabrá que su algoritmo *nunca será peor* que  $g$  multiplicada por una constante dada. Esto puede parecer muy vago, pero en el análisis asintótico tiene mucha importancia: al fin y al cabo, saber que nuestro algoritmo nunca será peor que  $n^{\log(n)}$  (por ejemplo) nos garantiza que ese algoritmo será subcuadrático incluso en el peor caso, aún si ignoramos los coeficientes precisos de la *verdadera* función  $f$ .

A partir de estos elementos, surgen algunas relaciones de orden básicas que debemos tener presentes, y que de alguna manera ayudan a justificar que en notación Big O se puede prescindir de las constantes y quedarse solo con el término o función dominante. Por otra parte, estas relaciones ayudan a poder estimar en forma rápida el comportamiento asintótico de un algoritmo sin entrar permanentemente en la necesidad de demostrar la relación de orden. No es necesario que estudie ni recuerde las demostraciones, pero viene bien tenerlas a mano [2]:

---

a.) Cualquier función polinómica en  $n$  de grado  $k$ , es orden  $n^k$ . Simbólicamente:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + a_2 n^2 + a_1 n + a_0 = O(n^k)$$

Demostración: si tomamos  $c = |a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_2| + |a_1| + |a_0|$  y  $n_0 = 1$ , entonces debemos probar que para todo  $n > n_0 = 1$ , se cumple que  $f(n) \leq c * n^k$ . O sea:

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + a_2 n^2 + a_1 n + a_0 \\ \Rightarrow f(n) &\leq |a_k| * n^k + |a_{k-1}| * n^{k-1} + |a_{k-2}| * n^{k-2} + \dots + |a_2| * n^2 + |a_1| * n + |a_0| \\ \Rightarrow f(n) &\leq |a_k| * n^k + |a_{k-1}| * n^k + |a_{k-2}| * n^k + \dots + |a_2| * n^k + |a_1| * n^k + |a_0| * n^k \\ \Rightarrow f(n) &\leq (|a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_2| + |a_1| + |a_0|) * n^k \end{aligned}$$

$$\Rightarrow f(n) \leq c * n^k \quad \text{que es lo que se quería probar.}$$

Este resultado muestra que si sabemos que nuestra función  $f$  en  $n$  (desconocida) es polinómica de grado  $k$ , entonces *podemos prescindir de todas las constantes y de todos los términos que no sean de grado  $k$* , y quedarnos sólo con el término dominante  $n^k$ , simplificando el análisis.

---

<sup>2</sup> En todo caso, buscar la forma analítica de una función que permita predecir el tiempo de ejecución de un algoritmo no es lo mismo que directamente predecir el futuro, como podía hacer el personaje de la película *Next* (del año 2007), dirigida por *Lee Tamahori* y protagonizada por *Nicolas Cage* y *Jessica Biel*. Se trata de la historia de Cris Johnson, un mago de Las Vegas que puede ver lo que ocurrirá en el futuro inmediato, pero sólo por un lapso de dos minutos en el futuro. Por causa de esa capacidad, el FBI lo busca y pretende detenerlo para que ayude a predecir y detener ataques terroristas, convirtiendo así la vida del mago en un martirio permanente.

---

b.) Sea  $f(n) = 2^{n+10}$  y  $g(n) = 2^n$ . Entonces  $2^{n+10} = O(2^n)$

Demostración: Debemos tomar dos constantes  $c$  y  $n_0$  para las que se cumpla que  $2^{n+10} \leq c * 2^n$ . Sabemos que:

$$2^{n+10} = 2^n * 2^{10} = 2^n * 1024$$

Por lo que si tomamos  $c = 1024$  sabemos que para  $n_0 = 1$ , la relación postulada se cumple.

Procediendo en forma similar, se puede probar que  $2^{n+k} = O(2^n)$  para cualquier  $k \geq 1$ , lo cual también simplifica el análisis.

---

c.) En notación *Big O*, la base de los logaritmos puede obviarse, ya que la función  $\log_b(n)$  es  $O(\log_2(n))$  para cualquier base  $b > 1$ . Simbólicamente:

Sea  $f(n) = \log_b(n)$  y  $g(n) = \log_2(n) = \log(n)$ . Entonces  $\log_b(n) = O(\log(n))$  [con  $b > 1$ ]

Demostración: Por defecto, asuma que la base 2 es implícita (o sea: asuma que  $\log_2(n) = \log(n)$ ). Intentamos entonces probar que  $\log_b(n) \leq c * \log(n)$  para alguna constante cualquiera  $c > 0$ .

Sea  $\log_b(n) = k$ . Entonces (por definición de logaritmo):  $b^k = n$ . Tomemos la constante  $p = \log(b)$ . Entonces  $2^p = b$ .

Entonces, de los dos hechos anteriores resulta:

$$\begin{aligned} n &= b^k = (2^p)^k \\ n &= 2^{pk} \\ \log(n) &= p * k \\ \log(n) &= p * \log_b(n) \\ \log_b(n) &= 1/p * \log(n) \end{aligned}$$

Por lo tanto, si tomamos  $c \geq 1/p$ , entonces  $\log_b(n) \leq c * \log(n)$  lo que prueba la relación 3. Hemos visto en fichas anteriores que este hecho resulta sumamente práctico cuando la relación de orden analizada incluye logaritmos: otra vez, se simplifica el análisis.

## 2.] Algunas consideraciones prácticas.

De ahora en adelante, se espera que el alumno sea capaz de al menos "intuir" una relación de orden para un algoritmo en el peor caso (es decir, dado un algoritmo, ser capaz de dar en notación *Big O* un orden para el peor caso de ese algoritmo en tiempo o espacio). Para ganar destreza en esa tarea, van algunos consejos prácticos:

- i. Dado el algoritmo que debe analizar (expresado en forma de diagrama de flujo, o de pseudocódigo, o de programa fuente o incluso expresado en forma coloquial o intuitiva), tenga en claro *cuál será el factor de eficiencia* que quiere analizar: tiempo de ejecución o memoria empleada, por ejemplo).
- ii. En ese algoritmo determine **con exactitud el tamaño del problema** (o volumen de datos a procesar). En muchos casos esto es simple de hacer (el tamaño  $n$  de un arreglo si se quiere ordenar ese arreglo o buscar en él) pero en algunos casos no es tan simple ni tan obvio (aunque no enfrentaremos casos extremos por ahora...) En algunos casos, el tamaño puede venir expresado con dos o más variables (por caso, si se pide procesar dos arreglos de

tamaños  $m$  y  $n$  respectivamente), y la expresión de orden final podría estar a su vez basada en dos o más variables (no es extraño encontrar expresiones de la forma  $t(v, w) = O(v * \log(w))$  siendo  $v$  y  $w$  dos variables, o casos como  $t(n, m) = O(m * n^2)$ , por ejemplo).

- iii. En el algoritmo analizado, determine claramente cuál es la *instrucción crítica* que ese algoritmo lleva a cabo. La **instrucción crítica es la instrucción o bloque de instrucciones que se ejecuta más veces a lo largo de toda la corrida**, y si está analizando el tiempo de ejecución del algoritmo, entonces la acumulación de los tiempos de ejecución de la operación crítica es lo que lleva al algoritmo a su tiempo final. En un ordenamiento, por lo general, la instrucción que más se ejecuta es la comparación (y lo mismo en una búsqueda) por lo que esa será la instrucción crítica.
- iv. En este momento, tenga en cuenta que no es lo mismo realizar un conteo exhaustivo y riguroso de operaciones críticas, que hacer un **análisis asintótico**. Si lo que necesita es un **conteo exhaustivo**, entonces deberá extremar el análisis y tratar de expresar una fórmula con **toda rigurosidad y detalle de constantes, términos no dominantes y términos independientes que indique cuántas operaciones críticas se ejecutan exactamente para un valor dado de  $n$** . Pero si lo que necesita es un **análisis asintótico**, entonces siga leyendo los puntos que vienen a continuación...
- v. Para el **análisis asintótico**, si ya tiene una expresión de **conteo exhaustiva** identifique el **término dominante** en ella, y límítese a ese término. No diga que un algoritmo es  $O(n^3 + n^2)$ : por la *relación a.)* vista en *página 411*, en este caso bastará con decir  $O(n^3)$ . Entienda que para  $n$  suficientemente grande, el término  $n^2$  será técnicamente despreciable frente a  $n^3$ . Si no tiene una expresión o fórmula exhaustiva, identifique en forma general la estructura del algoritmo que contiene a la operación crítica y deduzca en forma intuitiva. Valen los siguientes consejos para situaciones comunes, independientemente de otras situaciones que deberá resolver con otros criterios:
  - Si la operación crítica es (por ejemplo) una comparación y está contenida dentro de dos ciclos anidados de  $n$  iteraciones cada uno, entonces su algoritmo es  $O(n^2)$ .
  - Una operación crítica de tiempo de ejecución constante ( $O(1)$ ) incluida dentro de  $k$  ciclos anidados de  $n$  iteraciones cada uno, tendrá tiempo de ejecución  $O(n^k)$ .
  - Si su algoritmo consta de varios bloques de instrucciones independientes entre sí, entonces por la misma *relación a.)* de página 411 su algoritmo tendrá un tiempo de ejecución en el orden del que corresponda al bloque con mayor tiempo. Así, si su algoritmo tiene un primer bloque que ejecuta en tiempo  $O(n)$  y luego dos bloques más que ejecutan en tiempos  $O(n^2)$  y  $O(\log(n))$ , entonces el tiempo completo sería  $t(n) = O(n) + O(n^2) + O(\log(n))$  pero esto es asintóticamente igual a  $O(n^2)$ , por lo que  $t(n) = O(n^2)$ .
  - Si todo su algoritmo está compuesto sólo por un bloque de instrucciones de tiempo constante (asignaciones o condiciones, por ejemplo) sin ciclos ni procesos ocultos dentro de una función, entonces todo el algoritmo ejecuta en tiempo constante  $t(n) = O(1)$  (ya que sería  $t(n) = O(1) + O(1) + \dots + O(1)$  que es lo mismo que  $t(n) = O(1)$  (por la misma relación a.) de *página 411*).
  - Si su algoritmo se basa en tomar un bloque de  $n$  datos, dividirlo por 2, aplicar una operación de tiempo constante y luego procesar sólo una de las mitades de forma de volver a dividirla y continuar así hasta no poder hacer otra división, entonces su algoritmo ejecuta en tiempo  $t(n) = O(\log_2(n))$  que por la *relación c. de página 412*, es lo mismo que  $t(n) = O(\log(n))$  sin importar la base del logaritmo.
- vi. No incluya *constantes* dentro de la expresión de orden, salvo aquellas que indiquen el exponente específico del término dominante. En general no dirá  $O(2n)$  sino simplemente

$O(n)$ . La notación O le permite rescatar de un solo golpe de vista la forma de variación del término dominante, y en esa variación son en general despreciables las constantes.

- vii. No se preocupe por la base del logaritmo si su expresión de orden incluye logaritmos. La relación 3 prueba que la base no es relevante en análisis asintótico. Y en todo caso, la base del logaritmo es ella misma una constante.

En el *punto iv* de estas consideraciones prácticas, hemos indicado que no es lo mismo un **conteo exhaustivo** que un análisis de **comportamiento asintótico**. El primero es mucho más riguroso. El segundo es más amplio. Tomemos por ejemplo el ya conocido *Ordenamiento de Selección Directa* para un arreglo  $v$  de  $n$  componentes, que en general se plantea así:

```
n = len(v)
for i in range(n-1):
    for j in range(i+1, n):
        if v[i] > v[j]:
            v[i], v[j] = v[j], v[i]
```

Si queremos hacer un **conteo exhaustivo** del número de operaciones críticas (en este caso, las *comparaciones*) que este algoritmo ejecuta, debemos ver que el proceso consiste en tomar la primera casilla del arreglo (designada como *pivot*), comparar su contenido con los  $(n-1)$  casilleros restantes y dejar el menor valor en la casilla *pivot*. Luego se toma la segunda casilla como *pivot*, se compara contra las  $(n-2)$  restantes, y se vuelve a dejar el menor de lo que quedaba del arreglo en la casilla *pivot*. Así, se hacen  $(n-1)$  pasadas, y cuando el *pivot* sea la casilla  $(n-2)$  se hará una única y última comparación más contra la casilla  $(n-1)$  y el arreglo quedará ordenado. Pero esto significa que la *cantidad de comparaciones* a realizar sale de:

Pasada 1:	$n-1$ comparaciones
Pasada 2:	$n-2$ comparaciones
Pasada 3:	$n-3$ comparaciones
...	...
Pasada $n-2$ :	2 comparaciones
Pasada $n-1$ :	1 comparación

Lo anterior implica que el total de comparaciones  $t(n)$  a ejecutar para el total  $n$  de casilleros en el arreglo, será igual a la suma:

$$t(n) = 1 + 2 + \dots + (n-3) + (n-2) + (n-1)$$

que se puede demostrar que es igual a:

$$t(n) = (n-1) * n / 2$$

y entonces:

$$t(n) = (n^2 - n) / 2$$

que tiene forma claramente cuadrática... Esto quiere decir que el algoritmo de *Selección Directa* hará una cantidad de comparaciones que será *exactamente* función de  $n$  al cuadrado, y por lo tanto el tiempo  $t$  demorado por ese algoritmo en ordenar el arreglo será *proporcional a n al cuadrado*. Hasta aquí, hemos desarrollado un **conteo exhaustivo** de operaciones críticas y la función obtenida expresa con todo detalle ese conteo. Si ahora se nos pide un análisis de **comportamiento asintótico**, el camino está allanado porque ya tenemos la fórmula precisa del **conteo exhaustivo**. El quinto punto de la lista de recomendaciones que hemos mostrado antes, nos lleva directamente a la expresión de orden: el tiempo de ejecución de este algoritmo, en notación Big O, es  $t(n) = O(n^2)$ ,

prescindiendo de constantes y términos no dominantes. Pero aún sin tener la fórmula rigurosa del *conteo exhaustivo*, podríamos haber llegado a la misma conclusión simplemente analizando el código fuente (como también se indica en la quinta recomendación práctica): dos ciclos anidados de aproximadamente  $n$  repeticiones cada uno, y una instrucción condicional contenida en el ciclo más interno:  $O(n^2)$ .

Un análisis similar (tanto *exhaustivo* como *asintótico*) permite deducir que los otros algoritmos de *ordenamiento simples* o *directos* (como la ordenación por *Intercambio Directo* y la ordenación por *Inserción Directa* que ya hemos visto en la *Ficha 17*) también tienen un *comportamiento cuadrático* en el *peor caso*. El hecho es que el algoritmo de *Selección Directa* siempre hará la cantidad de comparaciones calculada más arriba, pero los otros dos podrían realizar una cantidad algo menor en *casos más favorables*. En otras palabras, *si sólo se consideran comparaciones* (y no por ejemplo la cantidad de veces que se hacen efectivamente intercambios), el método de *Selección* siempre cae en su peor caso, pero los otros dos podrían tener casos muy favorables dependiendo del estado inicial del arreglo. Los tres son de comportamiento cuadrático, y *asintóticamente pertenecen al mismo orden de complejidad*, pero las constantes que describen en forma analítica la función de *conteo exhaustivo* pueden ser diferentes.

En la siguiente tabla se muestran los tiempos de ejecución en notación *Big O* (comportamiento asintótico) para el peor caso de los algoritmos de ordenamiento vistos (o que veremos más adelante), más algunas consideraciones respecto de casos promedio o incluso mejores casos si esas situaciones son relevantes [3]:

**Tabla 1: Comportamiento asintótico de los principales algoritmos de ordenamiento.**

Algoritmo	Tiempo	Observaciones
Burbuja ( <i>Intercambio Directo</i> )	$O(n^2)$ (peor caso)	Mejor caso: $O(n)$ si el arreglo está ya ordenado.
Selección Directa	$O(n^2)$ (peor caso)	Siempre...
Inserción Directa	$O(n^2)$ (peor caso)	
Quicksort	$O(n * \log(n))$ (caso medio)	Peor caso (depende de la implementación): $O(n^2)$ .
Heapsort	$O(n * \log(n))$	Caso medio: también $O(n * \log(n))$ .
Shellsort	$O(n^{1.5})$	Para la serie de incrementos mostrada en clase.

Los siguientes son problemas y/o algoritmos muy comunes para los que hacemos un breve análisis asintótico del *peor caso* en notación *Big O*. Intente rápidamente (en lo posible, sin mirar el análisis que hacemos para cada uno) hacer su propia estimación, a modo de ejercicio [4]:

- *Búsqueda secuencial de un valor  $x$  en un arreglo desordenado de  $n$  componentes*: La operación crítica es la comparación, y en el peor caso debe hacerse  $n$  veces (si  $x$  no está en el arreglo o está muy al final). Entonces para el peor caso resulta  $t(n) = O(n)$ .
- *Búsqueda secuencial de un valor  $x$  en un arreglo ordenado de  $n$  componentes*: La operación crítica es la comparación, y en el peor caso (otra vez) debe hacerse  $n$  veces (si  $x$  no está en el arreglo y es mayor a todos los elementos del mismo). Entonces para el peor caso también resulta  $t(n) = O(n)$ .
- *Búsqueda binaria de un valor  $x$  en un arreglo de  $n$  elementos (por supuesto, ordenado)*: La operación crítica es la comparación, y en el peor caso debe hacerse  $\log_2(n)$  veces (si  $x$  no está en el arreglo), ya que en ese caso se hacen  $\log_2(n)$  divisiones por 2, y una comparación por cada división. Entonces para el peor caso  $t(n) = O(\log(n))$ .

- *Conteo de la cantidad de valores de un arreglo de tamaño  $n$  que son mayores al promedio de todos los valores del arreglo:* Hay un primer proceso en el cual deben hacerse  $n$  acumulaciones para calcular el promedio (hasta aquí,  $O(n)$ ). Y luego un segundo proceso en el cual se hace  $n$  comparaciones para determinar cuántos valores son mayores que el promedio (con lo que se tiene otra vez  $O(n)$ ). El tiempo total será  $t(n) = O(n) + O(n) = O(2n) = O(n)$  con lo que  $t(n) = O(n)$ .
- *Multiplicación de matrices (suponga, para simplificar, que las matrices son cuadradas y del mismo orden  $n$ ):* Sin entrar en demasiados detalles, el algoritmo clásico emplea tres ciclos for de  $n$  iteraciones cada uno, y en cada giro realiza la acumulación de un producto. Por lo tanto, resulta  $t(n) = O(n^3)$ .
- *Conteo de las frecuencias de aparición de los  $n$  números de un conjunto, que pueden venir repetidos, sin conocer el rango en el que vienen esos números y usando un arreglo de objetos de conteo:* Cada uno de los  $n$  números debe buscarse secuencialmente en un arreglo que en el peor caso también llegará a tener  $n$  casilleros si todos los números fuesen diferentes. Como una búsqueda secuencia ejecuta en tiempo  $t(n) = O(n)$ , y debemos hacer  $n$  búsquedas, tenemos un tiempo final  $t(n) = n * O(n)$  que es lo mismo que  $t(n) = O(n * n) = O(n^2)$ .
- *Conteo de las frecuencias de aparición de los  $n$  números de un conjunto, que pueden venir repetidos, conociendo el rango en el que vienen esos números y usando un arreglo de conteo directo:* Cada uno de los  $n$  números debe contarse en un arreglo de acceso directo (cada número se cuenta en la casilla cuyo índice coincide con el mismo número). Como un conteo de ese tipo ejecuta en tiempo constante  $t(n) = O(1)$ , y debemos hacer  $n$  conteos, tenemos un tiempo final  $t(n) = n * O(1)$  que es lo mismo que  $t(n) = O(n)$ .
- *Fusión de dos arreglos ordenados (tamaños  $m$  y  $n$ ) en un tercer arreglo ordenado:* En este caso, hay dos arreglos de entrada con tamaños  $m$  y  $n$  y hay que procesar todo el conjunto, por lo cual el tamaño del problema es  $m + n$ . El algoritmo esencial para hacer esta fusión, genera un arreglo de tamaño  $m + n$  ordenado. Se recorren los dos arreglos originales, cada uno con su índice, se comparan dos elementos (uno de cada vector) y el menor se lleva al arreglo de salida. Como se hace una comparación por cada uno de los  $m + n$  casilleros del arreglo de salida, el tiempo total resultante es  $t(n) = O(m + n)$ . Note que nada impide que la expresión de orden esté basada en dos o más variables.
- *Inserción y eliminación de un elemento en una pila o en una cola:* Sin importar cuántos elementos tenga la pila o la cola, la inserción y la eliminación proceden en tiempo constante (en nuestro caso, ambas estructuras han sido implementadas sobre un arreglo en Python, y en ambos casos, insertar o eliminar el elemento del frente o del fondo se hace en tiempo constante). Por lo tanto, para todas las operaciones pedidas, el tiempo de ejecución resulta ser  $t(n) = O(1)$ .

### 3.] Otras notaciones típicas del análisis de algoritmos.

En ocasiones, se desea poder indicar no ya un *límite o cota superior*, sino (por ejemplo) un *límite o cota inferior* para una función  $f$  dada. Para ese y otros casos, existen otras notaciones de orden (que solo citaremos a modo documentativo, aunque no usaremos frecuentemente) [1] [2].

Si se quiere expresar que el tiempo de ejecución o el espacio de memoria ocupado por un algoritmo *no se comportará mejor* que cierta función  $g$ , se usa la notación *Omega ( $\Omega$ )*. Si estamos analizando el tiempo de ejecución de un algoritmo, entonces al decir que el tiempo de ese algoritmo *no se comportará mejor* que cierta función  $g$ , queremos decir que los

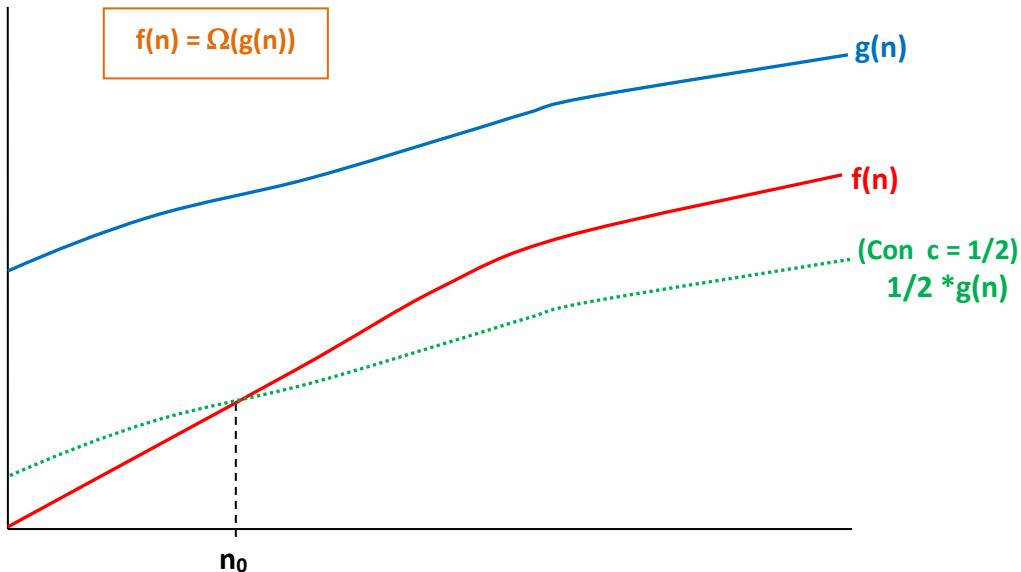
valores de tiempo de ejecución estarán siempre *por arriba* (o a lo sumo serán iguales) de los calculados para esa función límite. Así, por ejemplo, puede verse que si se considera la cantidad de comparaciones, el algoritmo de *Selección Directa* no sólo es  $O(n^2)$ , sino también  $\Omega(n^2)$  (léase: *Omega n cuadrado*): simplemente, no lo hará ni mejor ni peor que algún múltiplos o submúltiplo de  $n^2$ .

Formalmente, la función  $f$  que mide el tiempo o el espacio (o cualquier otro factor) de un algoritmo de acuerdo a los valores de  $n$ , es  $\Omega(g(n))$  si se ajusta a la siguiente descripción:

$$\text{si } f(n) \text{ es } \Omega(g(n)) \Rightarrow f(n) \geq c * g(n) \\ (\text{para } n \text{ suficientemente grande y algún } c > 0)$$

Gráficamente, si  $f$  es *omega*  $g$ , entonces podremos encontrar al menos una constante  $c$  mayor a cero para la cual los valores  $f(n)$  serán siempre mayores o iguales a  $c * g(n)$ , a partir de cierto número  $n_0$ :

**Figura 2: Idea general del significado de  $f(n) = \Omega(g(n))$ .**

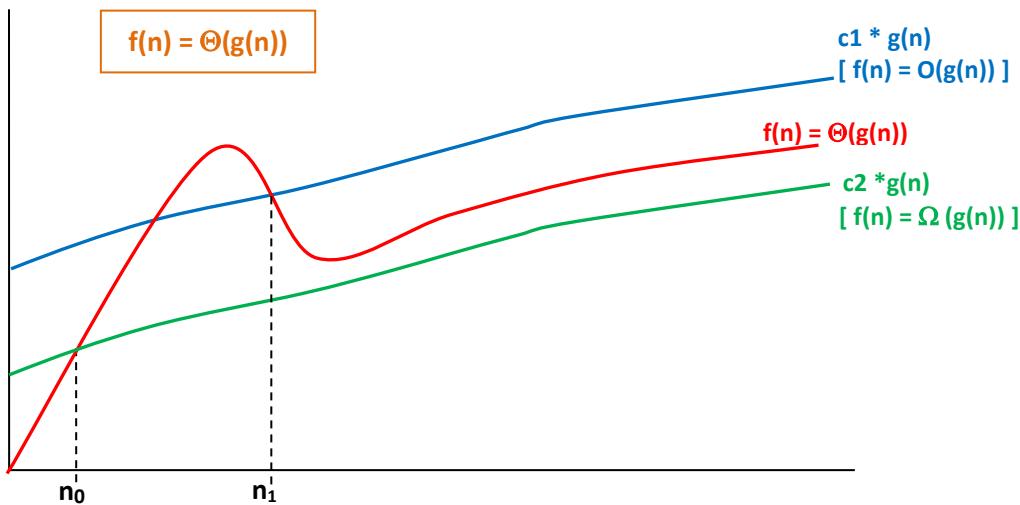


Por otra parte, si se quiere indicar que una función  $f$  se comporta tanto por arriba como abajo de la misma forma que múltiplos de otra función  $g$ , se puede usar también la notación *Theta* ( $\Theta$ ): si  $f(n)$  es  $\Theta(g(n))$ , significa que  $f$  puede acotarse tanto superior como inferiormente por múltiplos de  $g$ . En otras palabras:

$$\text{Si } f(n) \text{ es } \Theta(g(n)) \Rightarrow f(n) \text{ es } O(g(n)) \text{ y } f(n) \text{ es } \Omega(g(n)) \\ (\text{para } n \text{ suficientemente grande})$$

La notación *Theta* es muy precisa: no solo nos proporciona una cota superior para el análisis asintótico de una función, sino también *que también nos garantiza que esa cota superior es la mejor posible*. Note que este es el caso del algoritmo de Selección Directa respecto a  $g(n) = n^2$  al considerar comparaciones: el algoritmo no será mejor que una función cuadrática y una función cuadrática es la mejor aproximación que puede darse tanto superior como inferior. Se dice entonces que ese algoritmo es  $\Theta(n^2)$  (y se lee: *Theta n cuadrado*).

Gráficamente, si  $f$  es *Theta*  $g$ , entonces la gráfica de  $f$  podrá "encerrarse" entre dos curvas proporcionales a  $g$ , para dos constantes  $c_1$  y  $c_2$  diferentes:

Figura 3: Idea general del significado de  $f(n) = \Theta(g(n))$ .

Finalmente, otra notación que también suele usarse, es la notación  $o()$  (léase: *o minúscula o little o*). Esta notación es similar a  $O()$ , pero considerando *sólo relación de menor estricto* (sin el signo igual). Es útil cuando se quiere indicar una función *estRICTAMENTE SUPERIOR* para el comportamiento de un algoritmo. Así, si podemos decir que un algoritmo dado es (por ejemplo)  $O(n^{1.5})$  en cuanto a cierta cantidad de operaciones críticas, entonces se puede indicar también a ese algoritmo como  $o(n^2)$ : es una forma elegante (aunque menos precisa) de decir que el algoritmo analizado es *subcuadrático*... En símbolos:

$$\text{Si } f(n) \text{ es } o(g(n)) \Rightarrow f(n) < c \cdot g(n) \quad (\text{para } n \text{ suficientemente grande y algún } c > 0)$$

Con el siguiente cuadro mostramos un resumen de las cuatro notaciones (tomado y adaptado del libro *Estructuras de Datos en Java*, de *Mark Allen Weiss* – página 116):

Tabla 2: Resumen de notaciones típicas del análisis de algoritmos.

Expresión	Significado
$f(n)$ es $O(g(n))$	El crecimiento de $f(n)$ es $\leq$ que el crecimiento de $g(n)$
$f(n)$ es $\Omega(g(n))$	El crecimiento de $f(n)$ es $\geq$ que el crecimiento de $g(n)$
$f(n)$ es $\Theta(g(n))$	El crecimiento de $f(n)$ es $=$ que el crecimiento de $g(n)$
$f(n)$ es $o(g(n))$	El crecimiento de $f(n)$ es $<$ que el crecimiento de $g(n)$

## Bibliografía

- [1] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.
- [2] Coursera, "Take the world's best courses, online, for free.", 2012. [Online]. Available: <https://www.coursera.org/>. [Accessed 27 March 2013].
- [3] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

# Ficha 22

## Archivos

### 1.] Introducción.

En todos los programas que se desarrollaron hasta aquí existía un problema común: cada vez que el programa se ejecutaba, se debía volver a cargar todos sus datos y el programa volvía a calcular todos los resultados. Pero al terminar la ejecución de un programa, ni los datos cargados ni los resultados calculados quedaban *resguardados* de forma que pudieran volver a usarse si fuera necesario en otra corrida posterior. Todas las estructuras de datos que hasta aquí se estudiaron (arreglos de cualquier dimensión, registros, arreglos de registros, pilas, colas, etc.) tenían ese inconveniente, y debido a ello se dice que son *estructuras de datos de naturaleza volátil*.

Es obvio que esa situación es poco útil: ya se sabe lo tedioso e impráctico que resulta cargar  $n$  registros por teclado en un arreglo cada vez que un programa arranca, o cargar todos los valores de nuevo en un arreglo cada vez que se ejecuta un programa para verificar un algoritmo de ordenamiento o de búsqueda. Además, muchas veces un programa genera como resultado un arreglo de varios componentes, pero cuando el programa termina ese arreglo se pierde, y no queda disponible para que pueda ser usado en otros programas... Y estamos nombrando situaciones de programación que sólo se dan en las aulas: a nivel profesional es absolutamente inútil desarrollar un sistema complejo basado sólo en estructuras volátiles. Es fácil comprender que el sistema de información de un banco *debe* tener almacenados en *forma permanente* (y no volátil) los datos de todos sus clientes y movimientos de cuentas: sería inaceptable que los empleados del centro de cómputos del banco deban volver a cargar todos los datos para que el banco pueda operar cada vez que abre sus puertas por la mañana, (y aún cuando así lo hicieran, no les bastaría un solo día para cargar todos los datos: la cantidad de clientes y movimientos de cuentas que tiene un banco es enorme)<sup>1</sup>.

Para dar una solución a este inconveniente existen los *archivos*, que son conjuntos de datos *persistentes* (es decir, no volátiles): un *archivo* es una estructura de datos que en lugar de almacenarse en memoria principal, se almacena en dispositivos de almacenamiento

<sup>1</sup> Olvidar absolutamente todo suele tener consecuencias desastrosas, y el cine ha explorado en películas muy conocidas esa situación. En 2013, la película *Oblivion* (conocida en español como "*Oblivion: El Tiempo del Olvido*"), dirigida por *Joseph Kosinski* y protagonizada por *Tom Cruise*, gira alrededor de un planeta Tierra devastado por la guerra contra una civilización extraterrestre. Los invasores fueron derrotados pero la Tierra es inhabitable por la radiación, por lo que los sobrevivientes están abandonando en forma sistemática el planeta. Distintos grupos de trabajo de alta tecnología se encargan de vigilar los trabajos de transporte de toda el agua del mar hacia una nave gigante en el espacio, para llevarla consigo cuando se produzca la migración final. Pero uno de los vigilantes sospecha que algo anda mal... y finalmente descubre que su memoria ha sido borrada y reconfigurada, entre otros detalles...

externos (como discos o memorias flash) y por lo tanto el contenido de un *archivo* no se pierde al finalizar el programa que lo utiliza [1].

El uso de *archivos* permite el manejo de grandes volúmenes de datos sin tener que volverlos a cargar desde teclado cada vez que se deseé procesarlos. Por otra parte, una vez que un *archivo* fue creado y grabado en un dispositivo externo, ese *archivo* puede ser utilizado por cualquier otro programa, ya sea para obtener datos del archivo y/o para modificar los datos del mismo (siempre y cuando la estructura interna de ese archivo sea conocida por quien desarrolla esos otros programas).

Al igual que ocurre con la representación de cualquier tipo de valor en la memoria de un computador, los datos que se graban en un archivo se representan en sistema binario y por cada dato se utilizan tantos bytes como sea necesario para representar ese dato. De allí que, entonces, el *tamaño* de un *archivo* es la *cantidad total de bytes* que contiene el archivo.

Si bien, como acabamos de indicar, en **todos** los archivos se representa su contenido en base al sistema binario, en la práctica es común hacer una distinción entre los llamados *archivos de texto* y los llamados *archivos binarios* [2]:

- **Archivos de texto:** Todos los bytes del archivo son *interpretados* como *caracteres* (en base, por ejemplo, a la tabla ASCII). Si un archivo es de texto también contiene bytes, *pero se asume que todos esos bytes representan caracteres que pueden ser visualizados en pantalla*, con la única excepción del carácter de salto de línea (que solemos representar en Python como "\n"). En la tabla ASCII, por caso, los primeros 32 caracteres (numerados entre el 0 y el 31) no tienen representación visual y se designan como *caracteres de control* (de hecho, el carácter número 13 es el que corresponde al *retorno de carro* (que en Windows implica también un *salto de línea*) y lo denotamos como "\n"). Pero desde el carácter 32 (que es el espacio en blanco) en adelante, todos tienen representación visual (por ejemplo, el carácter 65 es la letra "A"). Pues bien: si un archivo es "de texto", debería contener solamente bytes cuyos valores numéricos sean mayores o iguales a 32, con la sola excepción del byte que representa un salto de línea<sup>2</sup>. Son archivos de texto, por ejemplo, los archivos fuente en Python de cada programa que hemos desarrollado a lo largo del curso, así como los archivos fuente de cualquier otro lenguaje. Estos archivos no sólo pueden ser abiertos y editados desde un IDE para el respectivo lenguaje, sino que pueden ser abiertos y editados con *cualquier editor de textos* que conozca.
- **Archivos binarios:** Las secuencias o bloques de bytes que el archivo contiene *representan información de cualquier tipo* (números en formato binario, caracteres, valores booleanos, etc.) y *no se asume* que cada byte representa un carácter. En sí mismo, el archivo no distingue si los bytes grabados pertenecen a un número entero o a un número flotante o a una cadena de caracteres: es el *programador* el que determina cómo interpretar el contenido. Un archivo binario puede entonces contener bytes que en caso de ser interpretados como caracteres llevarían a elementos de la tabla de conversión (ASCII u otra) que no tengan representación visual, haciendo que en su lugar el programa muestre caracteres de reemplazo, o bien haría que todos los bytes sean mostrados como caracteres

<sup>2</sup> En algunos archivos de texto se admite la inclusión adicional de caracteres de control e información a modo de marcas para darle formato al texto (por ejemplo, incorporando distintos tipos de letras (como *Arial* o *Courier*) y estilos (como *italic* o *bold*)). En ese sentido, los archivos de texto que sólo contienen bytes con valores desde el 32 en adelante y ocasionalmente algún salto de línea, *sin inclusión de información de formato*, se suelen designar también como *archivos de texto plano* para indicar que se está hablando de los archivos de texto más simples posibles.

provocando una secuencia ininteligible de caracteres en la pantalla. Archivos binarios típicos son, por ejemplo, los archivos ejecutables .exe de la plataforma Windows, o cualquier archivo que contenga una imagen, o (en general) *cualquier archivo que no sea específicamente de texto*.

Quede claro: **todos** los archivos contienen información representada en binario, y por lo tanto y en ese sentido, *todos los archivos son binarios*. Pero en la práctica, se habla de *archivos de texto* y *archivos binarios* para distinguir a aquellos en los que sólo se espera encontrar bytes que representen caracteres visualizables (los *archivos de texto*) de aquellos en los que no se asume nada respecto del significado previo de cada byte y la forma de interpretarlos (los *archivos binarios*) [1].

En esta Ficha de estudio nos concentraremos en el tratamiento de **archivos binarios**, dejando la gestión y aplicación de archivos de texto para una ficha posterior. Además, si bien es perfectamente posible almacenar datos de cualquier tipo en un archivo binario, nuestro enfoque estará orientado al trabajo con *archivos binarios en los que se guardarán registros*, debido a que el registro es una estructura de datos que permite describir muy bien a cualquier entidad de la cual se quiera almacenar datos; y analizaremos además distintas estrategias de aplicación para combinar el uso de arreglos, registros y archivos en situaciones de programación típicas.

## 2.] Archivos binarios en Python: conceptos básicos.

En Python el acceso a datos almacenados en dispositivos externos en forma de archivos, o a datos gestionados mediante algún otro tipo de recurso (como un buffer interno, o como el sistema estándar de entrada/salida) se realiza mediante *objetos*<sup>3</sup> conocidos como *file objects* o como *file-like objects*. Estos objetos se crean y se dejan disponibles para el programador a través de la función interna *open()* de Python y a partir de allí cada *file object* brinda acceso a **una colección de métodos** (funciones contenidas en el objeto) que facilitan el manejo de los datos representados por ese objeto [2] [3].

La siguiente instrucción en Python crea una **variable m** asociada a un *file object* y deja abierto el archivo *datos.dat* en modo de *grabación*:

```
m = open("datos.dat", "w")
```

El primer parámetro de la función es una cadena de caracteres con el *nombre físico* (o *file descriptor*) del archivo a abrir: es el nombre con el cual el archivo figura grabado en el sistema de carpetas del sistema operativo. Este nombre físico puede incluir también la ruta de acceso a ese archivo:

```
m = open("c:\\documents\\datos.dat", "w")
```

Si no se indica la ruta de acceso, el intérprete Python lo buscará en la carpeta actual del proyecto o programa que se está ejecutando.

---

<sup>3</sup> Esencialmente, un *objeto* es una variable similar a un registro, pero tal que además de contener *campos* de datos (llamados *atributos* en un objeto) contienen también *funciones* (designadas como *métodos* cuando forman parte de un objeto).

El segundo parámetro es otra cadena de caracteres indicando el *modo de apertura para el archivo*. Estos modos en general son los mismos que los del lenguaje C, y con los mismos significados (ver *Tabla 1*). Al igual que en C, si se agrega una 'b' como último carácter en la cadena que especifica el modo de apertura entonces el archivo será tratado como *archivo binario*. Si en lugar de una 'b' se agrega una 't' (o no se agrega ninguna de ambas) entonces el archivo será considerado como un archivo de texto:

**Tabla 1: Distintos modos de apertura de un archivo en Python.**

Modo	Significado
r (o rt)	El archivo se abre como <i>archivo de texto</i> en <i>modo de solo lectura</i> (no está permitido grabar). No será creado en caso de no existir previamente. Este es el modo por defecto si se invoca a <i>open()</i> sin especificar modo de apertura alguno.
w (o wt)	El archivo se abre como <i>archivo de texto</i> en <i>modo de solo grabación</i> . Si ya existía, su contenido se perderá y se abrirá vacío. Si el archivo no existía, será creado.
a (o at)	El archivo se abre como <i>archivo de texto</i> en <i>modo de solo append</i> (todas las grabaciones se hacen al final del archivo, <i>preservando su contenido previo</i> si el archivo ya existía). Si no existía, será creado.
r+ (o r+t)	El archivo se abre como <i>archivo de texto</i> en <i>modo de lectura y grabación</i> . El archivo debe existir previamente: no será creado en caso de no existir.
w+ (o w+t)	El archivo se abre como <i>archivo de texto</i> en <i>modo de grabación y lectura</i> . Si ya existía su contenido será eliminado y abierto vacío. Si no existía, será creado.
a+ (o a+t)	El archivo se abre como <i>archivo de texto</i> en <i>modo de lectura y de append</i> (todas las <i>grabaciones</i> se hacen al final del archivo, preservando su contenido previo). Si no existía, será creado.
rb	El archivo se abre como <i>archivo binario</i> en <i>modo de sólo lectura</i> . No será creado en caso de no existir previamente.
wb	El archivo se abre como <i>archivo binario</i> en <i>modo de sólo grabación</i> . Si ya existía su contenido será eliminado. Si no existía, será creado.
ab	El archivo se abre como <i>archivo binario</i> en <i>modo de sólo append</i> (todas las grabaciones se hacen al final del archivo, preservando su contenido previo si el archivo ya existía). Si no existía, será creado.
r+b	El archivo se abre como <i>archivo binario</i> en <i>modo de lectura y grabación</i> . El archivo debe existir previamente: no será creado en caso de no existir.
w+b	El archivo se abre como <i>archivo binario</i> en <i>modo de grabación y lectura</i> . Si ya existía su contenido será eliminado. Si no existía, será creado.
a+b	El archivo se abre como <i>archivo binario</i> en <i>modo de lectura y de append</i> (todas las <i>grabaciones</i> se hacen al final del archivo, preservando su contenido previo si ya existía). Si no existía, será creado.

La función *open()* es la encargada de abrir el canal de comunicación entre el dispositivo que contiene al archivo y la memoria principal. El objeto creado y returnedo por la función (que en nuestros ejemplos se designó con la variable *m*) se aloja en memoria y contiene diversos atributos con datos que permiten manejar el archivo. Por ese motivo, ese objeto se suele designar como el *manejador del archivo* [1].

Así como es necesario abrir un archivo para poder acceder a su contenido, también es necesario *cerrarlo* (o sea, cerrar el canal de comunicación) cuando se termine de usarlo. El método *close()* se usa para cerrar esa conexión y liberar además cualquier recurso que estuviese asociado al archivo. Luego de invocar a *close()*, la variable u objeto que representaba al archivo (el manejador del archivo) queda indefinida. Note, además, que en Python los archivos son cerrados automáticamente cuando la variable para accederlos sale

del ámbito en que fue definida, por lo que en muchos contextos no es estrictamente necesario que el programador invoque a `close()`. Pero en general, el programador debe estar seguro que el archivo que estuvo manejando se cierre oportunamente, ya sea en forma manual o en forma automática. La siguiente secuencia elemental de instrucciones abre un archivo en modo 'wb'. Si el archivo no existía, será entonces creado. E inmediatamente luego de crearlo/abrirlo, se procede a cerrarlo, sin grabar nada en su interior:

```
m = open('datos.dat', 'wb')
m.close()
```

Lo único que hará el script anterior será crear el archivo `datos.dat` en la carpeta del proyecto, dejándolo vacío (tamaño = 0).

El modo de apertura es muy importante al invocar a `open()`: el script anterior crea el archivo si el mismo no existía o lo abre y elimina su contenido si ya existía. Si en lugar de 'wb' se usase el modo 'ab', el efecto será el mismo, pero con una pequeña diferencia: si el archivo ya existiese, su contenido no será eliminado y los nuevos datos se agregarán al final (por lo cual el modo 'ab' suele ser el que se usa para permitir agregar nuevos datos a un archivo). Ahora bien: si el modo de apertura fuese 'rb' (sólo lectura) y el archivo que se quiere abrir no existe en el momento de invocar a `open()`, entonces el programa lanzará un *error de runtime* (lo que se conoce como una *excepción*) y se interrumpirá:

```
m = open('noexiste.num', 'rb')

Traceback (most recent call last):
576
  File "C:/[F22] Archivos/test00.py", line 20, in <module>
    test()
  File "C:/[F22] Archivos/test00.py", line 11, in test
    m = open('noexiste.num', 'rb')
FileNotFoundError: [Errno 2] No such file or directory: 'noexiste.num'
```

Los objetos de tipo `file` (*file object*) que se crean con `open()` contienen numerosos métodos adicionales para el manejo del archivo. Entre ellos, existen los métodos `read()` y `write()` que permiten respectivamente *leer* y *grabar datos en el archivo* [2]. Ambos métodos son directos y simples de usar cuando se trata de *archivos de texto*, pero no son tan directos ni tan simples cuando se trata de leer o grabar en un *archivo binario*, sobre todo si la intención es trabajar con registros.

Por este motivo, para la lectura y grabación de datos en un *archivo binario* emplearemos una técnica diferente, mucho más simple, que en general se designa como *serialización*. La *serialización* es un proceso por el cual el contenido de una variable normalmente de **estructura compleja** (como puede ser un registro, un objeto, una lista, etc.) **se convierte automáticamente en una secuencia de bytes listos para ser almacenados en un archivo**, pero de tal forma que luego esa secuencia de bytes puede recuperarse desde el archivo y volver a crear con ella la estructura de datos original [2] [3]. El mecanismo de *serialización* no sólo está disponible en *Python* sino también en muchos otros lenguajes (sobre todo orientados a objetos, *Java* entre ellos).

El mecanismo de serialización en *Python* puede ser aplicado empleando distintos módulos y funciones según la necesidad del programador. En nuestro caso, emplearemos el módulo ***pickle***, que entre otras funciones, provee las dos que necesitaremos: ***dump()*** y ***load()***.

Si bien lo normal es que se use serialización para almacenar (o *preservar*, y de allí el curioso nombre de *pickle*) en un archivo variables de estructura compleja, no hay problema en usarla para variables simples (como variables de tipo entero, flotante o boolean, por ejemplo) [2]. El siguiente programa está incluido como *test01.py* en el proyecto [F22] Archivos que acompaña a esta Ficha, y muestra un ejemplo simple de grabación y lectura de números enteros y flotantes en un archivo:

```
import os.path
import pickle

__author__ = 'Cátedra de AED'

def test():
    print('Procediendo a grabar números en el archivo')
    m = open('prueba.num', 'wb')
    x, y = 2.345, 19
    pickle.dump(x, m)
    pickle.dump(y, m)
    m.close()

    m = open('prueba.num', 'rb')
    a = pickle.load(m)
    b = pickle.load(m)
    m.close()
    print('Datos recuperados desde el archivo:', a, ' - ', b)

    print('Hecho...')

if __name__ == '__main__':
    test()
```

El programa anterior es simple pero muy ilustrativo: el primer bloque abre el archivo *prueba.num* (en modo 'wb', de forma que si no existía lo crea, y si ya existía elimina su contenido) y lo deja abierto para permitir grabaciones. La variable para manejar el archivo es *m*, en la que se ha asignado el objeto returned por *open()*. Luego, se usa dos veces la función *pickle.dump()* para grabar en el archivo los valores de las variables *x* e *y* (valiendo 2.345 y 19 respectivamente). Ambos números son grabados en el archivo representado por *m*, uno a continuación del otro, haciendo que el archivo aumente su tamaño en bytes (que inicialmente era cero). Al terminar estas dos grabaciones, el archivo se cierra con *m.close()*.

El hecho de cerrar el archivo al terminar de grabar, permite que ahora el mismo pueda ser reabierto pero *cambiando el modo de apertura*: como en este momento se pretende leer su contenido, se vuelve a abrir el mismo archivo pero ahora en modo 'rb', y a continuación se invoca dos veces a la función *pickle.load()* para leer el contenido del archivo: la primera invocación a *pickle.load()* traerá desde el archivo una copia del valor 2.345, y la segunda traerá una copia del 19. Ambos valores son almacenados en las variables *a* y *b* respectivamente. El archivo no pierde sus datos después de ser leído.

La función *pickle.dump()* toma dos parámetros: el primero es la variable cuyos datos se quieren grabar, y la segunda es la variable manejadora del archivo (creada previamente con *open()*) en el cual se desea grabar el valor de la variable que entra como primer parámetro. Y la función *pickle.load()* sólo toma un parámetro: la variable manejadora del archivo que se

quiere leer. Al terminar la lectura, `pickle.load()` reconstruye la estructura leída (sea simple o compleja) y la retorna [2].

Un ejemplo algo más complejo muestra la forma en que podemos hacer lo mismo, pero con un variables de tipo registro en lugar de variables simples (ver modelo `test02.py` en el proyecto [F22] Archivos):

```
import pickle

__author__ = 'Catedra de AED'

class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut

def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)

def test():
    print('Prueba de grabación de varios registros...')
    lib1 = Libro(2134, 'Fundación', 'Isaac Asimov')
    lib2 = Libro(5587, 'Fundación e Imperio', 'Isaac Asimov')
    lib3 = Libro(3471, 'Segunda Fundación', 'Isaac Asimov')

    fd = 'libros.dat'
    m = open(fd, 'wb')
    pickle.dump(lib1, m)
    pickle.dump(lib2, m)
    pickle.dump(lib3, m)
    m.close()
    print('Se grabaron varios registros en el archivo', fd)

    m = open(fd, 'rb')
    lib1 = pickle.load(m)
    lib2 = pickle.load(m)
    lib3 = pickle.load(m)
    m.close()

    print('Se recuperaron estos registros desde el archivo', fd, ':')
    display(lib1)
    display(lib2)
    display(lib3)

if __name__ == '__main__':
    test()
```

El modelo que acabamos de mostrar define un tipo registro llamado *Libro*, y las típicas funciones `init()` para inicializar un *Libro*, y `display()` para mostrar un *Libro*. La función `test()` crea tres registros y procede luego a grabar esos registros en un archivo `libros.dat`, manejado a través de la variable `m`. Cada una de las invocaciones a `pickle.dump()` graba el registro que toma como parámetro en el archivo representado por `m`, y lo hace de forma que el

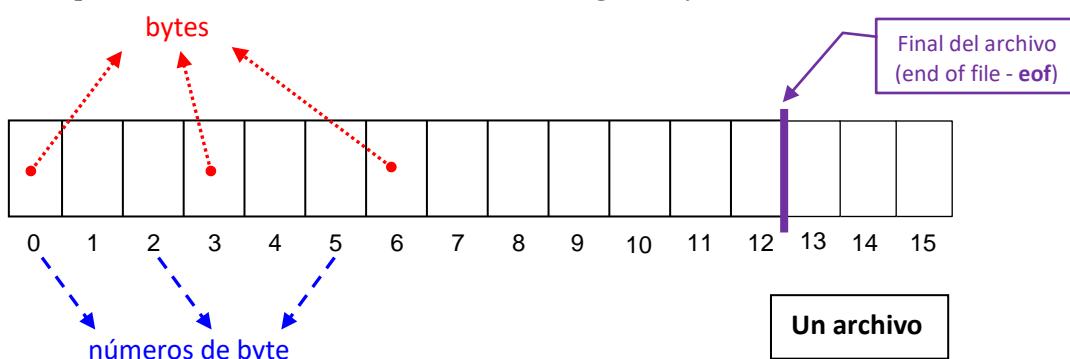
programador no deba preocuparse por los detalles de cómo se grabará cada campo. Los registros son grabados uno a continuación de otro, y luego el archivo se cierra.

La segunda parte de la función `test()` vuelve a abrir el archivo `libros.dat`, pero ahora en modo de sólo lectura, y procede a leer el contenido del mismo usando tres invocaciones a la función `pickle.load()`. En cada una de ellas, se recupera el contenido completo de uno de los registros grabados previamente, y cada registro recuperado se asigna nuevamente en las mismas variables iniciales. El archivo luego se cierra, y se muestran en pantalla los tres registros. Se puede observar que el mecanismo es exactamente el mismo al que se aplicó cuando se grabaron variables simples en el modelo `test01.py` anterior.

### 3.] Archivos binarios en Python: recorrido secuencial y acceso directo.

Esencialmente, y para completar nuestra aproximación conceptual de la sección anterior, un *archivo* (binario o de texto) puede entenderse como un gran *array o vector de bytes* ubicado en memoria externa en lugar de estar alojado en memoria principal. Cada *componente* de ese gran *array en memoria externa* es uno de los bytes grabados en el archivo, y cada byte tiene (a modo de índice) un número que lo identifica, correlativo desde el cero en adelante. El *byte cero* es el primer byte del archivo. Notar que entonces el número de cada byte es un *indicador de su posición relativa al inicio del archivo* [1]:

**Figura 1:** Esquema conceptual de un archivo como un arreglo de bytes en memoria externa.



El sistema operativo es el responsable de recordar en qué lugar termina un archivo. Ese punto se conoce como el *final del archivo* o *end of file* (abreviado comúnmente como *eof* por su significado en inglés). Es importante notar que los bytes que siguen al final del archivo (los bytes desde el 13 en adelante en la figura anterior) no pertenecen al archivo pero aún así están numerados en forma correlativa, continuando con la numeración que traía el archivo.

Y un detalle interesante que se deduce de lo anterior, es que debido al hecho de que la numeración de los bytes comienza en cero, entonces el *número total de bytes* (o *tamaño*) del archivo *coincide con el número del primer byte que se encuentra fuera del mismo* (en el gráfico anterior, el archivo tiene 13 bytes, y el byte 13 es el primero que está fuera del archivo: el tamaño del archivo es de 13 bytes).

En muchas situaciones el programador necesita obtener el *tamaño en bytes de un archivo*. Hay varias formas de hacerlo en Python, pero el siguiente ejemplo muestra la que quizás sea la más simple, usando la función `getsize()` incluida en el módulo `os.path` [2]:

```
import os.path
```

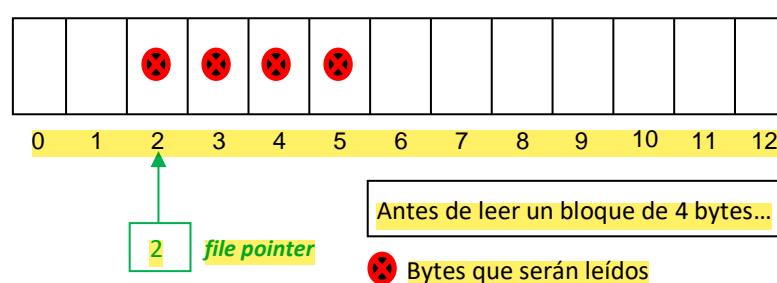
```
t = os.path.getsize('libros.dat')
print(t)
```

La función `os.path.getsize()` toma como parámetro el nombre físico del archivo (que aquí suponemos que se llama '*libros.dat*', y simplemente retorna su tamaño en bytes. Note que no es necesario que el archivo esté abierto con `open()` para poder usar esta función.

En general, todo archivo cuenta con una especie de *cursor* o *indicador* o *marcador interno* llamado **file pointer**, que no es otra cosa que una variable de tipo entero tal que, mientras el archivo está abierto, contiene el número del byte sobre el cual se realizará la próxima operación de lectura o de grabación. Tanto las operaciones de lectura como las de grabación, comienzan la operación en el byte indicado por el **file pointer** y al terminar la misma, dejan el **file pointer** apuntando al byte siguiente a aquel en el cual terminó la operación. Si una operación de salida graba bytes detrás del byte de finalización del archivo, entonces el tamaño del archivo crece y se ajusta para abarcar los nuevos bytes [1].

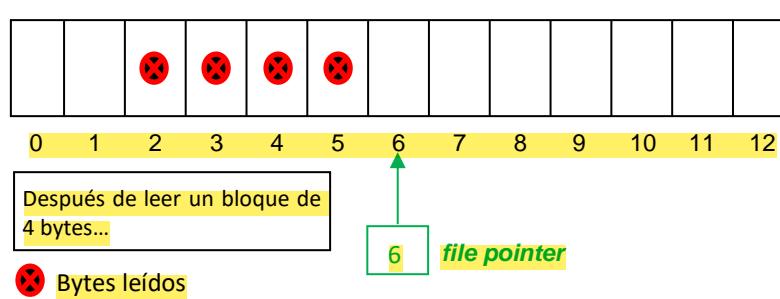
Por ejemplo, supongamos el mismo archivo de la *Figura 1*, con 13 bytes ya grabados. Supongamos también en un momento dado el **file pointer** está apuntando al byte 2 de este archivo:

**Figura 2: Esquema del estado del *file pointer* en un archivo antes de proceder a una lectura.**



Si en estas condiciones se lanza una operación de lectura para recuperar desde el archivo un bloque de cuatro bytes (por ejemplo, con la función `load()`), entonces dicha operación leerá desde el archivo 4 bytes comenzando desde el byte 2 (pues allí apunta ahora el **file pointer**). Al terminar de leer, el **file pointer** quedará apuntando al byte 6 (pues la lectura llegó hasta el 5 inclusive):

**Figura 3: Esquema del estado del *file pointer* en un archivo después de proceder a una lectura.**



En general el valor del **file pointer** es gestionado automáticamente por las funciones y métodos básicos para gestión de archivos: `open()` abre el archivo y asigna el valor inicial al

**file pointer** (típicamente el valor cero), mientras que `dump()` y `load()` (si se aplica serialización a través de `pickle`) ajustan el valor del **file pointer** para dejarlo apuntando al byte en donde comenzará la próxima operación. Si el programador se limita a usar estos mecanismos grabando y leyendo datos estrictamente en el orden que surge de este manejo automático del file pointer, entonces *el archivo siempre será accedido y recorrido en forma estrictamente secuencial*, comenzando desde el primer byte y avanzado sobre el resto sin saltarse ninguno.

El acceso y recorrido en forma secuencial es común y simple de implementar. Muchos procesos requieren justamente tomar todos los datos de un archivo, uno por uno en el orden en que aparecen, y procesarlos en ese orden. Por ejemplo, esto será necesario si se desea hacer un listado de todo el contenido del archivo, o buscar un dato particular en ese archivo.

Pero en muchas otras ocasiones, el programador necesitará poder acceder en forma directa a ciertos datos o posiciones en el archivo, y en casos así sería deseable no tener que pasar en forma secuencial por los datos que estén grabados antes que los que se quiere acceder, para evitar la pérdida de tiempo. Y para lograr esto, el programador necesita poder controlar en forma manual el valor del **file pointer**.

En ese sentido, los objetos tipo *file object* creados con `open()` contienen un método `tell()` que retorna el valor del **file pointer** en forma de un número entero, y también un método `seek()` que permite cambiar el valor del mismo (y por lo tanto, `seek()` permite elegir cuál será el próximo byte que será leído o grabado) [2] [3].

La función `tell()` es muy útil cuando por algún motivo se requiere saber en qué byte está posicionado un archivo en un momento dado. Esto es particularmente necesario cuando se quiere leer el contenido completo de un archivo en forma secuencial, usando un ciclo. Suponga que se tiene un archivo que contiene numerosos registros grabados mediante `pickle.dump()` y se necesita leer ese archivo y mostrar su contenido en pantalla. Está claro que el programador requerirá un ciclo, de forma de hacer una lectura (con `pickle.load()`) y una visualización en cada iteración. Pero si se desconoce el número de registros que contiene el archivo, ¿cómo hará el programador para controlar que no se haya llegado al final del archivo en una lectura? En general, si se intenta leer un bloque de bytes ubicado más allá del final del archivo, los métodos y funciones de lectura provocarán un error de runtime y el programa se interrumpirá.

Una forma de resolver el problema, consiste en chequear en cada vuelta del ciclo y antes de invocar a `pickle.load()` si el valor del **file pointer** es menor que el tamaño del archivo. Como vimos, el tamaño del archivo es igual al número del primer byte que se encuentra fuera del mismo y puede obtenerse con la función `os.path.getsize()`. Si el **file pointer** en un momento dado está apuntando a un byte cuyo número es menor que el tamaño, entonces el **file pointer** está posicionado en un byte que pertenece al archivo y la lectura sería en ese caso válida. Consideremos el siguiente programa a modo de ejemplo (incluido en el modelo `test03.py` del proyecto [F22] Archivos):

```
import pickle
import os.path

__author__ = 'Catedra de AED'
```

```

class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut

def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)

def test():
    print('Prueba de grabación de varios registros...')
    lib1 = Libro(2134, 'Fundación', 'Isaac Asimov')
    lib2 = Libro(5587, 'Fundación e Imperio', 'Isaac Asimov')
    lib3 = Libro(3471, 'Segunda Fundación', 'Isaac Asimov')
    lib4 = Libro(1122, 'Los Límites de la Fundación', 'Isaac Asimov')
    lib5 = Libro(2286, 'Fundación y Tierra', 'Isaac Asimov')

    fd = 'libros.dat'
    m = open(fd, 'ab')
    pickle.dump(lib1, m)
    pickle.dump(lib2, m)
    pickle.dump(lib3, m)
    pickle.dump(lib4, m)
    pickle.dump(lib5, m)
    m.close()
    print('Se grabaron varios registros en el archivo', fd)

    m = open(fd, 'rb')
    t = os.path.getsize(fd)

    print('Se recuperaron estos registros desde el archivo', fd, ':')
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)
    m.close()

    t = os.path.getsize(fd)
    print('Tamaño del archivo al terminar:', t, 'bytes')

if __name__ == '__main__':
    test()

```

La función `test()` almacena varios registros de tipo `Libro` en el archivo `libros.dat` y luego lo cierra. Inmediatamente después se usa la función `os.path.getsize()` para obtener el tamaño en bytes del archivo (que se almacena en la variable `t`):

```

# fd = 'libros.dat'

m = open(fd, 'rb')
t = os.path.getsize(fd)

```

Luego, la función `test()` lanza un ciclo para leer en forma secuencial el archivo y mostrar su contenido:

```

print('Se recuperaron estos registros desde el archivo', fd, ':')
while m.tell() < t:
    lib = pickle.load(m)
    display(lib)
m.close()

```

En cada vuelta de ese ciclo, se usa el método `tell()` para controlar si el valor del *file pointer* del archivo gestionado con `m` es menor que el tamaño en bytes del archivo (almacenado en la variable `t`). Si eso es cierto, el ciclo ejecuta su bloque de acciones, invocando a `pickle.load()` para leer el siguiente registro, mostrándolo por pantalla inmediatamente después. Como cada vez que se hace una lectura (o una grabación) el valor del *file pointer* pasa automáticamente al inicio del siguiente bloque a leer, el resultado será que en algún momento el ciclo se detendrá por llegar al final del archivo (en cuyo caso el *file pointer* estará apuntando al primer byte exterior al archivo).

La técnica que hemos mostrado permite controlar con mucha sencillez el recorrido y lectura secuencial de un archivo, y será aplicada en numerosas oportunidades prácticas.

Por su parte, el método `seek(offset, from_what)` permite cambiar el valor del *file pointer*. En general recibe dos parámetros: el primero (que aquí llamamos `offset`) indica cuántos bytes debe moverse el *file pointer*, y el segundo (llamado aquí `from_what`, si está presente) indica desde donde se hace ese salto (un valor 0 indica saltar desde el principio del archivo, un 1 desde donde está el *file pointer* en este momento y un 2 indica saltar desde el final). El valor por default del segundo parámetro es 0, por lo cual por omisión se asume que los saltos son desde el inicio del archivo. El valor del segundo parámetro puede ser indicado como un número o por medio de una de las siguientes tres constantes predefinidas (las tres dentro del módulo `io` que debe ser importado en el programa para poder usarlas) [2]:

Tabla 2: Constantes predefinidas para el método `seek()`.

Constante	Valor	Significado
<code>io.SEEK_SET</code>	0	Reposicionar comenzando desde el principio del archivo. El valor del primer parámetro ( <code>offset</code> ) puede ser 0 o positivo (pero no negativo).
<code>io.SEEK_CUR</code>	1	Reposicionar comenzando desde la posición actual del puntero de registro activo. El valor de <code>offset</code> puede ser entonces negativo, 0 o positivo.
<code>io.SEEK_END</code>	2	Reposicionar comenzando desde el final del archivo. El valor de <code>offset</code> típicamente es negativo, aunque puede ser 0 o positivo.

Si no se indica el segundo parámetro al invocar a `seek()`, se asume por defecto que su valor es 0, y en ese caso el reposicionamiento se hará desde el inicio del archivo.

En general, la forma de entender el funcionamiento de este método consiste en suponer que el *file pointer* está apuntando al lugar indicado por el segundo parámetro, y sumar a esa posición el valor del primer parámetro. Veamos algunos ejemplos. Siempre suponemos la instrucción `import io` está incluida en el programa, que el archivo está abierto, y que la variable manejadora del archivo se llama `m`:

- a.) Sin importar donde esté ubicado el *file pointer* en este momento, suponga que queremos cambiar su valor para que pase a valer 10. Podemos hacerlo así:

```
m.seek(10, io.SEEK_SET)
```

Si el segundo parámetro es SEEK\_SET (o sea, 0), se debe asumir que el file pointer está ubicado al inicio del archivo (o sea, su valor sería el 0), a ese valor sumarle el 10 que se pasó como primer parámetro, y asignar el resultado en el *file pointer* que pasará entonces a valer el número 10.

- b.) Sin importar donde esté ubicado el *file pointer* en este momento, suponga que queremos cambiar su valor para que salte al final del archivo. Podemos hacerlo así:

```
m.seek(0, io.SEEK_END)
```

Si el segundo parámetro es SEEK\_END (o sea, 2), se debe asumir que el file pointer está ubicado al final del archivo (o sea, su valor sería el número del primer byte que está fuera del archivo), a ese valor sumarle el 0 que se pasó como primer parámetro, y asignar el resultado al *file pointer* que pasará entonces a valer el número del primer byte que está fuera del archivo (es decir, el *file pointer* quedará apuntando al final).

- c.) Suponga que el *file pointer* en este momento está apuntando al byte 7 del archivo (el valor del *file pointer* es 7), y suponga que queremos cambiar su valor para que pase a apuntar al byte 4. Podemos hacerlo así:

```
m.seek(-3, io.SEEK_CUR)
```

Si el segundo parámetro es SEEK\_CUR (o sea, 1), se debe asumir que el file pointer está ubicado en su posición actual (o sea, su valor sería el byte 7 que ya tenía), a ese valor sumarle el -3 que se pasó como primer parámetro, y asignar el resultado en el file pointer. Como  $7 + (-3) = 4$ , el *file pointer* pasará a valer 4.

- d.) Está claro que el ejemplo anterior también podría haber sido resuelto así:

```
m.seek(4, io.SEEK_SET)
```

Si el segundo parámetro es SEEK\_SET (o sea, 0), se debe asumir que el file pointer está ubicado al inicio del archivo (o sea, su valor sería el 0), a ese valor sumarle el 4 que se pasó como primer parámetro, y asignar el resultado en el file pointer, que pasará a valer 4.

Aunque el método *seek()* admite todas estas variantes, en la práctica lo más común es que se utilice siempre con el segundo parámetro valiendo *io.SEEK\_SET*, que es claramente lo más cómodo: si se usa *io.SEEK\_SET*, entonces el valor pasado como primer parámetro es directamente el valor que tomará el file pointer, y el programador no tiene que realizar en este caso ningún cálculo auxiliar.

No obstante lo anterior, ocurre con frecuencia que el programador necesite llevar el *file pointer* al final del archivo, y en ese caso el ejemplo b.) muestra la forma de hacerlo. Una aplicación de esto consiste en la siguiente forma alternativa de calcular el tamaño del archivo (sin usar la función *os.path.getsize()*):

```
def size(fd):
    file = open(fd, 'rb')
    file.seek(0, io.SEEK_END)
    t = file.tell()
    file.close()
    return t
```

La función anterior toma como parámetro una cadena de caracteres que representa el nombre físico del archivo cuyo tamaño se quiere medir. Luego abre el archivo con *open()* (en modo de sólo lectura es suficiente) y posiciona el file pointer el final del archivo con *seek()*. Al hacer esto, el file pointer queda valiendo el número del primer byte que está fuera del

archivo, que como ya sabemos, es también igual al tamaño del archivo. Por lo tanto, se invoca a `tell()` para obtener ese valor, y se lo retorna sin más.

El uso del método `seek()` permite cambiar el valor del *file pointer* a voluntad del programador, de forma que luego puede leer o grabar en cualquier posición del archivo sin tener que hacer un recorrido secuencial previo. Si el programador sabe o puede calcular en qué byte específico del archivo necesita leer o grabar, sólo debe invocar a `seek()`, cambiar el valor del *file pointer* al lugar requerido, y luego leer o grabar.

Si `m` es la variable para manejar un archivo, y ese archivo está abierto en un modo que permite tanto leer como grabar, la siguiente secuencia grabará el contenido de la variable `x` (sea `x` del tipo que sea...) a partir del byte cuyo número es `p` dentro del archivo:

```
m.seek(p, io.SEEK_SET)
pickle.dump(x, m)
```

Lo anterior es equivalente a:

```
m.seek(p)
pickle.dump(x, m)
```

ya que si el segundo parámetro en `seek()` se omite, se asume que vale justamente `io.SEEK_SET`.

No olvide que luego de grabar o leer, las funciones `dump()` y `load()` cambian automáticamente el valor del *file pointer* para que quede apuntando al byte siguiente a aquel en el cual terminó la operación. Por lo tanto, en el ejemplo anterior, el valor del *file pointer* al terminar la grabación no será el mismo que se le asignó con `seek()`.

Y si lo que buscaba era leer en lugar de grabar, la siguiente secuencia leerá los bytes que correspondan desde el archivo gestionado por `m`, comenzando desde la posición `p`, y retornará la estructura reconstruida asignándola en `x`:

```
# m.seek(p)
m.seek(p, io.SEEK_SET)
x = pickle.load(m)
```

Cuando se trabaja de esta forma se dice que el programador está realizando *acceso directo* (o también *acceso aleatorio*) al contenido del archivo, en lugar del *acceso secuencial* que antes hemos analizado. Ambas técnicas pueden tranquilamente coexistir en un mismo programa, del mismo modo que el procesamiento de un arreglo puede requerir recorridos secuenciales y accesos directos a sus componentes en la misma aplicación.

#### 4.] Procesamiento combinado de registros, arreglos y archivos.

Se propone ahora un caso de aplicación en el que se pedirá combinar el uso de distintas estructuras de datos, tales como arreglos, registros y archivos. Cada una de las situaciones será analizada y resuelta, y en el análisis que corresponda a cada una de ellas se explicarán algunas técnicas elementales de procesamiento combinado.

**Problema 53.)** *Desarrollar un programa controlado por menú de opciones, que permita gestionar un arreglo de registros (puede suponer el mismo tipo Libro que hemos usado como ejemplo en esta misma Ficha), y a partir de las opciones del menú proceda a generar un*

*archivo con los datos del arreglo (o viceversa: volver a crear el arreglo a partir de los datos del archivo). Las opciones que debería incluir son las siguientes:*

- a.) *Crear y cargar un arreglo v de n registros de tipo Libro (puede hacer esta carga en forma automática).*
- b.) *Mostrar el arreglo.*
- c.) *Crear un archivo libros.dat que contenga todos los registros del arreglo original. Si el archivo ya existía, eliminar su contenido y comenzar desde cero.*
- d.) *Mostrar el contenido del archivo libros.dat.*
- e.) *Crear nuevamente el archivo libros.dat, de forma que ahora contenga sólo los datos de los libros cuyo código sea menor que x, cargando el código x por teclado. Si el archivo ya existía, eliminar su contenido y comenzar desde cero.*
- f.) *A partir del archivo libros.dat, volver a crear en memoria el arreglo v, de forma que contenga todos los registros del archivo. Reemplace el arreglo creado en el punto a.) por el que se le pide crear ahora.*
- g.) *A partir del archivo libros.dat, volver a crear en memoria el arreglo v, que contenga sólo los registros de los libros cuyo código sea mayor a x (cargue x por teclado). Reemplace el arreglo creado en el punto a.) por el que se le pide crear ahora.*

**Discusión y solución:** El proyecto [F22] Archivos que acompaña a esta Ficha contiene un modelo *test04.py* con el programa completo que resuelve este caso de análisis.

La declaración del registro *Libro* es directa y similar a los hecho en casos anteriores. Se importan además algunos módulos que serán necesarios en distintos puntos del desarrollo del programa:

```
import random
import pickle
import os
import os.path
import io

__author__ = 'Catedra de AED'

class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut

    def display(libro):
        print('ISBN:', libro.isbn, end='')
        print(' - Título:', libro.titulo, end='')
        print(' - Autor:', libro.autor)
```

Como es costumbre, la función *test()* gestiona el menú principal, y en ella se crea el vector *v* inicialmente vacío, además de una variable *fd* que contiene el nombre del archivo con el que se trabajará:

```
def test():
    v = []
```

```

fd = 'libros.dat'
op = -1
while op != 8:
    print('Procesamiento combinado de arreglos, registros y archivos...')
    print('1. Crear el arreglo de libros (en forma automática)')
    print('2. Mostrar el arreglo de libros')
    print('3. Crear el archivo con TODOS los libros del arreglo')
    print('4. Mostrar el archivo de libros')
    print('5. Crear el archivo con ALGUNOS de los libros del arreglo')
    print('6. Volver a crear el arreglo con TODOS los libros del archivo')
    print('7. Volver a crear el arreglo con ALGUNOS de los libros del archivo')
    print('8. Salir')
    op = int(input('\t\tIngrese número de opción: '))
    print()

    if op == 1:
        cargar_arreglo(v)

    elif op == 2:
        mostrar_arreglo(v)

    elif op == 3:
        crear_archivo_todos(v, fd)

    elif op == 4:
        mostrar_archivo(fd)

    elif op == 5:
        crear_archivo_algunos(v, fd)

    elif op == 6:
        v = crear_arreglo_todos(fd)

    elif op == 7:
        v = crear_arreglo_algunos(fd)

    elif op == 8:
        pass

if __name__ == '__main__':
    test()

```

De aquí en más, siguen las funciones invocadas desde el menú. Las dos primeras son las encargadas de crear y cargar el arreglo en forma automática (evitando la carga por teclado, para ganar tiempo en este programa de prueba) y de mostrar el contenido del arreglo:

```

def cargar_arreglo(v):
    n = int(input('Cuantos libros desea cargar?: '))
    for i in range(n):
        cod = random.randint(1, 10000)
        tit = 'Título ' + str(i)
        aut = 'Autor ' + str(i)
        lib = Libro(cod, tit, aut)
        v.append(lib)

def mostrar_arreglo(v):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    print('Los libros registrados son:')
    for libro in v:

```

```
display(libro)
```

```
print()
```

La función `crear_archivo_todos()` crea el archivo pedido, almacenando en él todos los registros contenidos en el arreglo `v`:

```
def crear_archivo_todos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # pickle.dump(v, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial...
    for lib in v:
        pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con todos los registros del vector')
    print()
```

Esta función comienza chequeando si el arreglo `v` tomado como parámetro está vacío, en cuyo caso retorna al menú sin hacer nada. Luego abre el archivo en modo 'wb', y procede a grabar el contenido del arreglo `v`. Aquí tenemos dos posibles técnicas (que en el código fuente están incluidas y marcadas con comentarios): si el programador necesita que el archivo contenga directamente grabado el arreglo de tal forma que pueda ser directamente recuperado luego, entonces puede aplicar la *forma 1*, que consiste en usar `pickle.dump()` y pasarle todo el arreglo como parámetro:

```
# forma 1
m = open(fd, 'wb')

# forma 1: si se desea que el archivo contenga un vector...
pickle.dump(v, m)

m.close()
```

El mecanismo de serialización está definido de manera que es posible no sólo grabar y recuperar variables simples y registros: en rigor, la serialización es una técnica general para preservar el contenido de cualquier tipo de objeto, sin importar su complejidad interna. Las variables de tipo list, así como las tuplas, las cadenas, y los rangos, son objetos... y pueden ser serializados en forma directa, con una sola invocación a `pickle.dump()`, o recuperados (des-serializados) con una sola invocación a `pickle.load()`.

La forma anterior de guardar todo el arreglo en un archivo es muy útil cuando el programador está seguro de que luego volverá a recuperar el arreglo entero, y que no necesitará procesar uno por uno los registros que se grabaron en el archivo.

Si el programador necesitase almacenar uno por uno los registros en el archivo (por ejemplo, si no está seguro de volver a necesitarlos a todos en un arreglo más adelante, o si sospecha que necesitará procesar esos registros directamente desde el archivo, sin subirlos a memoria

en un arreglo) entonces puede aplicarse la *forma 2*, que consiste en usar un ciclo para recorrer el arreglo y grabar uno por uno todos los registros:

```
# forma 2
for lib in v:
    pickle.dump(lib, m)
```

que sería exactamente lo mismo que:

```
# forma 2
for i in range len(v):
    pickle.dump(v[i], m)
```

Ambas técnicas son simples y directas. La aplicación de una u otra dependerá del contexto del problema y de las necesidades del programador. En cada una de las funciones que quedan en el programa, hemos incluido ambas formas de trabajo: una de ellas, la *forma 1*, está en el código fuente pero en forma de comentarios. La otra, la *forma 2*, es la que efectivamente está activa en todo el programa. Si el estudiante quisiera activar la *forma 1* y desactivar la *forma 2*, sólo tiene que debe "apagar" con comentarios la *forma 2* y eliminar los comentarios de la *forma 1*, pero recuerde: debe hacerlo en todo el programa: sólo una de ambas formas debería estar activa, si quiere evitar luego problemas cuando quiera leer el archivo y recuperar con *pickle.load()* los objetos grabados.

La función *crear\_archivo\_algunos()* crea también el archivo pedido, pero almacenando en él sólo los registros contenidos en el arreglo *v* que tengan isbn o código menor que el valor *x* que se carga por teclado:

```
def crear_archivo_algunos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo... ')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # v2 = []
    # for lib in v:
    #     if lib.isbn < x:
    #         v2.append(lib)
    # pickle.dump(v2, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial..
    for lib in v:
        if lib.isbn < x:
            pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con los registros con código <', x)
    print()
```

La *forma 1* consiste en crear un vector *v2* que contenga referencias a los libros cuyo código sea menor a *x*, y luego grabar ese nuevo arreglo en el archivo (que de esta forma, también contendrá un arreglo en lugar de registros grabados uno por uno) Recuerde que esta

alternativa está incluida entre comentarios en el programa, y por lo tanto no es la que efectivamente se está aplicando:

```
# forma 1
v2 = []
for lib in v:
    if lib.isbn < x:
        v2.append(lib)
pickle.dump(v2, m)
```

La *forma 2*, consiste en grabar uno por uno los registros en el archivo, directamente, sin almacenarlos primero en un arreglo auxiliar (esta es la forma que está activa en el programa):

```
# forma 2
for lib in v:
    if lib.isbn < x:
        pickle.dump(lib, m)
```

que también equivale a:

```
# forma 2
for i in range(len(v)):
    if v[i].isbn < x:
        pickle.dump(v[i], m)
```

Cuando el archivo ha sido creado (con todos los registros del arreglo o con algunos), la función *mostrar\_archivo()* es la encargada de mostrar el contenido del archivo en pantalla:

```
def mostrar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    print('Contenido actual del archivo', fd, ':')
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)
    # for lib in v:
    #     display(lib)

    # forma 2: si el archivo contenía registros almacenados
    # uno a uno en forma secuencial...
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)

    m.close()
    print()
```

Lo primero que hace la función es verificar si el archivo que se quiere mostrar realmente existe. Si no fuese el caso, la función informa con un mensaje y retorna al menú sin hacer nada más. La forma de comprobar si un archivo existe o no, consiste en usar la función *os.path.exists()*, la cual toma como parámetro el nombre físico del archivo a verificar, y retorna *True* si ese archivo existe, o *False* en caso contrario:

```

if not os.path.exists(fd):
    print('El archivo', fd, 'no existe... ')
    print()
    return

```

Si todo está bien, la función *mostrar\_archivo()* abre el archivo en modo de sólo lectura y lo que sigue depende de cómo haya sido grabado el archivo: si se aplicó la *forma 1* y el archivo contiene directamente el vector de registros serializado, entonces sólo hay que recuperar ese arreglo con una sola invocación a *pickle.load()* y mostrar ese arreglo (esta técnica está incluida con comentarios en el código fuente):

```

# forma 1: si el archivo contenía un vector...
v = pickle.load(m)
for lib in v:
    display(lib)

```

O bien:

```

# forma 1: si el archivo contenía un vector...
v = pickle.load(m)
for i in range(len(v)):
    display(v[i])

```

Pero si al crear el archivo se aplicó la forma 2, entonces contiene los registros grabados uno por uno, y deben ser recuperados uno por uno (en este caso, se recuperan y se muestran, sin almacenarlos en un arreglo):

```

# forma 2: si el archivo contenía registros almacenados
# uno a uno en forma secuencial...
t = os.path.getsize(fd)
while m.tell() < t:
    lib = pickle.load(m)
    display(lib)

```

Sólo nos quedan las dos funciones que vuelven a crear el vector, a partir de los datos del archivo. La primera es la función *crear\_arreglo\_todos()*, que debe recuperar todo el contenido del archivo y volver a crear un arreglo con ellos:

```

def crear_arreglo_todos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe... ')
        print()
        return

    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        v.append(lib)

    m.close()
    print('Se creó el vector con todo el contenido del archivo', fd)

```

```

print()

return v

```

Si el archivo se creó con la *forma 1* y contiene entonces el arreglo grabado en forma directa, entonces la tarea es simple: sólo hay que invocar a *pickle.load()*, recuperarlo y retornarlo sin más antes de terminar la ejecución de la función (esta forma está desactivada con comentarios en el programa):

```

# forma 1: si el archivo contenía un vector...
v = pickle.load(m)

```

Pero si el archivo fue generado con la *forma 2*, grabando uno por uno los registros, entonces se debe leer el archivo registro por registro, con un ciclo, y almacenar en el arreglo los registros a medida que se leen: (este es el proceso que está activo en el programa):

```

# forma 2: si el archivo contenía los registros almacenados
# uno a uno en forma secuencial...
v = []
t = os.path.getsize(fd)
while m.tell() < t:
    lib = pickle.load(m)
    v.append(lib)

```

Finalmente, la función *crear\_arreglo\_algunos()* debe volver a crear el arreglo *v*, pero recuperando sólo los registros de los libros cuyo código sea mayor a *x* (cargando *x* por teclado):

```

def crear_arreglo_algunos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = []
    # v2 = pickle.load(m)
    # for lib in v2:
    #     if lib.isbn > x:
    #         v.append(lib)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        if lib.isbn > x:
            v.append(lib)

    m.close()
    print('Se creó el vector con parte del archivo', fd)
    print()

    return v

```

Y otra vez, si el archivo fue creado con la *forma 1* y contiene un arreglo completo grabado en forma directa, entonces debe recuperarse ese arreglo y a partir de él crear un segundo arreglo que contenga los registros pedidos, retornando el segundo arreglo al finalizar (esta forma está desactivada con comentarios en el código fuente):

```
# forma 1: si el archivo contenía un vector...
v = []
v2 = pickle.load(m)
for lib in v2:
    if lib.isbn > x:
        v.append(lib)
```

Y si el archivo fue creado con la *forma 2*, registro por registro, entonces esos registros deben ser leídos uno por uno de forma que se copien en el arreglo *v* sólo los que tengan código mayor a *x* (en este caso no es necesario un segundo arreglo):

```
# forma 2: si el archivo contenía los registros almacenados
# uno a uno en forma secuencial...
v = []
t = os.path.getsize(fd)
while m.tell() < t:
    lib = pickle.load(m)
    if lib.isbn > x:
        v.append(lib)
```

El programa completo se muestra a continuación:

```
import random
import pickle
import os
import os.path
import io

__author__ = 'Catedra de AED'

class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut

def display(libro):
    print('ISBN:', libro.isbn, end='')
    print(' - Título:', libro.titulo, end='')
    print(' - Autor:', libro.autor)

def cargar_arreglo(v):
    n = int(input('Cuantos libros desea cargar?: '))
    for i in range(n):
        cod = random.randint(1, 10000)
        tit = 'Título ' + str(i)
        aut = 'Autor ' + str(i)
        lib = Libro(cod, tit, aut)
        v.append(lib)

def mostrar_arreglo(v):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
    print()
```

```

    return

print('Los libros registrados son:')
for libro in v:
    display(libro)

print()

def crear_archivo_todos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # pickle.dump(v, m)

    # forma 2: si se desea que al archivo contenga los registros
    # almacenados uno a uno en forma secuencial..
    for lib in v:
        pickle.dump(lib, m)

    m.close()
    print('Se creó el archivo', fd, 'con todos los registros del vector')
    print()

def mostrar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    print('Contenido actual del archivo', fd, ':')
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)
    # for lib in v:
    #     display(lib)

    # forma 2: si el archivo contenía registros almacenados
    # uno a uno en forma secuencial...
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        display(lib)

    m.close()
    print()

def crear_archivo_algunos(v, fd):
    if len(v) == 0:
        print('No hay datos en el arreglo...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'wb')

    # forma 1: si se desea que el archivo contenga un vector...
    # v2 = []
    # for lib in v:
    #     if lib.isbn < x:

```

```
#           v2.append(lib)
# pickle.dump(v2, m)

# forma 2: si se desea que al archivo contenga los registros
# almacenados uno a uno en forma secuencial..
for lib in v:
    if lib.isbn < x:
        pickle.dump(lib, m)

m.close()
print('Se creó el archivo', fd, 'con los registros con código <', x)
print()

def crear_arreglo_todos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = pickle.load(m)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        v.append(lib)

    m.close()
    print('Se creó el vector con todo el contenido del archivo', fd)
    print()

    return v

def crear_arreglo_algunos(fd):
    if not os.path.exists(fd):
        print('El archivo', fd, 'no existe...')
        print()
        return

    x = int(input('Ingrese el código a comparar: '))
    m = open(fd, 'rb')

    # forma 1: si el archivo contenía un vector...
    # v = []
    # v2 = pickle.load(m)
    # for lib in v2:
    #     if lib.isbn > x:
    #         v.append(lib)

    # forma 2: si el archivo contenía los registros almacenados
    # uno a uno en forma secuencial...
    v = []
    t = os.path.getsize(fd)
    while m.tell() < t:
        lib = pickle.load(m)
        if lib.isbn > x:
            v.append(lib)

    m.close()
    print('Se creó el vector con parte del archivo', fd)
    print()
```

```

    return v

def test():
    v = []
    fd = 'libros.dat'
    op = -1
    while op != 8:
        print('Procesamiento combinado de arreglos, registros y archivos...')
        print('1. Crear el arreglo de libros (en forma automática)')
        print('2. Mostrar el arreglo de libros')
        print('3. Crear el archivo con TODOS los libros del arreglo')
        print('4. Mostrar el archivo de libros')
        print('5. Crear el archivo con ALGUNOS de los libros del arreglo')
        print('6. Volver a crear el arreglo con TODOS los libros del archivo')
        print('7. Volver a crear el arreglo con ALGUNOS de los libros del archivo')
        print('8. Salir')
        op = int(input('\t\tIngrese número de opción: '))
        print()

        if op == 1:
            cargar_arreglo(v)

        elif op == 2:
            mostrar_arreglo(v)

        elif op == 3:
            crear_archivo_todos(v, fd)

        elif op == 4:
            mostrar_archivo(fd)

        elif op == 5:
            crear_archivo_algunos(v, fd)

        elif op == 6:
            v = crear_arreglo_todos(fd)

        elif op == 7:
            v = crear_arreglo_algunos(fd)

        elif op == 8:
            pass

    if __name__ == '__main__':
        test()

```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 23

## Archivos: Gestión ABM

### 1.] Introducción.

Hemos indicado que un archivo puede contener datos del cualquier tipo que necesite el programador, pero que será muy común que se requiera almacenar *registros* de algún tipo, de forma que luego se procesen esos registros en base a alguna estrategia. Hemos visto que muchas veces se puede copiar algunos o todos los registros del archivo a un arreglo en memoria y trabajar luego con el arreglo (que será la forma que más comúnmente emplearemos en los ejercicios y problemas que quedan).

Pero en la práctica real lo más común es que dado un archivo de registros el programador necesite operar *directamente con el archivo*, sin pasar sus registros a un arreglo. En ese sentido, son básicamente tres las operaciones que permanentemente suelen aplicarse sobre su contenido: la *operación de agregar un registro al archivo* (operación que suele designarse como *alta de un registro*, o simplemente una *alta*), la operación de *eliminar un registro* (que se conoce como *baja de un registro* o simplemente una *baja*), y la *operación de modificar el contenido de un registro que ya existía* (operación que se conoce como *modificación de un registro*) [1].

En los programas o modelos de programas que hasta aquí hemos desarrollado para mostrar el manejo de archivos, hemos visto esquematizada de una forma u otra la operación de alta de un registro. Sin embargo, es común que en un sistema de gran envergadura y complejidad se incluyan subsistemas completamente dedicados a realizar el *soporte de mantenimiento* de cada archivo que el sistema usa. Cuando hablamos de un *subsistema de soporte de mantenimiento de un archivo*, nos referimos a un programa (o a una librería o a un módulo de un programa) que sea capaz de realizar las tres operaciones básicas que hemos citado sobre un archivo (*altas*, *bajas* y *modificaciones*), más alguna otra operación adicional que tenga utilidad, como por ejemplo la visualización por pantalla del contenido del archivo (en forma completa o en forma parcial), o la búsqueda de un registro particular, entre otras<sup>1</sup>. Un programa de esta clase suele designarse como *programa ABM*, siendo *ABM*

<sup>1</sup> El mantenimiento de un archivo o de una base de datos para un sistema informático crítico no es un tema para tomar a la ligera. La pérdida de esos datos, la modificación negligente de los mismos y/o la intervención malintencionada en los mecanismos de control del sistema puede tener consecuencias muy dañinas, como se reflejó en la película del 2007 *Live Free or Die Hard* (conocida aquí como *Duro de Matar 4.0*), dirigida por *Len Wiseman* y protagonizada por *Bruce Willis*. La película muestra el accionar de una banda de hackers ultra-tecnológicos contra todo el sistema informático de soporte de servicios públicos y financieros de los Estados Unidos, por medio de un ataque informático masivo designado como *fire sale* (o *venta de liquidación*). Un policía "chapado a la antigua" y un hacker "obligado a hacer de bueno" intentan desactivar el ataque, y la película estalla (literalmente...) en un impresionante "crescendo" de escenas de acción. Y todo por un "hacker malo"...

una abreviatura formada por las primeras letras de las palabras *altas*, *bajas* y *modificaciones*.

El hecho es que a la hora de desarrollar un **programa ABM** surgen algunos inconvenientes conceptuales que conviene estudiar en detalle para no cometer errores involuntarios y para no caer en pérdidas de eficiencia en los planteos que se realicen. A continuación, hacemos un breve análisis de cuales son esos problemas y las formas de resolverlos. En toda la explicación que sigue, supondremos que nuestro objetivo es diseñar un **programa ABM** para un archivo que contenga registros con datos de *estudiantes* de una carrera universitaria, con campos para el número de *legajo*, el *nombre* y el *promedio* de cada estudiante. Los detalles de código fuente irán apareciendo a medida que se vayan dejando en claro algunos aspectos prácticos.

## 2.] Bajas en un archivo de registros.

Básicamente, los problemas comienzan cuando se quiere plantear la forma de hacer una *baja de un registro*. La idea es que el programa debe pedir que se cargue por teclado el valor de la *clave de identificación del registro*<sup>2</sup> que se quiere borrar (en nuestro caso, si suponemos que en el archivo almacenaremos registros con datos de estudiantes, podemos tomar como clave al campo *legajo*). Luego, el programa debe buscar un registro con esa clave en el archivo, y en caso de encontrarlo, proceder a eliminar el registro completo.

Hasta aquí, todo parece fácil... pero el hecho es que el registro está grabado en el archivo, y no existe una instrucción en Python que permita eliminar un registro del archivo en forma similar a lo que hace la instrucción *del* para eliminar un componente de un arreglo representado en una variable de tipo *list*. Frente a esto, hay dos vías de solución clásicas [1]:

- a.) La primera es usar un *segundo archivo* (designado como *archivo temporal*), inicialmente vacío, y proceder de la forma siguiente: El archivo original se abre en modo de lectura ('rb'), y el archivo temporal en modo de grabación pero de forma que si el archivo existía se elimine su contenido ('wb'). Con un ciclo se recorre y se lee (con *pickle.load()*) hasta el final el archivo original. En cada vuelta del ciclo se toma el registro leído desde archivo original y se verifica si el campo clave (el legajo en nuestro caso) coincide con el valor buscado. Si el valor *no* coincide, entonces simplemente se toma ese registro y se lo graba (con *pickle.dump()*) en el archivo *temporal*. El único registro que *no se copia* al temporal, es aquel cuyo campo clave coincide con el valor buscado.

Al finalizar el ciclo, el archivo temporal será una copia exacta del archivo *original*, salvo que no contendrá al registro que se quería borrar. Entonces lo único que queda por hacer es *destruir el archivo original* (borrarlo del *sistema de archivos* o *file system* del disco), y cambiar el nombre del archivo temporal para que ahora tome el nombre que tenía el original.

Este proceso completo efectivamente permite eliminar de un archivo a un registro, aunque para ello trabaja transitoriamente con un archivo temporal que se usa a modo de réplica del original. En general, cualquier operación que permita eliminar un registro de un archivo de forma que el registro no ocupe más lugar dentro del archivo, se conoce como una *baja física* de registro.

<sup>2</sup> En general, se entiende como *clave* de un registro al campo (o conjunto de campos) cuyos valores permiten identificar en forma única a un registro en el archivo. El número de legajo suele ser una clave válida en la Universidad, puesto que para cada alumno el número de legajo es único. Normalmente, son buenos candidatos para claves de registro los campos cuyos valores no pueden (o no deben...) repetirse, pero en algunas ocasiones se usan claves que admitan valores repetidos.

En general, el proceso de **baja física** hace que el registro realmente sea eliminado, disminuyendo el tamaño del archivo original, pero como contrapartida puede verse claramente que el mecanismo completo insume mucho tiempo, ya que todos los registros del archivo original deben ser leídos y grabados uno por uno en el archivo temporal (salvo el que se quiere eliminar). En archivos pequeños este tiempo puede ser despreciable, pero en situaciones reales con archivos muy grandes, la demora será importante.

- b.) La otra forma de hacer una baja consiste en no tratar de eliminar el registro físicamente, pues como se indicó esto lleva mucho tiempo. Lo que se puede hacer es una variante conocida como **baja lógica** (o **marcado lógico**) del registro: en vez de eliminarlo físicamente, simplemente se lo marca usando un campo especialmente definido a modo de *bandera*: si este campo vale *False*, significa que el registro está *marcado como eliminado* y por lo tanto no debe tenerse en cuenta en ninguna operación que se realice sobre el archivo, pero si vale *True*, entonces el registro está *activo* y por lo tanto debe ser procesado cuando se recorra el archivo.

Cuando un registro nuevo se graba en el archivo, se hace de forma que ese campo de marcado lógico valga *True*, y sólo se cambia ese valor cuando se desea eliminar el registro. De esta forma, el proceso de **baja lógica** consiste en buscar el registro con la clave dada usando un ciclo, leerlo con `pickle.load()`, cambiar el campo de marcado lógico a *False*, y volver a grabar el registro en el mismo lugar original usando `pickle.dump()`.

Ahora bien: si se aplica siempre el proceso de baja lógica, puede ocurrir que en algún momento el archivo quede *disperso*: por más que los registros se marcan como borrados, siguen ocupando lugar físico dentro del archivo. Con el tiempo, los registros marcados pero no borrados implican un tiempo de procesamiento extra que equilibra el tiempo que se pudiera haber ganado con las bajas lógicas en lugar de las físicas, además del uso innecesario del espacio de disco para registros que no se usan.

En la práctica, y para plantear una solución de equilibrio entre las ventajas de ambas estrategias, se suele aplicar un *esquema combinado*: cuando se requiere hacer bajas en el momento (por ejemplo, en casos de atención frente al público en tiempo real), se aplica un proceso de **baja lógica** para no perder tiempo. Y luego, en algún momento no crítico de ciclo de uso del archivo (por ejemplo, a la hora del cierre o de la apertura de la organización que usa el sistema, si es que en algún momento el mismo se cierra), el encargado del mantenimiento del sistema efectúa *un* proceso de **bajas físicas**, pero eliminando de una sola pasada todos los registros que quedaron marcados. Este último proceso suele llamarse *depuración* del archivo (o también *compactación* u *optimización del espacio físico* del archivo).

Cabe aclarar que el proceso de **baja lógica**, planteado en la forma que aquí se hizo, requiere recorrer el archivo hasta encontrar el registro buscado. En el peor caso, si el registro se encuentra muy al final, eso supondría que el archivo debería ser leído casi completamente, y por ello se perdería básicamente el mismo tiempo que con una baja física (en la **baja física** se agrega el tiempo extra de grabar el registro en el archivo secundario). Si el proceso de **baja lógica** se hace en base a un recorrido secuencial del archivo, entonces en promedio no hay diferencia significativa de tiempo con respecto a la **baja física**.

Entonces, ¿por qué insistir? El hecho es que la técnica para realizar **bajas lógicas** realmente permite ganar tiempo si el archivo está organizado de forma que permita accesos rápidos a sus registros individuales. Esto puede lograrse con métodos de organización muy conocidos, pero que escapan al alcance de este curso, como la *organización de claves para acceso*

directo, la *indexación*, o la *búsqueda por dispersión* (llamada más comúnmente *hashing*). En todo caso, lo que aquí se pretende es mostrar al estudiante la forma en que puede hacerse una *baja lógica*, y el mismo estudiante podrá oportunamente adaptar la técnica para situaciones más provechosas [1].

El programa *abm.py* que viene incluido en el proyecto [F23] Gestión ABM anexo a esta Ficha, aplica estas ideas sobre un archivo con registros de estudiantes. El planteo y análisis detallado del programa se deja para la sección de *Temas Avanzados* de esta misma Ficha.

### 3.] Altas en un archivo de registros.

Una vez resuelto (o al menos, diseñado) el tema de las *bajas* (usando un campo de marcado lógico y/o combinando con un proceso de eliminación física posterior), entonces el proceso de *alta* resulta simple. Si se asume que en el archivo no se permitirán registros con claves repetidas (lo cual es muy común), entonces los pasos a seguir son los siguientes: se carga por teclado el valor del campo clave del registro a dar de alta. Usando un ciclo se recorre el archivo y en cada iteración se lee el registro actual usando *pickle.load()*. Con cada registro así leído desde el archivo, se compara su clave con el valor cargado por teclado.

Si se encuentra algún registro cuya clave coincide con este último, *pero que además no esté marcado como eliminado*, entonces la operación de *alta* se rechaza (pues ya existe un registro activo con esa clave). Si se llega al final del archivo y no se encontró repetida la clave, entonces se terminan de cargar por teclado los campos del registro a dar de alta, se asigna en *True* su campo de marcado lógico, y se graba el registro *al final* del archivo usando *pickle.dump()* (típicamente, el archivo debería estar abierto en modo 'a+b': es necesario poder leerlo para buscar el registro y saber si estaba repetido o no, y es necesario poder grabar el nuevo registro si fuese el caso, pero agregándolo al final).

Si en el archivo se admiten registros con claves repetidas, entonces el proceso de *alta* es mucho más simple pues ahora no es necesaria la fase de búsqueda para saber si el registro está repetido. Simplemente se abre el archivo en modo 'ab', se cargan por teclado los datos del nuevo registro, se pone su campo de marcado lógico en *True*, y se graba el registro sin más trámite (se grabará al final, porque el archivo se abrió en modo 'ab'). En el citado modelo *abm.py* que viene en el proyecto anexo a esta Ficha, se supone que las repeticiones no se permiten, y se deja como tarea para el alumno el adaptar ese programa para aceptar claves repetidas [1].

### 4.] Modificaciones y listados en un archivo de registros.

Otra tarea muy común es la de pedir la *modificación* de uno o más campos de un registro específico dentro del archivo. Para esto, el proceso es similar al de la *baja lógica*: se carga por teclado el valor de la clave del registro que se quiere modificar. Se busca dicho registro en el archivo usando un ciclo y leyendo y comparando uno por uno. Si se encuentra un registro cuya clave coincide con la buscada y tenga en *True* en el campo de marcado lógico, se muestran los campos del mismo por pantalla, y con un pequeño menú de opciones se pide al operador que elija los campos que quiere modificar, cargando por teclado los valores nuevos.

Como esta modificación se está haciendo en memoria principal, todavía queda *volver a grabar el registro en el disco pero en su posición original*, para lo cual se usa `seek()` para volver atrás el *file pointer*, y se vuelve a grabar el registro con `pickle.dump()`. Decimos que este proceso es similar al de una *baja lógica*, porque en ambos se busca el registro, se lo sube a memoria, se modifica su contenido, y se vuelve a grabar en la posición original. La diferencia es que en la *baja* se cambia sólo el valor del campo de marcado lógico, y en la *modificación* se cambia el resto de los campos (y no el campo de marcado lógico) [1].

Si bien el proceso general es el que acabamos de describir, en la práctica surge un inconveniente adicional debido al hecho de usar serialización para la grabación de registros. Cuando un registro se graba mediante `pickle.dump()` en la operación de alta, la función `pickle.dump()` transforma el registro en una secuencia de bytes de cierto tamaño  $k$ , y graba esa secuencia en el archivo. Pero en Python una variable ocupará tantos bytes como sea necesario para poder representar en binario el valor asignado en ella. Un número entero pequeño (por caso, el 10 o el 28) ocupará menos bytes que un número más grande (por caso, el 50000). Y una cadena de caracteres ocupará más bytes mientras más caracteres tenga.

Todo lo anterior quiere decir que dos registros del mismo tipo (por ejemplo, dos registros de tipo *Estudiante*) *no necesariamente ocuparán la misma cantidad de bytes*. Esto no es un problema en memoria principal, pero sí implica algunos inconvenientes cuando esos registros se graban en un archivo y se pretende aplicar un *proceso ABM* en ese archivo.

¿Cuál es concretamente el potencial problema? Suponga que un registro  $r_1$  fue grabado en cierta posición  $p_1$  del archivo cuando fue dado de alta, y también suponga que ese registro ocupaba una cantidad  $k_1$  de bytes. Por lo tanto, el siguiente registro  $r_2$  comenzará en la posición  $p_2 = p_1 + k_1$ . Suponga ahora que se modifica el contenido del registro  $r_1$ , pero al hacerlo suponga también que los nuevos valores de sus campos *aumentan* el tamaño del registro  $r_1$  al valor  $k_2$  ( $> k_1$ ). Cuando el registro  $r_1$  se vuelva a grabar en su posición original  $p_1$ , en lugar de ocupar  $k_1$  bytes ocupará  $k_2$  bytes, y como  $k_2 > k_1$ , entonces el registro  $r_1$  ocupará bytes que antes le pertenecían en el archivo al registro  $r_2$  (es decir, el registro  $r_1$  ahora ocupa bytes que están más allá de la posición  $p_2 = p_1 + k_1$ ).

La consecuencia de esto es que se perderán bytes del registro  $r_2$ , y se perderá también la sincronización en el proceso de lectura secuencial cuando se aplique `pickle.load()`: Cuando se intente leer el registro  $r_2$ , el *file pointer* no estará ubicado en  $p_2 = p_1 + k_1$  sino en la posición  $p_3 = p_1 + k_2$ , y desde esa posición `pickle.load()` no podrá ya recuperar el registro original. El proceso completo terminará con un *error de runtime* cuando `pickle.load()` intente seguir leyendo y se encuentre de forma inesperada con el final del archivo.

Hay varias maneras de evitar estos inconvenientes. La forma más tradicional consiste en *forzar a cada registro a ocupar el mismo espacio k en bytes* cuando se grabe en el archivo, eliminando de esta forma todo problema (ya que todos los bloques que se graben, ahora serán del mismo tamaño  $k$ ). En nuestro caso, y dado que Python modifica el tamaño de cada variable en función del valor que toma, aplicar esta solución no es tan simple ni tan directo y por lo tanto se deben formular algunas restricciones.

Por lo pronto, si un campo de un registro es de tipo cadena de caracteres haremos que esa cadena se *ajuste a una cantidad de caracteres prefijada por el programador*, llenándola con espacios en blanco a la derecha si fuese necesario. Por ejemplo, si el registro  $r$  tiene un campo llamado *nombre* asignado con la cadena 'Luis' en ese campo, y cuando se quiera

grabar el registro se decide que el *nombre* ocupe siempre 30 caracteres, se podría hacer en la forma siguiente:

```
r.nombre = 'Luis'  
r.nombre = r.nombre.ljust(30)
```

El pequeño script anterior usa el método *ljust()* contenido en cualquier variable de tipo cadena de caracteres, para justificar hacia la izquierda el contenido de la cadena forzándola a tener tantos caracteres como indica el número tomado como parámetro. Si ese parámetro vale 30, se agregarán al final de la cadena tantos espacios en blanco como hagan falta para completar los 30 caracteres pedidos. Por lo tanto, si la cadena asignada era 'Luis', se agregarán 26 espacios en blanco a la derecha (la cadena quedará con 30 caracteres de largo, justificada hacia la izquierda).

Si el método *ljust()* se invoca pasando sólo un parámetro numérico (como en el ejemplo), entonces se asume que el relleno debe hacerse con *espacios en blanco*. Opcionalmente, se puede enviar un segundo parámetro indicando cuál es el carácter de relleno que debería usarse. Lo anterior, en este caso, sería lo mismo que:

```
r.nombre = 'Luis'  
r.nombre = r.nombre.ljust(30, ' ')
```

El método *ljust()* ajusta el contenido de la variable para llegar a la cantidad de caracteres pedida, y finalmente retorna la nueva cadena.

Si el registro contiene campos numéricos de tipo entero, Python asignará a ese campo tantos bytes como necesite para representar en binario ese valor. Por lo tanto, otra vez, podríamos tener problemas con el cambio de tamaño de un registro si el mismo estaba grabado en el archivo y luego se modifica el valor de un campo de tipo entero (ese campo podría ahora ocupar más bytes que en el registro original). De nuevo, hay muchas formas de solucionar este problema, y el programador deberá esforzarse en cada caso para aplicar la mejor solución. Una estrategia es directamente manejar ese campo como una cadena de caracteres de tamaño fijo que sólo contenga dígitos, y cuando se necesite usar el valor en forma de número, convertirlo a número por medio de la función *int()*. Otra, es validar el valor de ese campo para asegurar que siempre tenga un valor en un intervalo conocido, y asegurarse que todos los números en ese intervalo tengan el mismo tamaño en bytes.

En general, la operación de modificación del contenido de un registro se hace sobre los campos que *no son la clave del registro o no forman parte de la clave del registro*. Si se desea modificar el campo clave, debe hacerse primero la baja del registro, y luego volver a darlo de alta con la nueva clave. En el ABM que estamos planteando para el archivo de estudiantes, el campo clave es el *legajo*, que es de tipo entero. Pero como no permitiremos la modificación de este campo, los problemas potenciales por el cambio de tamaño de ese campo no se producirán.

Finalmente, si el registro contiene algún campo numérico de tipo *float* entonces el programador no deberá preocuparse: los valores de tipo *float* se representan siempre con *doble precisión* (o sea, 8 bytes por número), y cuando un *float* es serializado mediante *pickle.dump()* su representación se expande a 12 bytes por número en el archivo. En nuestro caso, el registro que representa a un estudiante contiene el campo *promedio*, de tipo *float*, que por lo ya indicado no será un problema en cuanto a posible cambio de tamaño.

La operación general de **modificación** es la última de las tres más básicas que se esperan en un *programa ABM*. Sin embargo, como dijimos, este tipo de programas puede incluir muchas otras opciones de gestión, la mayoría en forma de tareas de búsqueda y/o de listados de contenido.

Un proceso de **listado completo** es aquel que simplemente muestra el contenido completo del archivo, y se plantea en forma simple: el archivo se abre en modo de solo lectura (rb), con un ciclo se leen uno por uno sus registros y se muestran en pantalla a medida que se leen. Obviamente, los registros que se muestran son sólo aquellos que en el momento de ser leídos no estén marcados como eliminados. Y por otra parte, un **listado parcial** (también conocido como **listado con filtro**) es aquel en el cual se recorre y se lee todo el archivo, pero sólo se muestran en pantalla los registros que cumplen con cierta condición (por ejemplo, mostrar sólo los registros de los alumnos cuyo promedio sea mayor o igual a 7). Como las necesidades de consulta de un archivo suelen ser muchas y todas ellas con distintos criterios de filtrado, el menú de un *programa ABM* suele tener también muchísimas opciones y variantes. En el modelo que acompaña a esta Ficha hemos incluido los procesos esenciales (altas, bajas lógicas, modificaciones, depuración) y un par de opciones de listado de contenido (una para un listado completo, y otra para un listado con filtro). Vea la sección siguiente de esta Ficha.

## 5.] Desarrollo de un programa completo de gestión ABM.

Se propone ahora el estudio detallado de la estructura de un programa completo de *gestión ABM* que aplique los conceptos desarrollados en forma conceptual en la primera parte de esta Ficha. Formalmente, lo enunciamos como un problema a modo de caso de análisis:

**Problema 54.)** *Desarrollar un programa controlado por menú de opciones, que permita realizar en forma completa la gestión ABM de un archivo de registros de estudiantes de una carrera universitaria. Por cada estudiante, prevea tres campos para el legajo, el nombre y el promedio (además del campo de marcado lógico para las bajas lógicas). El programa debe incluir opciones que permitan:*

- a.) *Realizar altas de registros en el archivo.*
- b.) *Realizar bajas lógicas.*
- c.) *Realizar la modificación de los datos de un registro.*
- d.) *Mostrar el contenido completo del archivo.*
- e.) *Mostrar los datos de los estudiantes con promedio mayor o igual a 7.*
- f.) *Realizar la depuración del archivo (proceso de bajas físicas).*

**Discusión y solución:** El proyecto [F23] Gestión ABM que acompaña a esta Ficha contiene un modelo *abm.py* con el programa completo que resuelve este caso de análisis.

El programa *abm.py* comienza con la ya tradicional importación de módulos y la definición del tipo de registro *Estudiante*, más un par de funciones *\_\_init\_\_()* y *to\_string()* para manejar estos registros y dos funciones de validación de carga para los campos *legajo* y *promedio*.

Note que la función constructora *\_\_init\_\_()* agrega e inicializa los campos en un registro pero dentro de ese conjunto de campos ya incluye el campo llamado **activo** de tipo *boolean*, al cual inicializa directamente con el valor *True* cuando un registro es creado (sin tomar su

valor como parámetro). Ese campo *activo* es el que usaremos como *campo de marcado* para las *bajas lógicas* [1]:

```
import io
import os
import pickle
import os.path

__author__ = 'Cátedra de AED'

class Estudiante:
    def __init__(self, leg, nom, prom):
        self.legajo = leg
        self.nombre = nom
        self.promedio = prom
        self.activo = True

    def to_string(self):
        r = ''
        r += '{:<20}'.format('Legajo: ' + str(self.legajo))
        r += '{:<30}'.format('Nombre: ' + self.nombre.strip())
        r += '{:<20}'.format('Promedio: ' + str(self.promedio))
        return r

def validar_legajo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Entre '+str(inf)+' y '+str(sup)+' por favor: '))
        if n < inf or n > sup:
            print('Se pidió entre', inf, 'y', sup, '... cargue de nuevo...')
    return n

def validar_promedio():
    n = -1.0
    while n < 0.0 or n > 10.0:
        n = float(input('Entre 0 y 10 por favor (acepta decimales): '))
        if n < 0 or n > 10:
            print('Error... se pidió entre 0 y 10... cargue de nuevo...')
    return n
```

Note también que la función *to\_string()* convierte a cadena de caracteres a todos los campos de un registro, salvo el valor del campo *activo*, que es un campo de uso interno y no es necesario que se muestre luego al usuario final. Además, antes de concatenar a la cadena de salida el valor del campo *nombre*, la función usa el método *strip()* que elimina los espacios en blanco que ese *nombre* pudiese tener al principio o al final (recuerde que para asegurar que el tamaño del registro se mantenga constante, al hacer la alta se ajustará la cantidad de caracteres del campo *nombre* para agregar espacios en blanco y llegar a 30 caracteres).

Lo siguiente en el programa, y para facilitar los procesos, es el desarrollo de una función que permite *buscar* un registro con una clave dada dentro del archivo, y retorne la dirección de ese registro si lo encuentra (por dirección del registro entendemos el número del byte dentro del archivo en el cual comienza el registro encontrado). Esa función se llama *buscar()* y es usada en las funciones de *altas*, *bajas lógicas* y *modificaciones*. Su estructura se muestra

a continuación (el nombre del archivo es *estudiantes.utn* y se supone que ese nombre está almacenado en una variable ***global FD*** (creada y definida en la función *main()* que contiene el menú de opciones y el lanzamiento del programa)):

```
def buscar(m, leg):
    global FD
    t = os.path.getsize(FD)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        est = pickle.load(m)
        if est.activo and est.legajo == leg:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
return posicion
```

La función toma dos parámetros: el primero (*m*) es la variable *file object* que representa al archivo que se está gestionando. La función supone que ese archivo ya viene abierto, y además supone que viene abierto en un modo que permite lecturas. Si estos supuestos no se cumplen, *la función provocará un error de runtime y el programa se interrumpirá*.

El segundo parámetro (*leg*) es un número entero con el número de legajo que se quiere buscar en el archivo. La función buscará un registro con ese legajo. En caso de encontrarlo, detendrá la búsqueda y retornará el número del byte interno del archivo en el cual comienza el registro que se acaba de encontrar (por lo tanto, retornará en este caso un número mayor o igual a cero). Si el archivo no contiene ningún registro cuyo legajo coincida con *leg*, la función retornará el valor -1 (el cual entonces debe ser chequeado y entendido como un flag avisando que la búsqueda no tuvo éxito). La variable local *posicion* se usa para almacenar el valor a retornar: comienza valiendo -1 y si luego el registro es encontrado, su valor cambia para contener el número de su byte de inicio.

Como la función necesita leer el archivo desde el inicio (pues de lo contrario podrían quedar registros sin revisar), se usa el método *seek()* para llevar el file pointer al inicio del archivo (el byte número cero). Como se vio en la ficha anterior, esto puede hacerse así [2] [3]:

```
m.seek(0, io.SEEK_SET)
```

Luego de esto comienza el ciclo de lectura del archivo. Se aplica el ya conocido mecanismo de usar un ciclo *while* y controlar que el valor actual del *file pointer* sea menor que el tamaño en bytes del archivo. En cada iteración del ciclo, se lee el registro actual con *pickle.load()*, pero teniendo la precaución de almacenar previamente en la variable local *fp* el valor que en ese momento (antes de la lectura) tenía el *file pointer*. De esta forma, si el registro que luego se lee con *pickle.load()* fuese efectivamente el registro que se estaba buscando, la variable *fp* contendrá la dirección o número del byte donde ese registro comenzaba, y sólo deberá copiarse su valor en la variable *posicion* para retornarlo al final (recuerde que cada vez que se lee o graba en un archivo, el valor del *file pointer* se actualiza para quedar apuntando al byte inmediatamente siguiente a aquel en el cual terminó la lectura o la grabación, por lo

cual el valor retornado por `tell()` después de leer no será la dirección del registro leído, sino la del siguiente...) [1].

Un detalle final: como la función recibe el archivo ya abierto, el *file pointer* del mismo está ya posicionado en algún byte particular. Y es de esperar que el programador que haya invocado a la función `buscar()` espere que el archivo no sólo vuelva abierto, sino que además vuelva con el *file pointer* apuntando al mismo lugar donde el mismo estaba antes de invocar a la función (aunque en este caso el contexto no exige que se cumple ese requisito). Por ese motivo, antes de volver el *file pointer* al inicio del archivo y comenzar el ciclo de lectura, se invoca a `tell()` y se almacena la posición actual del *file pointer* en la variable `fp_inicial`. Cuando el ciclo termina, y antes de retornar el resultado final, se vuelve a usar `seek()` para llevar el *file pointer* a la posición indicada por `fp_inicial`, dejando el archivo tal como estaba antes de comenzar la búsqueda.

La función `alta()` es la encargada de agregar nuevos registros al archivo. Su estructura es la que sigue:

```
def alta():
    global FD
    m = open(FD, 'a+b')

    print()
    print('Legajo del estudiante a registrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            # ...ajustar a 30 caracteres, llenando con blancos al final...
            nom = nom.ljust(30, ' ')

        print('Promedio...')
        pro = validar_promedio()

        est = Estudiante(leg, nom, pro)

        # ...lo grabamos...
        pickle.dump(est, m)

        # ...volcamos el buffer de escritura
        # para que el sistema operativo REALMENTE
        # grabe en disco el registro...
        m.flush()

        print('Registro grabado en el archivo...')

    else:
        print('Legajo repetido... alta rechazada...')

    print()
    print('Otro legajo a registrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)

m.close()
```

```

print()
print('Operación de altas finalizada...')
input('Presione <Enter> para seguir...')


```

La idea es la que ya hemos explicado en la sección 3 de esta ficha. El archivo se abre en modo 'a+b', de forma que si el mismo no existía, será creado, y en caso de existir, su contenido será preservado. Cuando se pida grabar (con *pickle.dump()*) el nuevo registro será agregado al final.

La función usa un ciclo de carga por doble lectura para leer por teclado un legajo (controlando que sea diferente de cero). Si es así, se usa la función *buscar()* ya explicada, para determinar si existe o no un registro con ese legajo. Si ya existiese un registro con ese legajo, la operación de alta se rechaza (no admitiremos alumnos con registros duplicados) y se carga otro legajo. Sólo si el legajo no estuviese repetido, se carga el resto de los campos y se graba con *pickle.dump()* el nuevo registro. Recuerde que la función *init()* asigna el valor *True* en el campo activo de marcado lógico.

Al cargar por teclado el *nombre* del estudiante, se usa el ya citado método *ljust()* para agregar espacios en blanco al final del campo *nombre*, ajustar el tamaño del campo a 30 caracteres, y evitar así futuros problemas de cambio de tamaño cuando se modifique ese registro.

El único detalle extraño es que luego de invocar a *pickle.dump()* para grabar el registro, se está invocando a otro método *flush()* que hasta aquí no habíamos utilizado. El motivo: cuando un archivo se abre para grabación, el verdadero responsable de grabar los nuevos datos en el mismo es el *sistema operativo* (y no el lenguaje de alto nivel que se esté usando). Pero por razones de eficiencia y de mejor aprovechamiento del tiempo de trabajo del procesador, el sistema operativo utiliza una zona de memoria intermedia (o *buffer*) en la cual va guardando los datos que el programa solicita grabar. Sólo cuando ese buffer está lleno, el sistema operativo va al disco y efectivamente graba los datos en el archivo. Mientras tanto, esos datos no están realmente grabados... por lo que en este caso, cuando la función *buscar()* es invocada para determinar si existe o no un registro con legajo repetido podría llegar a informar que un registro que *se acaba de agregar* no existe, simplemente porque sus datos aún no fueron grabados en el archivo.

Cuando se invoca al método *close()*, todos los datos que el sistema operativo tenía en el buffer del archivo se vuelcan al mismo. Pero en el caso de la función *altas()*, el archivo no se cierra hasta terminar el ciclo de carga por doble lectura (se graban varios registros antes de cerrar el archivo). Para no tener que cerrar y volver a abrir el archivo cada vez que se graba un registro, se usa el método *flush()*: este método simplemente vuelca el contenido del buffer del archivo y lo graba en el mismo en forma efectiva, sin tener que esperar a que ese buffer se llene o que se invoque a *close()* [2].

Note finalmente que el problema que acabamos de describir sólo se da cuando el archivo se abre en *modo de grabación* (de hecho, *flush()* no tiene ningún efecto si el archivo está abierto en modo de sólo lectura).

La función *baja()* lleva a cabo el proceso de *baja lógica* (eliminación por marcado lógico de un registro):

```

def baja():
    global FD

```

```

if not os.path.exists(FD):
    print('El archivo', FD, 'no existe...')
    print()
    return

m = open(FD, 'r+b')

print()
print('Legajo del estudiante a borrar (cargue 0 para salir): ')
leg = validar_legajo(0, 99999)
while leg != 0:
    # buscamos el registro con ese legajo...
    pos = buscar(m, leg)
    if pos != -1:
        # encontrado... procedemos a cargarlo...
        m.seek(pos, io.SEEK_SET)
        est = pickle.load(m)

        # ...mostramos el registro tal como estaba...
        print()
        print('El registro actualmente grabado es:')
        print(to_string(est))

        # ...chequemos si el usuario está seguro de lo que hace...
        r = input('Está seguro de querer borrarlo (s/n)? ')
        if r in ['s', 'S']:
            # lo marcamos como borrado, y ya...
            est.activo = False

            # ...reubicamos el file pointer...
            m.seek(pos, io.SEEK_SET)

            # ...y lo volvemos a grabar...
            pickle.dump(est, m)

            print()
            print('Registro eliminado del archivo...')

    else:
        print('Ese registro no existe en el archivo...')

    print()
    print('Otro legajo de estudiante a borrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)

m.close()

print()
print('Operación de bajas finalizada...')
input('Presione <Enter> para seguir...')

```

Esta función también usa un ciclo de carga por lectura doble para poder hacer varias operaciones de baja de registros si fuese necesario. Se carga un legajo, y se invoca a la función *buscar()* otra vez. Si no existe un registro con ese legajo, la baja no puede hacerse y se pide un nuevo legajo.

Si el legajo existe, se usa *seek()* para posicionar el *file pointer* en el byte donde comienza ese registro (valor que fue retornado por *buscar()* en ese caso), y se lee el registro con *pickle.load()*. Antes de eliminarlo, se muestra el registro en pantalla y se pregunta al usuario

si está seguro de querer eliminarlo. Si el usuario confirma la operación, simplemente se cambia el valor del campo *activo* por el valor *False*, y se vuelve a grabar el registro en la misma posición original.

Sólo hay que recordar que cuando el registro fue leído, el *file pointer* se movió al registro siguiente, por lo que antes de volver a grabarlo se debe hacer retroceder el *file pointer* a la posición donde el registro comienza: esto requiere una segunda invocación a *seek()* (a la misma posición retornada por *buscar()*) y recién entonces invocar a *pickle.dump()* para volver a grabar el registro [1].

La función *modificacion()* es la que permite cambiar el valor de alguno de los campos de un registro. Es muy similar a la función *baja()*, pero implica algo más de trabajo en la interfaz de usuario (incluyendo *el ajuste a 30 caracteres del campo nombre* si este fuese vuelto a leer desde el teclado):

```
def modificacion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a modificar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(to_string(est))

        # ...modificamos el valor de los campos...
        op = 0
        while op != 3:
            print('1. Modificar nombre.')
            print('2. Modificar promedio.')
            print('3. Terminar modificaciones.')
            op = int(input('\t\tIngrese opción: '))

            if op == 1:
                nom = input('Nuevo nombre: ')
                est.nombre = nom.ljust(30, ' ')

            elif op == 2:
                print('Nuevo promedio:')
                est.promedio = validar_promedio()

            elif op == 3:
                pass
```

```

# ...registro modificado en memoria...
# ...ahora nos volvemos a su posición en el archivo...
m.seek(pos, io.SEEK_SET)

# ...y volvemos a grabar el registro modificado...
pickle.dump(est, m)

print()
print('Los datos del registro se actualizaron...')

else:
    print('Ese registro no existe en el archivo...')

print('Otro legajo a modificar (cargue 0 para salir): ')
leg = validar_legajo(0, 99999)

m.close()

print()
print('Operación de modificaciones finalizada...')
input('Presione <Enter> para seguir...')


```

Esta función trabaja en la misma forma que la función *baja()*. La diferencia es que ahora, cuando se muestra el registro encontrado, en lugar de preguntar al usuario si está seguro de querer borrarlo se muestra un pequeño menú de opciones en el cual se le da a elegir al operador el campo cuyo valor quiere cambiar. Cuando esos cambios han sido realizados, el registro se vuelve a grabar en la misma posición en la que fue leído, teniendo en cuenta los mismos detalles ya señalados en cuanto al *file pointer* para la función *baja()* (aunque en este caso, obviamente, el valor del campo de marcado lógico se mantiene en *True*) [1].

La función encargada de depurar el archivo y proceder a eliminar físicamente los registros marcados como eliminados, es la función *depuracion()*:

```

def depuracion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)

    original = open(FD, 'rb')
    temporal = open('temporal.dat', 'wb')

    print('Procediendo a optimizar el archivo', FD)
    input('Presione <Enter> para seguir...')

    while original.tell() < tbm:
        # cargar un registro del archivo original...
        est = pickle.load(original)

        # ...si no estaba marcado, grabarlo en el archivo temporal...
        if est.activo:
            pickle.dump(est, temporal)

    # cerrar ambos archivos...
    original.close()
    temporal.close()


```

```

# eliminar el archivo original...
os.remove(FD)

# y renombrar el temporal...
os.rename('temporal.dat', FD)

print('Optimización terminada... se eliminaron los registros marcados')
input('Presione <Enter> para seguir...')

```

El archivo *original* se abre en modo de sólo lectura ('rb'), y el *temporal* en modo de creación y eliminación de contenido anterior ('wb'). Un ciclo *while* lee uno por uno los registros del *original*, y a cada uno cuyo campo *activo* valga *True* lo graba a su vez en el *temporal*.

Cuando el ciclo termina, ambos archivos se cierran. Se procede entonces a eliminar el archivo *original*, con la función *os.remove()*. Esta toma como parámetro el *nombre físico del archivo* (y *no* la variable *file object* para gestión del archivo) que se quiere eliminar y simplemente procede a removerlo del sistema de archivos del disco. La función *os.remove()* no se limita a vaciar el contenido del archivo (cosa que se puede hacer abriendo el mismo en modo 'wb'), sino que lo elimina efectivamente de su carpeta contenedora [2].

Una vez eliminado el *original*, se debe renombrar el *temporal* para que asuma el nombre físico que tenía el *original*. Esto se hace con la función *os.rename()*, la cual toma dos parámetros: el *viejo nombre físico* del archivo a renombrar, y el *nuevo nombre físico* que se quiere dar al archivo. La función cambia el *viejo nombre* del archivo en el sistema de archivos, y lo reemplaza por el *nuevo nombre*. Note que tanto la función *os.remove()* como la función *os.rename()* requieren que los archivos con los que se va a trabajar estén *cerrados* (no se puede eliminar o renombrar un archivo que esté en uso) y ambas pertenecen al *módulo os* que fue convenientemente importado al inicio del programa [2].

Las funciones encargadas de mostrar el contenido del archivo (en forma completa o en forma filtrada) son directas y sencillas, por lo que dejamos su análisis para el estudiante. Lo mismo vale para la función *main()* que contiene el menú de opciones y es el punto de entrada del programa. El programa completo se muestra a continuación:

```

import io
import os
import pickle
import os.path

__author__ = 'Cátedra de AED'

class Estudiante:
    def __init__(self, leg, nom, prom):
        self.legajo = leg
        self.nombre = nom
        self.promedio = prom
        self.activo = True

def to_string(est):
    r = ''
    r += '{:<20}'.format('Legajo: ' + str(est.legajo))
    r += '{:<30}'.format('Nombre: ' + est.nombre.strip())
    r += '{:<20}'.format('Precio: ' + str(est.promedio))
    return r

```

```

def validar_legajo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Entre '+str(inf)+' y '+str(sup)+' por favor: '))
        if n < inf or n > sup:
            print('Se pidió entre',inf,'y',sup,'... cargue de nuevo... ')
    return n

def validar_promedio():
    n = -1.0
    while n < 0.0 or n > 10.0:
        n = float(input('Entre 0 y 10 por favor (acepta decimales): '))
        if n < 0 or n > 10:
            print('Error... se pidió entre 0 y 10... cargue de nuevo... ')
    return n

def buscar(m, leg):
    global FD
    t = os.path.getsize(FD)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        est = pickle.load(m)
        if est.activo and est.legajo == leg:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion

def alta():
    global FD
    m = open(FD, 'a+b')

    print()
    print('Legajo del estudiante a registrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            # ...ajustamos a 30 caracteres, rellenando con blancos...
            nom = nom.ljust(30, ' ')

            print('Promedio...')
            pro = validar_promedio()

            est = Estudiante(leg, nom, pro)

            # ...lo grabamos...

```

```

pickle.dump(est, m)

# ...volcamos el buffer de escritura
# para que el sistema operativo REALMENTE
# grabe en disco el registro...
m.flush()

print('Registro grabado en el archivo...')

else:
    print('Legajo repetido... alta rechazada...')

print()
print('Otro legajo de estudiante (cargue 0 para salir): ')
leg = validar_legajo(0, 99999)

m.close()

print()
print('Operación de altas finalizada...')
input('Presione <Enter> para seguir...')

def baja():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a borrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(to_string(est))

            # ...chequemos si el usuario está seguro de lo que hace...
            r = input('Está seguro de querer borrarlo (s/n)?: ')
            if r in ['s', 'S']:
                # lo marcamos como borrado, y ya...
                est.activo = False

                # ...reubicamos el file pointer...
                m.seek(pos, io.SEEK_SET)

                # ...y lo volvemos a grabar...
                pickle.dump(est,m)

            print()

```

```

        print('Registro eliminado del archivo...')

    else:
        print('Ese registro no existe en el archivo...')

    print()
    print('Otro legajo de estudiante a borrar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)

m.close()

print()
print('Operación de bajas finalizada...')
input('Presione <Enter> para seguir...')


def modificacion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    m = open(FD, 'r+b')

    print()
    print('Legajo del estudiante a modificar (cargue 0 para salir): ')
    leg = validar_legajo(0, 99999)
    while leg != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, leg)
        if pos != -1:
            # encontrado... procedemos a cargarlo...
            m.seek(pos, io.SEEK_SET)
            est = pickle.load(m)

            # ...mostramos el registro tal como estaba...
            print()
            print('El registro actualmente grabado es:')
            print(to_string(est))

            # ...modificamos el valor de los campos...
            op = 0
            while op != 3:
                print('1. Modificar nombre.')
                print('2. Modificar promedio.')
                print('3. Terminar modificaciones.')
                op = int(input('\t\tIngresé opción: '))

                if op == 1:
                    nom = input('Nuevo nombre: ')
                    est.nombre = nom.ljust(30, ' ')

                elif op == 2:
                    print('Nuevo promedio:')
                    est.promedio = validar_promedio()

                elif op == 3:
                    pass

            # ...registro modificado en memoria...

```

```
# ...ahora nos volvemos a su posición en el archivo...
m.seek(pos, io.SEEK_SET)

# ...y volvemos a grabar el registro modificado...
pickle.dump(est, m)

print()
print('Los datos se actualizaron en el archivo...')

else:
    print('Ese registro no existe en el archivo...')

print('Otro legajo de estudiante a modificar (con 0 sale): ')
leg = validar_legajo(0, 99999)

m.close()

print()
print('Operación de modificaciones finalizada...')
input('Presione <Enter> para seguir...')

def depuracion():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)

    original = open(FD, 'rb')
    temporal = open('temporal.dat', 'wb')

    print('Archivo', FD, '(eliminación física de registros borrados)')
    while original.tell() < tbm:
        # cargar un registro del archivo original...
        est = pickle.load(original)

        # ...si no estaba marcado, grabarlo en el archivo temporal...
        if est.activo:
            pickle.dump(est, temporal)

    # cerrar ambos archivos...
    original.close()
    temporal.close()

    # eliminar el archivo original...
    os.remove(FD)

    # y renombrar el temporal...
    os.rename('temporal.dat', FD)

    print('Se eliminaron los registros marcados...')
    input('Presione <Enter> para seguir...')

def listado_completo():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
```

```
print()
return

tbm = os.path.getsize(FD)

m = open(FD, 'rb')

print('Listado general de estudiantes registrados:')
while m.tell() < tbm:
    est = pickle.load(m)
    if est.activo:
        print(to_string(est))

m.close()

print()
input('Presione <Enter> para seguir...')

def listado_filtrado():
    global FD
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
    print()
    return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')
    print('Listado de estudiantes con promedio mayor o igual a 7:')
    while m.tell() < tbm:
        est = pickle.load(m)
        if est.activo and est.promedio >= 7:
            print(to_string(est))

    m.close()

    print()
    input('Presione <Enter> para seguir...')

def main():
    global FD
    FD = 'estudiantes.utn'

    op = 0
    while op != 7:
        print('Opciones ABM del archivo de estudiantes')
        print('    1. Alta de estudiantes')
        print('    2. Baja de estudiantes')
        print('    3. Modificación de estudiantes')
        print('    4. Listado completo de estudiantes')
        print('    5. Listado de estudiantes con promedio >= a 7')
        print('    6. Depuración del archivo de estudiantes')
        print('    7. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
        print()

        if op == 1:
            alta()

        elif op == 2:
```

```
baja()

elif op == 3:
    modificacion()

elif op == 4:
    listado_completo()

elif op == 5:
    listado_filtrado()

elif op == 6:
    depuracion()

elif op == 7:
    pass

# script principal...
if __name__ == '__main__':
    main()
```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 24

## Archivos: Aplicaciones Prácticas

### 1.] Introducción.

En esta Ficha no se agregan temas teóricos nuevos. El contenido completo se orienta al desarrollo de ejercicios y casos de análisis para integrar las estructuras de arreglos, registros y archivos que se han visto hasta ahora y mostrar distintas formas prácticas en que un archivo puede usarse para almacenar datos que estaban contenidos en otras estructuras.

En todos los ejercicios que siguen, se suponen situaciones directas en la que se requiere procesar datos que están en una estructura (por ejemplo, un arreglo de registros) y copiar parte de estos datos (o un subconjunto de ellos) a otra estructura (por ejemplo, un archivo). No se trata de situaciones completas en las que un archivo de registros podría requerir un procesamiento ABM, sino de aplicaciones inmediatas para la práctica en el uso combinado de distintas estructuras de datos.

Por lo tanto, en la gestión de los archivos de registros que se pedirá manejar, no será necesario prever un *campo de marcado lógico* para eventuales operaciones de *bajas lógicas*, ni aplicar las estrategias básicas y esenciales que soportan a un programa ABM en general: sólo se espera que se apliquen las estrategias generales para cumplir con el objetivo inmediato sugerido por cada enunciado.

Cada uno de los casos de análisis que siguen, se presenta en su propia sección o capítulo dentro de esta Ficha, para facilitar su identificación y agrupar mejor las explicaciones que acompañen a cada uno.

### 2.] Caso de análisis: Combinación de estructuras – Arreglo generado en forma ordenada.

El primer problema o caso propuesto incluye una novedad, como es la generación de un arreglo de registros pero de forma que al agregar un nuevo registro se mantenga ordenado el contenido del arreglo (en base al valor de un campo particular). El enunciado es el siguiente:

**Problema 55.)** *Un consultorio médico necesita un programa para gestionar los datos de sus pacientes. Por cada paciente, se deben almacenar los siguientes elementos: número de historia clínica (un número entero), nombre del paciente, fecha de la última visita (en días – otro número entero) y código de la enfermedad o problema registrado (un valor entre 0 y 9 incluidos). Los datos deben cargarse y almacenarse inicialmente en un arreglo de registros, a razón de un registro por paciente, el cual debe mantenerse en todo momento ordenado de menor a mayor de acuerdo al valor del número de historia clínica de los pacientes. El programa debe incluir un menú con las opciones siguientes:*

1. *Cargar el arreglo con los registros pedidos (recuerde: el arreglo debe mantenerse ordenado por historia clínica: cada registro debe insertarse en el lugar correcto cuando se agrega al arreglo).*
2. *Cargar por teclado un número entero **d**, y mostrar por pantalla los datos de todos los pacientes del arreglo que hayan asistido al consultorio por última vez en un período de **d** días o más.*
3. *Determinar si en el arreglo existe un paciente con número de historia clínica igual a **x**. Si existe, mostrar todos sus datos. Si no, dar un mensaje de error.*
4. *Mostrar todos los datos del arreglo.*
5. *Grabe todos los datos del arreglo en un archivo (para favorecer el desarrollo de los puntos que siguen, asegúrese de hacerlo de forma que cada registro se grabe por separado, uno por uno). El archivo debe ser creado si no existía, y todo dato que hubiese contenido debe ser eliminado si ya existía.*
6. *Mostrar el archivo generado en el punto anterior.*
7. *Usando el archivo creado en el punto 5, crear en memoria **otro** arreglo que contenga los registros de los pacientes cuyo código de enfermedad sea 8 o 9. Recuerde: debe crear **otro arreglo** de registros, y no eliminar ni modificar el original que se creó en el punto 1.*
8. *Mostrar el arreglo creado en el punto 7.*

**Discusión y solución:** El proyecto [F24] Archivos - Aplicaciones que acompaña a esta Ficha contiene un modelo *ej01.py* con el programa completo que resuelve este caso de análisis.

La parte inicial del programa contiene las instrucciones para importar los módulos que serán necesarios, la declaración del registro *Paciente*<sup>1</sup>, las funciones *\_\_init\_\_()* y *to\_string()* que tradicionalmente usamos para gestionar un registro, y un par de funciones de validación de carga por teclado:

```
import pickle
import os.path

__author__ = 'Catedra de AED'

class Paciente:
    def __init__(self, hc, nom, fec, cod):
        self.hist_clinica = hc
        self.nombre = nom
```

---

<sup>1</sup> El cine ha explorado desde siempre el mundo de la medicina y nos ha entregado conmovedoras historias que constituyen un merecido reconocimiento a muchos médicos e investigadores que realizaron avances extraordinarios en el campo de la salud humana. En ese sentido, en 2004 la película para televisión cuyo título original es *Something The Lord Made* (dirigida por Joseph Sargent e interpretada por Alan Rickman y Mos Def) (en algunos sitios web el título fue traducido como *Una Creación del Señor*) está basada en la historia real del Dr. Alfred Blalock y su ayudante Vivien Thomas, quienes en la década de 1940 desarrollaron una técnica quirúrgica para salvar la vida de muchos niños recién nacidos que mostraban una malformación cardíaca congénita llamada "Tetralogía de Fallot" (o también "Síndrome de los Bebés Azules" debido a que esa malformación genera fallas en la irrigación sanguínea y por lo tanto en la oxigenación de los bebés, que se manifiesta en una coloración azulada en los labios y las puntas de los dedos). Sin la técnica de Blalock y Thomas, esos niños estaban condenados a morir a los pocos meses de nacidos. No dejen de verla...

```

        self.fecha = fec
        self.cod_problema = cod

def to_string(pac):
    r = ''
    r += '{:<25}'.format('Historia Clinica: ' + str(pac.hist_clinica))
    r += '{:<30}'.format('Nombre: ' + pac.nombre)
    r += '{:<20}'.format('Fecha: ' + str(pac.fecha))
    r += '{:<20}'.format('Problema: ' + str(pac.cod_problema))
    return r

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... se pidió mayor a', lim, '... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup) + ' : '))
        if n < inf or n > sup:
            print('\t\tError... cargue de nuevo...')
    return n

```

La carga por teclado del arreglo original  $p$  se hace mediante la función **cargar\_arreglo\_ordenado()**, la cual a su vez invoca a la función **add\_in\_order()** para insertar cada registro en la posición correcta del arreglo y mantenerlo ordenado de menor a mayor por número de historia clínica [1]:

```

def add_in_order(p, paciente):
    n = len(p)
    pos = n
    for i in range(n):
        if paciente.hist_clinica < p[i].hist_clinica:
            pos = i
            break

    p[pos:pos] = [paciente]

def cargar_arreglo_ordenado():
    p = []
    print('Cantidad de pacientes...')
    n = validar_mayor(0)

    print()
    print('Ingrese los datos de los pacientes...')
    for i in range(n):
        print('Número de historia clínica...')
        hc = validar_mayor(0)

        nom = input('Nombre: ')

        print('Días transcurridos desde su última visita...')
        dias = validar_mayor(0)

        print('Código de enfermedad o problema...')
        cod = validar_intervalo(0, 9)

        paciente = Paciente(hc, nom, dias, cod)

```

```

    add_in_order(p, paciente)
    print()

    return p

```

La función `cargar_arreglo_ordenado()` comienza creando un arreglo *p* inicialmente vacío. Luego pide al operador que ingrese la cantidad de pacientes *n* que serán cargados y mediante un ciclo *for* se cargan los datos de cada uno, en forma tradicional y validando aquellos que lo requieran. Cuando todos los datos de un paciente han sido cargados, se crea un registro de tipo *Paciente* mediante la función *init()*, y en lugar de agregarlo directamente al arreglo con el conocido método *append()*, se invoca a en su lugar a la función `add_in_order()`.

Esta función toma dos parámetros: el arreglo de registros *p*, y el registro *paciente* que se acaba de crear. En la función `add_in_order()` se supone que el arreglo *p* en todo momento está ordenado de menor a mayor de acuerdo al número de historia clínica, y la idea es recorrer el arreglo, encontrar la posición en la que debería insertarse el nuevo registro para el arreglo continúe ordenado, y agregarlo en esa posición.

La idea es esencialmente simple: se recorre el arreglo *p* con un ciclo *for*, controlando que en cada casilla el valor *p[i].hist\_clinica* sea menor que *paciente.hist\_clinica*: esto es, continuar mientras el paciente que ya está en el arreglo tenga un número de historia clínica menor al número del paciente que se quiere insertar. Está claro que el nuevo paciente *no puede* insertarse delante de otro cuyo número sea menor al propio, pues se perdería el orden de menor a mayor. Si durante el recorrido se encuentra un registro *p[i]* cuyo número de historia clínica sea mayor (o incluso igual) al que se quiere agregar, entonces en ese momento se detiene el ciclo, guardando en una variable local *pos* el índice de la casilla que contenía al registro con número mayor.

Al salir del ciclo, la variable *pos* contiene el índice del paciente que debería quedar como el siguiente del que se quiere insertar (el primer paciente con número de historia clínica mayor al del paciente nuevo). Por lo tanto, el registro *paciente* debe agregarse justamente en la posición *pos*, corriendo hacia la derecha a todos los registros desde allí en adelante, lo cual sabemos que puede hacerse con la instrucción siguiente [2] [3]:

```
p[pos:pos] = [paciente]
```

Un detalle interesante es que si el registro *paciente* que se quiere agregar tiene un número de historia clínica *mayor al de todos los registros del arreglo p*, entonces el paciente debe agregarse al final del arreglo. Antes de comenzar el ciclo *for*, la variable *pos* se inicializa con el valor *n* (el tamaño del arreglo en ese momento). Si el ciclo *for* no encuentra ningún paciente con número de historia clínica menor al que se quiere agregar, se detendrá al llegar al final del arreglo sin modificar el valor de *pos*. Pero en este caso, *pos* quedará valiendo finalmente *n*, por lo que la instrucción

```
p[pos:pos] = [paciente]
```

agregaría efectivamente el registro *paciente* al final del arreglo.

Note que la función `add_in_order()` efectivamente agrega un registro en un arreglo ordenado, de forma que el arreglo siga ordenado, pero para encontrar el *punto de inserción* (el lugar donde insertarse el nuevo registro) usa el algoritmo de búsqueda secuencial. Claramente esto podría mejorarse: en el contexto de este problema, el *arreglo está siempre*

ordenado por número de historia clínica, y por lo tanto, se puede aplicar el *algoritmo de búsqueda binaria* para encontrar el punto de inserción, en lugar del algoritmo de búsqueda secuencial. El tiempo de ejecución de esta búsqueda bajaría entonces de  $O(n)$  a  $O(\log(n))$ , mejorando mucho el rendimiento si el arreglo fuese muy grande. La forma de hacer este ajuste se discute en la página 490 de esta misma Ficha.

Las dos funciones que siguen muestran en pantalla el contenido del arreglo. La primera se llama *listado\_por\_dias()*, y toma como parámetro al arreglo *p* y a un número *d* que representa una cantidad de días. La función recorre el arreglo y muestra un *listado filtrado*: sólo los registros en los que el campo *fecha* sea mayor o igual al valor *d* recibido como parámetro. La segunda función se llama *mostrar\_arreglo()* y también toma dos parámetros: el arreglo *p* y una variable llamada *mensaje* que contiene una cadena de caracteres con el título que debe mostrarse antes del listado del contenido del arreglo. La función recorre ese arreglo y muestra su contenido completo (sin filtro):

```
def listado_por_dias(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    d = int(input('Días a comprobar desde la última visita: '))
    print('Pacientes con', d, 'o más días desde la última visita')
    for paciente in p:
        if paciente.fecha >= d:
            print(to_string(paciente))

    print()

def mostrar_arreglo(p, mensaje='Contenido:'):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print(mensaje)
    for paciente in p:
        print(to_string(paciente))

    print()
```

La que sigue es la función *buscar()*, que toma como parámetro al arreglo *p* y un número *x*, y aplica el ya conocido algoritmo de *búsqueda binaria* para determinar si existe un registro cuyo número de historia clínica coincida con *x*, mostrando su contenido en caso de encontrarlo o un mensaje aclaratorio en caso contrario. En este caso, *se puede aplicar sin dudarlo* el algoritmo de *búsqueda binaria* [1] porque el arreglo *p* en todo momento está ordenado justamente por el campo *hist\_clinica*:

```
def buscar(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = int(input('Número de historia clínica a buscar: '))

    n = len(p)
    izq, der = 0, n - 1
    while izq <= der:
        c = (izq + der) // 2
```

```

    if p[c].hist_clinica == x:
        print('Paciente encontrado...')
        print(to_string(p[c]))
        print()
        return

    if x < p[c].hist_clinica:
        der = c - 1
    else:
        izq = c + 1

print('No hay un paciente con ese número de historia clínica')
print()

```

Las funciones *crear\_archivo()* y *mostrar\_archivo()* son directas: la primera toma como parámetro el arreglo *p* y graba con la función *pickle.dump()* su contenido completo, registro por registro, en el archivo cuyo nombre físico viene en la variable global *FD*. La grabación se hace registro por registro debido a que en el punto 7 se pide recuperar *una parte* de esos registros para almacenarlos en un segundo arreglo. Si el arreglo original hubiese sido almacenado en forma directa con una sola instrucción *pickle.dump()*, entonces para generar el segundo arreglo se tendría que volver a recuperar con *pickle.load()* el arreglo original completo, y crear el segundo a partir de este (ambas técnicas son válidas). La segunda función se llama *mostrar\_archivo()* y también es directa: recorre, lee y muestra registro por registro el contenido completo del archivo *FD*:

```

def crear_archivo(p):
    global FD

    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Grabando todos los datos en el archivo', FD, '...')
    m = open(FD, 'wb')
    for paciente in p:
        pickle.dump(paciente, m)

    m.close()
    print('... hecho')
    print()

def mostrar_archivo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        print(to_string(pac))

    m.close()
    print()

```

Sólo nos queda la función `crear_segundo_arreglo()`, que toma el archivo generado con `crear_archivo()`, y crea a partir de este un segundo arreglo contenido solamente los registros de pacientes cuyo código de problema sea 8 o 9, retornando luego el arreglo creado:

```
def crear_segundo_arreglo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    p2 = []
    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Creando el segundo vector desde el archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        if pac.cod_problema in [8, 9]:
            p2.append(pac)

    m.close()
    print('... hecho')
    print()

    return p2
```

La función `main()` es la que despliega y gestiona el menú de opciones y es el punto de entrada del programa completo que se muestra a continuación:

```
import pickle
import os.path

__author__ = 'Catedra de AED'

class Paciente:
    def __init__(self, hc, nom, fec, cod):
        self.hist_clinica = hc
        self.nombre = nom
        self.fecha = fec
        self.cod_problema = cod

    def to_string(pac):
        r = ''
        r += '{:<25}'.format('Historia Clinica: ' + str(pac.hist_clinica))
        r += '{:<30}'.format('Nombre: ' + pac.nombre)
        r += '{:<20}'.format('Fecha: ' + str(pac.fecha))
        r += '{:<20}'.format('Problema: ' + str(pac.cod_problema))
        return r

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... se pidio mayor a', lim, '... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
```

```

while n < inf or n > sup:
    n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup)+ ' : '))
    if n < inf or n > sup:
        print('\t\tError... cargue de nuevo...')
    return n

def add_in_order(p, paciente):
    n = len(p)
    pos = n
    for i in range(n):
        if paciente.hist_clinica < p[i].hist_clinica:
            pos = i
            break
    p[pos:pos] = [paciente]

"""
# Misma función, pero con búsqueda binaria...
def add_in_order(p, paciente):
    n = len(p)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2
        if p[c].hist_clinica == paciente.hist_clinica:
            pos = c
            break

        if paciente.hist_clinica < p[c].hist_clinica:
            der = c - 1
        else:
            izq = c + 1

    if izq > der:
        pos = izq
    p[pos:pos] = [paciente]
"""

def cargar_arreglo_ordenado():
    p = []
    print('Cantidad de pacientes...')
    n = validar_mayor(0)

    print()
    print('Ingrese los datos de los pacientes...')
    for i in range(n):
        print('Número de historia clínica...')
        hc = validar_mayor(0)

        nom = input('Nombre: ')

        print('Días transcurridos desde su última visita...')
        dias = validar_mayor(0)

        print('Código de enfermedad o problema...')
        cod = validar_intervalo(0, 9)

        paciente = Paciente(hc, nom, dias, cod)
        add_in_order(p, paciente)
        print()

    return p

```

```

def listado_por_dias(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    d = int(input('Días a comprobar desde la última visita: '))
    print('Listado de pacientes con', d, 'o más días desde la última visita')
    for paciente in p:
        if paciente.fecha >= d:
            print(to_string(paciente))

    print()

def mostrar_arreglo(p, mensaje='Contenido:'):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print(mensaje)
    for paciente in p:
        print(to_string(paciente))

    print()

def buscar(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = int(input('Número de historia clínica a buscar: '))

    n = len(p)
    izq, der = 0, n - 1
    while izq <= der:
        c = (izq + der) // 2
        if p[c].hist_clinica == x:
            print('Paciente encontrado...')
            print(to_string(p[c]))
            print()
            return

        if x < p[c].hist_clinica:
            der = c - 1
        else:
            izq = c + 1

    print('No hay un paciente registrado con ese número de historia clínica')
    print()

def crear_archivo(p):
    global FD

    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Grabando todos los datos en el archivo', FD, '...')
    m = open(FD, 'wb')
    for paciente in p:
        pickle.dump(paciente, m)

```

```

m.close()
print('... hecho')
print()

def mostrar_archivo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        print(to_string(pac))

    m.close()
    print()

def crear_segundo_arreglo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    p2 = []
    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Creando el segundo vector desde el archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        if pac.cod_problema in [8, 9]:
            p2.append(pac)

    m.close()
    print('... hecho')
    print()

    return p2

def main():
    global FD
    FD = 'pacientes.med'

    p, p2 = [], []
    op = 0
    while op != 9:
        print('Consultorio Dr. Matasanos')
        print(' 1. Registrar pacientes en forma ordenada')
        print(' 2. Listado de pacientes según días de su última visita')
        print(' 3. Buscar un paciente por historia clínica')
        print(' 4. Listado completo de pacientes')
        print(' 5. Creación de un archivo con todos los datos registrados')
        print(' 6. Mostrar el archivo')
        print(' 7. Crear arreglo desde el archivo (código de enfermedad 8 o 9)')
        print(' 8. Mostrar el arreglo (código de enfermedad 8 o 9)')
        print(' 9. Salir')
        op = int(input('\t\tIngresé número de la opción elegida: '))

```

```

print()

if op == 1:
    p = cargar_arreglo_ordenado()

elif op == 2:
    listado_por_dias(p)

elif op == 3:
    buscar(p)

elif op == 4:
    mostrar_arreglo(p, 'Listado completo de pacientes registrados')

elif op == 5:
    crear_archivo(p)

elif op == 6:
    mostrar_archivo()

elif op == 7:
    p2 = crear_segundo_arreglo()

elif op == 8:
    mostrar_arreglo(p2, 'Pacientes (código de enfermedad 8 o 9)')

elif op == 9:
    pass

# script principal...
if __name__ == '__main__':
    main()

```

### 3.] Caso de análisis: Gestión de un archivo en forma directa.

Se propone ahora otro ejercicio a modo de caso de análisis para trabajar en clase, sobre manejo de archivos. La idea es mostrar una aplicación práctica que trabaje *directamente* sobre archivos, para resaltar que *no es obligatorio mover y/o copiar datos del archivo a un arreglo o viceversa...* Es correcto dominar las técnicas para mover datos de una estructura a la otra, pero también se debe comprender que operar directamente con los datos de un archivo es muy común, y muchas veces es lo único que se necesita hacer [1]. El enunciado del ejercicio propuesto es el siguiente:

**Problema 56.)** *La oficina regional de la Junta Electoral Provincial ha pedido un programa que almacene en un archivo los datos del padrón electoral de cierta localidad. Por cada persona habilitada para votar en esa localidad, se guarda su dni, su nombre, su edad y un indicador del sexo de esa persona ('v' para varón o 'm' para mujer). El programa debe incluir un menú de opciones que permita:*

1. *Cargar datos en el archivo, pero cuidando que no se repita ninguna persona dentro del mismo.*
2. *Mostrar los datos del archivo completo.*
3. *Determinar si en el archivo existe un votante con número de dni igual a x. Si existe, mostrar todos sus datos. Si no, informar que no existe.*
4. *Determinar cuántos votantes son varones y cuántos son mujeres en el archivo.*

5. Genere un segundo archivo, que contenga sólo los datos de los votantes mayores a 70 años.
6. Mostrar el archivo generado en el punto anterior.

**Discusión y solución:** El proyecto [F24] Archivos - Aplicaciones que acompaña a esta Ficha contiene un modelo *ej02.py* con el programa completo que resuelve este caso de análisis.

La parte inicial del programa como siempre, contiene las instrucciones para importar los módulos que serán necesarios, la declaración del registro *Votante*, las funciones *\_\_init\_\_()* y *to\_string()* que tradicionalmente usamos para gestionar un registro, y un par de funciones de validación de carga por teclado:

```
import io
import pickle
import os.path

__author__ = 'Catedra de AED'

class Votante:
    def __init__(self, dn, nom, ed, sex):
        self.dni = dn
        self.nombre = nom
        self.edad = ed
        self.sexo = sex

def to_string(vot):
    sx = 'Hombre'
    if vot.sexo == 'm':
        sx = 'Mujer'

    r = ''
    r += '{:<20}'.format('DNI: ' + str(vot.dni))
    r += '{:<30}'.format('Nombre: ' + vot.nombre)
    r += '{:<20}'.format('Edad: ' + str(vot.edad))
    r += '{:<20}'.format('Sexo: ' + sx)
    return r

def validar_dni(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_edad(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('Error... cargue de nuevo...')
    return n

def validar_sexo(caracteres):
```

```

c = ''
while c not in caracteres:
    c = input('Letras válidas ' + str(caracteres) + ': ')
    if c not in caracteres:
        print('\t\tError... letra no válida... cargue de nuevo...')
return c

```

La función *cargar\_archivo()* es la encargada de las operaciones de altas de nuevos registros en el archivo. Como se pide controlar que no se repitan los registros (es decir, controlar que no se graben dos registros con el mismo *dni*), se usa además la función *buscar()* para recorrer el archivo y controlar si un número de *dni* está ya registrado o no. Ambas funciones son prácticamente iguales a las que se presentaron en la *Ficha 23* para realizar altas y búsquedas, por lo que dejamos su análisis y repaso para el estudiante:

```

def buscar(m, dni):
    global FD1
    t = os.path.getsize(FD1)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        vot = pickle.load(m)
        if vot.dni == dni:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion

def cargar_archivo():
    global FD1
    m = open(FD1, 'a+b')

    print()
    print('DNI del votante a registrar (cargue 0 para salir): ')
    dni = validar_dni(-1)
    while dni != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, dni)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            print('Edad...')
            edad = validar_edad(0, 120)

            print('Sexo ("v": varón - "m": mujer)... ')
            sex = validar_sexo(['V', 'v', 'M', 'm'])

            vot = Votante(dni, nom, edad, sex)

            # ...lo grabamos...
            pickle.dump(vot, m)

            # ...volcamos el buffer de escritura
            # para que el sistema operativo REALMENTE

```

```

        # grabe en disco el registro...
        m.flush()
        print('Registro grabado en el archivo...')

    else:
        print('DNI repetido... alta rechazada...')

    print()
    print('Otro dni de votante a registrar (cargue 0 para salir): ')
    dni = validar_dni(-1)

m.close()
print()

```

La función *mostrar\_archivo()* muestra en forma completa el contenido de un archivo. Como el problema sugiere mostrar dos archivos diferentes a través de dos opciones del menú, y dado que esos archivos contienen registros del mismo tipo, la función *mostrar\_archivo()* toma un parámetro *FD* con el nombre físico del archivo a mostrar y abre, lee y muestra registro por registro ese archivo. La única diferencia entre esta función y otras que hemos visto para recorrer y mostrar un archivo, es que en esas funciones el nombre físico del archivo se suponía alojado en una *variable global*, y ahora ese nombre *viene como parámetro* (justamente, para poder usar la función con archivos de entrada diferentes):

```

def mostrar_archivo(FD):
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        vot = pickle.load(m)
        print(to_string(vot))

    m.close()
    print()

```

La función *buscar\_votante()* carga por teclado el número de *dni* de un votante, y utiliza la ya citada función *buscar()* para determinar si el archivo contiene o no a ese votante. Si existía, la función *buscar\_votante()* simplemente se reposiciona con *seek()* [2] [3] en el byte donde comienza el registro, lo lee y finalmente lo muestra. Y en caso de no existir, sólo se avisa con un mensaje:

```

def buscar_votante():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
    m = open(FD1, 'rb')

    print('DNI del votante a buscar: ')
    dni = validar_dni(0)
    pos = buscar(m, dni)
    if pos != -1:
        m.seek(pos, io.SEEK_SET)

```

```

        vot = pickle.load(m)
        print('Votante encontrado...')
        print(to_string(vot))

    else:
        print('Ese votante no está registrado...')

m.close()
print()

```

La función *cantidad\_por\_sexo()* abre el archivo en modo de sólo lectura, lo lee registro por registro, y va contando en dos contadores cuántos de los votantes son varones y cuántas mujeres. El proceso es directo, y dejamos su análisis para alumno:

```

def cantidad_por_sexo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
    tbm = os.path.getsize(FD1)
    m = open(FD1, 'rb')

    cv, cm = 0, 0
    while m.tell() < tbm:
        vot = pickle.load(m)
        if vot.sexo == 'v':
            cv += 1
        else:
            cm += 1

    m.close()

    print('Cantidad de votantes varones:', cv)
    print('Cantidad de votantes mujeres:', cm)
    print()

```

La última función se llama *crear\_segundo\_archivo()* y se usa para generar un segundo archivo que contenga una copia de todos los registros del archivo original que correspondan a votantes con más de 70 años de edad. El nombre de ambos archivos viene dado respectivamente por las variables globales *FD1* y *FD2*. El original se abre en modo de sólo lectura, mientras que el nuevo archivo se abre en modo 'wb'. A medida que se lee un registro del archivo original, se chequea si el campo edad del mismo es mayor a 70, y en caso de serlo, simplemente se procede a grabarlo en el segundo archivo:

```

def crear_segundo_archivo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD1)
    m = open(FD1, 'rb')
    s = open(FD2, 'wb')

    print('Creando archivo', FD2, 'con mayores a 70 años...')
    while m.tell() < tbm:

```

```

vot = pickle.load(m)
if vot.edad > 70:
    pickle.dump(vot, s)

m.close()
s.close()
print('... hecho')
print()

```

El programa completo que mostramos a continuación, incluye la función *main()* para manejar el menú de opciones, y esa función es el punto de entrada del programa:

```

import io
import pickle
import os.path

__author__ = 'Catedra de AED'

class Votante:
    def __init__(self, dn, nom, ed, sex):
        self.dni = dn
        self.nombre = nom
        self.edad = ed
        self.sexo = sex

def to_string(vot):
    sx = 'Hombre'
    if vot.sexo == 'm':
        sx = 'Mujer'

    r = ''
    r += '{:<20}'.format('DNI: ' + str(vot.dni))
    r += '{:<30}'.format('Nombre: ' + vot.nombre)
    r += '{:<20}'.format('Edad: ' + str(vot.edad))
    r += '{:<20}'.format('Sexo: ' + sx)
    return r

def validar_dni(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_edad(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('Error... cargue de nuevo...')
    return n

def validar_sexo(caracteres):
    c = ' '
    while c not in caracteres:

```

```

c = input('Letras válidas ' + str(caracteres) + ': ')
if c not in caracteres:
    print('\t\tError... letra no válida... cargue de nuevo...')
return c

def buscar(m, dni):
    global FD1
    t = os.path.getsize(FD1)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        vot = pickle.load(m)
        if vot.dni == dni:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion

def cargar_archivo():
    global FD1
    m = open(FD1, 'a+b')

    print()
    print('DNI del votante a registrar (cargue 0 para salir): ')
    dni = validar_dni(-1)
    while dni != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, dni)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            print('Edad...')
            edad = validar_edad(0, 120)

            print('Sexo ("v": varón - "m": mujer)... ')
            sex = validar_sexo(['V', 'v', 'M', 'm'])

            vot = Votante(dni, nom, edad, sex)

            # ...lo grabamos...
            pickle.dump(vot, m)

            # ...volcamos el buffer de escritura
            # para que el sistema operativo REALMENTE
            # grabe en disco el registro...
            m.flush()

            print('Registro grabado en el archivo...')

        else:
            print('DNI repetido... alta rechazada...')

    print()

```

```
print('Otro dni de votante a registrar (cargue 0 para salir): ')
dni = validar_dni(-1)

m.close()
print()

def mostrar_archivo(FD):
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        vot = pickle.load(m)
        print(to_string(vot))

    m.close()
    print()

def buscar_votante():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
    m = open(FD1, 'rb')

    print('DNI del votante a buscar: ')
    dni = validar_dni(0)
    pos = buscar(m, dni)
    if pos != -1:
        m.seek(pos, io.SEEK_SET)
        vot = pickle.load(m)
        print('Votante encontrado,...')
        print(to_string(vot))

    else:
        print('Ese votante no está registrado...')

    m.close()
    print()

def cantidad_por_sexo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
```

```
tbm = os.path.getsize(FD1)
m = open(FD1, 'rb')

cv, cm = 0, 0
while m.tell() < tbm:
    vot = pickle.load(m)
    if vot.sexo == 'v':
        cv += 1
    else:
        cm += 1

m.close()

print('Cantidad de votantes varones:', cv)
print('Cantidad de votantes mujeres:', cm)
print()

def crear_segundo_archivo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD1)
    m = open(FD1, 'rb')
    s = open(FD2, 'wb')

    print('Creando archivo', FD2, 'con datos de mayores a 70 años...')
    while m.tell() < tbm:
        vot = pickle.load(m)
        if vot.edad > 70:
            pickle.dump(vot, s)

    m.close()
    s.close()
    print('... hecho')
    print()

def main():
    global FD1, FD2
    FD1 = 'votantes.vot'
    FD2 = 'mayores.vot'

    op = 0
    while op != 7:
        print('Padrón electoral')
        print('    1. Registrar votantes en el padrón')
        print('    2. Listado de votantes')
        print('    3. Buscar un votante por dni')
        print('    4. Cantidad de varones y mujeres')
        print('    5. Crear archivo con votantes mayores a 70 años')
        print('    6. Mostrar el archivo de votantes mayores a 70 años')
        print('    7. Salir')
        op = int(input('\t\tIngresé número de la opción elegida: '))
        print()

        if op == 1:
```

```

cargar_archivo()

elif op == 2:
    mostrar_archivo(FD1)

elif op == 3:
    buscar_votante()

elif op == 4:
    cantidad_por_sexo()

elif op == 5:
    crear_segundo_archivo()

elif op == 6:
    mostrar_archivo(FD2)

elif op == 7:
    pass

# script principal...
if __name__ == '__main__':
    main()

```

#### 4.] Caso de análisis: Combinación de estructuras y uso de una matriz de conteo.

Este ejercicio vuelve a la combinación práctica de estructuras de datos diversas, pero ahora agregando también una *matriz de conteo*. El enunciado es el siguiente:

**Problema 57.)** Una editorial encargada de la publicación de una revista científica ha solicitado que se desarrolle un programa para gestionar su operatoria. Se deben almacenar en un arreglo unidimensional (un vector) los datos relacionados con los artículos disponibles para su publicación (cargar por teclado la cantidad *n* de artículos). Cada artículo tiene los siguientes datos: código (int), título (cadena), cantidad de páginas, tipo de artículo (puede ser un valor entre 0 y 9 identificando el campo de aplicación) y el idioma en que está escrito (un valor entre 0 y 5). Se pide un programa controlado por menú de opciones que permita:

1. Cargar por teclado el arreglo de artículos, que debe quedar ordenado alfabéticamente de acuerdo al título de los artículos. Alternativamente, la carga puede hacerse en forma automática, generando en forma aleatoria los valores a contener en cada registro. Además, verificar que ningún artículo aparezca repetido en el arreglo (no permita dos artículos con el mismo título).
2. Mostrar el arreglo completo.
3. Cargar por teclado el título de un artículo y determinar si existe alguno con ese título. Mostrar sus datos si existe, o informar que no existe.
4. Cargar por teclado el código de un artículo y determinar si existe alguno con ese código. Mostrar sus datos si existe, o informar que no existe.
5. Determinar la cantidad de artículos por tipo e idioma que hay en el arreglo (es decir, cuántos artículos tipo 0 están escritos en el idioma 0, cuántos tipo 0 están en el idioma 1, y así sucesivamente... con un total de  $10 \times 6 = 60$  contadores).

6. Generar un archivo que contenga todos los datos de los artículos cuya cantidad de páginas sea superior a 10.
7. Mostrar el archivo completo.

**Discusión y solución:** El proyecto [F24] Archivos - Aplicaciones que acompaña a esta Ficha contiene un modelo *ej03.py* con el programa completo que resuelve este caso de análisis.

Como de costumbre, la parte inicial del programa contiene las instrucciones para importar los módulos que serán necesarios, la declaración del registro *Artículo*, las funciones *init()* y *display()* para gestionar un registro, y las funciones de validación de carga por teclado.

La novedad es que ahora aparece una función *validar\_titulo()* para verificar que el título del artículo propuesto no esté ya cargado en el arreglo. Y como el arreglo se genera de forma que quede ordenado por *título*, entonces la búsqueda del *título* en el vector se hace con la función *busqueda\_binaria()*, que aplica justamente el algoritmo de *búsqueda binaria*. La misma función se usa luego para buscar un artículo dado el título del mismo, y mostrarlo en pantalla.

Como el enunciado sugiere que también se deberá buscar un artículo pero dado ahora su *código* (y no su *título*), entonces el programa incluye también una función llamada *busqueda\_secuencial()* para buscar secuencialmente ese *código* (ya que en ese caso, la *búsqueda binaria* no es aplicable: el arreglo no está ordenado por código...)

La carga del arreglo se hace mediante la función *cargar\_arreglo\_ordenado()*, la cual carga por teclado los datos de cada registro, y añade cada registro al arreglo mediante la función *add\_in\_order()*, que a su vez busca el punto de inserción aplicando también *búsqueda binaria* (en forma similar a lo expuesto en la sección de *Temas Avanzados*).

La función *conteo()* aplica una *matriz de conteos* para determinar cuántos artículos había de cada uno de los 10 tipos posibles y escritos en cada uno de los 6 idiomas disponibles. La matriz *cant* se crea con 10 filas (una por cada *tipo*) y 6 columnas (una por cada *idioma*), de forma que cada casillero valga inicialmente 0. Se recorre el arreglo de artículos, y por cada registro se toma el valor del campo *tipo* (lo cual da el número de fila *f* de su contador en la matriz) y el valor del campo *idioma* (que da el número de columna *c*). Luego se accede en forma directa al casillero *cant[f][c]* y se incrementa en uno su valor. Al finalizar, se recorre la matriz con un par de ciclos anidados, y se muestra adecuadamente cada casillero que sea diferente de cero [1].

Las funciones *crear\_archivo()* y *mostrar\_archivo()* no agregan novedad alguna en cuanto a las estrategias ya vistas para procesar un archivo, por lo que se deja su análisis para el estudiante. El programa completo se ve a continuación:

```
import pickle
import os.path

__author__ = 'Catedra de AED'

class Artículo:
    def __init__(self, cod, tit, pg, tp, id):
        self.codigo = cod
        self.título = tit
        self.páginas = pg
        self.tipo = tp
```

```

        self.idioma = id

def to_string(art):
    r = ''
    r += '{:<30}'.format('Titulo: ' + art.titulo)
    r += '{:<20}'.format('Codigo: ' + str(art.codigo))
    r += '{:<20}'.format('Paginas: ' + str(art.paginas))
    r += '{:<20}'.format('Tipo: ' + str(art.tipo))
    r += '{:<20}'.format('Idioma: ' + str(art.idioma))
    return r

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup)+ ': '))
        if n < inf or n > sup:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_titulo(p):
    tit = ''
    while tit == '' or busqueda_binaria(p, tit) != -1:
        tit = input('Título: ')
        if tit == '' or busqueda_binaria(p, tit) != -1:
            print('Nombre repetido o vacío... cargue de nuevo...')
    return tit

def busqueda_binaria(p, tit):
    # busca por título: algoritmo de búsqueda binaria...
    # ...el arreglo está ordenado por título...
    n = len(p)
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2

        if p[c].titulo == tit:
            return c

        if tit < p[c].titulo:
            der = c - 1
        else:
            izq = c + 1

    return -1

def busqueda_secuencial(p, cod):

```

```

# busca por código: algoritmo de búsqueda secuencial...
# ...el arreglo no está ordenado por código...
n = len(p)
for i in range(n):
    if p[i].codigo == cod:
        return i

return -1

def add_in_order(p, articulo):
    # inserta un registro en el arreglo, en forma ordenada...
    # ...pero aplicando búsqueda binaria para encontrar
    # el punto de inserción...
    n = len(p)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2

        if p[c].titulo == articulo.titulo:
            pos = c
            break

        if articulo.titulo < p[c].titulo:
            der = c - 1
        else:
            izq = c + 1

    if izq > der:
        pos = izq

    p[pos:pos] = [articulo]

def cargar_arreglo_ordenado():
    p = []
    print('Cantidad de artículos...')
    n = validar_mayor(0)

    print()
    print('Ingrese los datos de los artículos...')
    for i in range(n):
        tit = validar_título(p)

        print('Código...')
        cod = validar_mayor(0)

        print('Cantidad de páginas...')
        pg = validar_mayor(0)

        print('Tipo...')
        tp = validar_intervalo(0, 9)

        print('Idioma...')
        id = validar_intervalo(0, 5)

        articulo = Articulo(cod, tit, pg, tp, id)
        add_in_order(p, articulo)
        print()

```

```
return p

def mostrar_arreglo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Listado de artículos disponibles')
    for articulo in p:
        print(to_string(articulo))

    print()

def buscar_titulo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = input('Título a buscar: ')
    pos = busqueda_binaria(p, x)

    if pos != -1:
        print('Artículo encontrado...')
        print(to_string(p[pos]))
    else:
        print('No hay un artículo con ese título')

    print()

def buscar_codigo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = int(input('Código a buscar: '))
    pos = busqueda_secuencial(p, x)

    if pos != -1:
        print('Artículo encontrado...')
        print(to_string(p[pos]))
    else:
        print('No hay un artículo con ese código')

    print()

def conteo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    cf, cc = 10, 6
    cant = [cc*[0] for f in range(cf)]
```

```

for art in p:
    f = art.tipo
    c = art.idioma
    cant[f][c] += 1

print('Cantidad de artículos por tipo e idioma...')
for f in range(cf):
    for c in range(cc):
        if cant[f][c] != 0:
            print('Tipo', f, 'Idioma', c, 'Cantidad:', cant[f][c])

def crear_archivo(p):
    global FD

    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Cantidad de páginas a controlar...')
    cp = validar_mayor(0)

    print('Grabando artículos con más de', cp, 'en el archivo', FD, '...')
    m = open(FD, 'wb')
    for art in p:
        if art.paginas > cp:
            pickle.dump(art, m)

    m.close()
    print('... hecho')
    print()

def mostrar_archivo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        art = pickle.load(m)
        print(to_string(art))

    m.close()
    print()

def main():
    global FD
    FD = 'articulos.edi'

    p = []
    op = 0
    while op != 8:

```

```

print('Editorial UTN')
print('    1. Registrar artículos ordenados por título')
print('    2. Listado completo de artículos')
print('    3. Buscar un artículo por título')
print('    4. Buscar un artículo por código')
print('    5. Conteo por tipo e idioma')
print('    6. Crear archivo desde el arreglo (cantidad de páginas)')
print('    7. Mostrar el archivo')
print('    8. Salir')
op = int(input('\t\tIngrese número de la opción elegida: '))
print()

if op == 1:
    p = cargar_arreglo_ordenado()

elif op == 2:
    mostrar_arreglo(p)

elif op == 3:
    buscar_titulo(p)

elif op == 4:
    buscar_codigo(p)

elif op == 5:
    conteo(p)

elif op == 6:
    crear_archivo(p)

elif op == 7:
    mostrar_archivo()

elif op == 8:
    pass

# script principal...
if __name__ == '__main__':
    main()

```

## 5.] Inserción ordenada en un arreglo: localización del punto de inserción por búsqueda binaria.

En el *problema 55* de esta misma Ficha, se introdujo una función *add\_in\_order(p, paciente)* (ver 465 y siguientes) que tomaba como parámetro un arreglo de registros del tipo *Paciente*, supuestamente ordenado de acuerdo al campo *hist\_clinica* de los registros contenidos (el número de historia clínica), y procedía a agregar en el arreglo el registro *paciente* que también entraba como parámetro, pero de forma tal que el arreglo se mantenga ordenado de menor a mayor según el número de historia clínica (en el *problema 52* de esta misma Ficha, otra vez apareció la función *add\_in\_order()* pero esta vez para agregar el registro con los datos de un *artículo* a publicar... la idea es la misma). La función original era la siguiente:

```

def add_in_order(p, paciente):
    n = len(p)
    pos = n
    for i in range(n):
        if paciente.hist_clinica < p[i].hist_clinica:
            pos = i

```

```

        break
p[pos:pos] = [paciente]

```

Como dijimos oportunamente, la función aplica el algoritmo de *búsqueda secuencial* para encontrar el *punto de inserción* (es decir, el índice del casillero donde debería agregarse el nuevo registro) para que el arreglo continúe ordenado. Pero como también dijimos, si el arreglo está ya ordenado podemos aplicar *búsqueda binaria* en lugar de búsqueda secuencial para encontrar ese punto de inserción, y hacer el proceso mucho veloz: en lugar de un tiempo de ejecución  $O(n)$ , pasaríamos a un tiempo  $O(\log(n))$ .

El algoritmo de búsqueda binaria que conocemos determina si un arreglo ordenado contiene o no a un determinado valor  $x$ , retornando el índice del casillero que lo contiene en caso de existir, o -1 en caso de no existir. Pero ahora necesitamos que el algoritmo *busque y retorne el índice del primer elemento del arreglo que sea mayor o igual al que se quiere insertar, o bien retorne el tamaño n del arreglo si el valor a insertar es mayor que todos los que el arreglo contiene*, y para ello necesitamos algunos ajustes.

Por lo pronto, si el número de historia clínica del paciente a agregar ya existe en la posición  $pos$  en el arreglo, podemos detener la búsqueda en forma normal, e insertar el nuevo paciente en la misma posición  $pos$ , moviendo a la derecha a todos los registros desde allí en adelante (incluyendo al que tenía el mismo número de historia clínica).

Pero si no existe un paciente con el mismo número de historia clínica, el ciclo de búsqueda continuará hasta que los índices auxiliares  $izq$  y  $der$  se crucen (es decir, hasta que  $izq$  se haga mayor que  $der$ ). Lo interesante es que en el momento en que se crucen, el valor de  $izq$  será el índice de la casilla con el *primer valor mayor* que el que se quiere agregar (y de hecho,  $der$  contendrá el índice del casillero con el *último valor menor* al que se quiere insertar). Por lo tanto, en este caso, sólo debemos insertar el nuevo registro en la posición indicada por  $izq$ , y correr hacia la derecha a todos los registros desde esa posición en adelante.

Con estos cambios, la función *add\_in\_order()* podría quedar finalmente así:

```

def add_in_order(p, paciente):
    n = len(p)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2
        if p[c].hist_clinica == paciente.hist_clinica:
            pos = c
            break

        if paciente.hist_clinica < p[c].hist_clinica:
            der = c - 1
        else:
            izq = c + 1

    if izq > der:
        pos = izq
    p[pos:pos] = [paciente]

```

La variable  $pos$  comienza valiendo  $n$ , el tamaño del arreglo, y ese es el valor que quedará asignado en  $pos$  **si no existe** un paciente con el mismo número de historia clínica que el que se quiere agregar. Si existiese un paciente con el mismo número, el valor de  $pos$  cambia para tomar el valor del índice  $c$  de la casilla que lo contiene, y el ciclo de búsqueda corta con *break*.

Al terminar el ciclo, se chequea si el valor de *izq* terminó siendo mayor que *der*. Si ese fue el caso, significa que el ciclo continuó su marcha sin activar el *break* citado en el párrafo anterior, y esto a su vez implica que ese registro no existía en el arreglo. Por lo tanto, sólo en ese caso, el valor de *pos* cambia para tomar el valor de *izq*, y finalmente se inserta el nuevo registro en la posición indicada por *pos*. Asegúrese el estudiante de hacer un análisis detallado del mecanismo aplicado en esta función, y comprender los detalles.

---

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 25

## Archivos de Texto

---

### 1.] Introducción.

Hemos visto que en general, en programación se acepta clasificar a los archivos en *archivos binarios* y *archivos de texto*; y hasta aquí nos hemos concentrado en la gestión de *archivos binarios*. También hemos indicado que en realidad *todo archivo es binario*, ya que en todo archivo finalmente se almacenan secuencias de bits que representan cualquier clase de información almacenada en el archivo [1]. Sin embargo, por razones de claridad y simplificación de terminología, es normal hablar de *archivos de texto* para referirse a ciertos tipos particulares de archivos binarios en los que cada byte del archivo se grabó con la intención real y efectiva de hacer que ese byte represente un carácter visualizable o forme parte de la representación de un carácter visualizable (y no, por ejemplo, de hacer que un conjunto de bytes agrupados representen un número entero o un número flotante).

Existen en ese sentido numerosos estándares de codificación binaria de caracteres. Esos estándares están basados en tablas de conversión en las que a cada carácter posible se le asocia un número, y la conversión a binario de ese número representa finalmente al carácter. La tabla más conocida, y también la más simple y una de las más antiguas usadas en la industria informática es la tabla *ASCII* (por *American Standard Code for Information Interchange*), que también se conoce como *ASCII 7* o también como *US-ASCII*. En el estándar *ASCII 7* cada carácter se representa exactamente con un byte, dejando el primer bit en 0 y usando el resto de los siete bits para representar el número de orden del carácter. Como sólo se usan siete bits, se pueden representar entonces no más de 128 caracteres (lo cual en un principio era consistente con las necesidades mínimas del mercado: letras del alfabeto inglés, dígitos, algunos signos de puntuación, algunos operadores y un puñado de caracteres de control para indicar saltos de línea, retornos de carro, saltos de página, etc.)

Sin embargo, con el correr del tiempo se hizo evidente que un conjunto tan reducido de caracteres era insuficiente, sobre todo con el advenimiento de los sistemas operativos de interfaz gráfica y el uso cada vez más globalizado de la informática. Así comenzaron entonces a aparecer variantes y expansiones del estándar *ASCII* original. Una de esas variantes se conoce como *ASCII 8* y emplea los ocho bits de un byte para representar caracteres, con lo cual el número total se extiende a 256 caracteres (y es la variante que se usó para incorporar caracteres especiales no incluidos en el idioma inglés pero comunes en otros idiomas (como los de base latina): la ñ del español, la ç del francés o el portugués, o las letras vocales acentuadas, por ejemplo).

Otras expansiones del estándar fueron claramente necesarias: con sólo 256 combinaciones no era posible incorporar caracteres de alfabetos no latinos (como el griego) o de alfabetos orientales (como el japonés, el chino o el coreano), ni representar caracteres icónicos tan

comunes y cotidianos hoy en día (como "las caritas felices" por ejemplo... ☺)<sup>1</sup>. Para lograr mayor amplitud en la representación, se planteó el estándar *Unicode* con fundamentos más complejos pero mucho más amplio que el estándar *ASCII*. Esencialmente, *Unicode* admite como un subconjunto válido al estándar *ASCII*, por lo que un texto codificado en *ASCII* será perfectamente legible por una aplicación o programa basado en *Unicode*.

A su vez, se han planteado tres formas de codificación de caracteres en base al estándar *Unicode*, designadas como *UTF-8*, *UTF-16* y *UTF-32* (y siendo *UTF* la abreviatura de *Unicode Transformation Format* o *Formato de Transformación Unicode*), y a su vez estas tres formas dan lugar a numerosos esquemas técnicos de codificación cuya explicación escapa a los alcances de esta Ficha. En general, es suficiente con saber que mediante *Unicode* (en cualquiera de las tres formas de codificación que se emplee) se puede representar la totalidad de los caracteres existentes en todos los idiomas, incluidas lenguas muertas, lenguas orientales, lenguas latinas y no latinas y caracteres gráficos especiales.

El lenguaje Python emplea por default el estándar *Unicode* bajo la forma de codificación *UTF-8* para interpretación de texto [2]. Como el estándar *ASCII* está incluido como un subconjunto de *Unicode*, cualquier archivo de texto con formato *ASCII* puro será leído y gestionado sin problemas. Cuando se abre un archivo de texto en Python, se pueden leer y grabar cadenas de caracteres en ese archivo con relativa sencillez. Los caracteres serán interpretados al leerlos (o codificados al grabarlos) en formato *UTF-8*.

Un detalle que tiene su importancia en cuanto a la posibilidad de usar archivos de texto en distintas plataformas, es la forma en que se interpretan los *caracteres de fin de línea* que estén contenidos en un archivo de texto. Básicamente, un *caracter de fin de línea* es un *caracter especial de control*, que se inserta en un texto para que al ser leído se interprete de forma de producir un *cambio de renglón* o *salto de línea* cuando ese texto se muestre en pantalla.

Como sabemos, un *caracter de control* se representa con una barra (\) seguida de un carácter que es el que especifica de qué carácter de control se trata. El hecho es que en distintos sistemas operativos el carácter de control de salto de línea se representa en formas diferentes, trayendo luego problemas cuando el mismo archivo debe ser leído en distintas plataformas. En el sistema operativo Unix (así como en Linux) el carácter de fin de línea se representa con \n, mientras en Windows se representa con una pareja de caracteres de la forma \r\n (esto es, un retorno de carro (\r) seguido de un salto de línea (\n)).

Para evitar problemas debidos a esta incómoda situación, cuando en Python **se lee un archivo de texto** la acción por default es convertir en la cadena leída los caracteres de fin de línea que aparezcan (el \n de Unix o el par \r\n de Windows) simplemente en un \n. De esta

---

<sup>1</sup> Los modernos sistemas de codificación digital de caracteres parecerían cosa de magia para los hombres de las épocas previas al siglo XV en que apareció la primera imprenta de tipos móviles (atribuida a Johannes Gutenberg). En esos tiempos medievales los libros se escribían, copiaban e ilustraban a mano; y era muy común en Occidente que esa tarea fuera realizada por monjes en las bibliotecas de los monasterios cristianos. La película *Der Name Der Rose* (o *El Nombre de la Rosa*) de 1986, dirigida por Jean-Jacques Annaud y protagonizada por Sean Connery, está ambientada en una de esas bibliotecas y gira alrededor de una serie de crímenes que parecen sobrenaturales. El fraile Guillermo de Baskerville llega al convento para investigar esos misteriosos asesinatos y devolver la paz a los monjes, en un contexto tenebroso donde no falta la superstición, el éxtasis religioso, la Inquisición, la hipocresía y algún que otro romance *non sancto...* La película está basada en la novela original del mismo nombre del genial Umberto Eco, publicada en 1980.

forma, cualquiera sea la plataforma de origen del archivo, los saltos de renglón en la pantalla serán interpretados en forma consistente. Y en forma recíproca, cuando **se graba en un archivo de texto** la acción por default es convertir los caracteres \n que las cadenas a grabar contengan y grabar en el archivo los caracteres que correspondan según el sistema operativo huésped en ese momento (es decir, si se está trabajando en Windows y se graba una cadena de caracteres en un archivo de texto, los caracteres \n que esa cadena tenga se grabarán convertidos en parejas \r\n) [2].

Como esta acción eventualmente modifica el contenido original del archivo, el programador debe estar especialmente seguro que el archivo que está procesando es de texto, y abrirlo como tal (colocando la letra 't' en el modo de apertura, o no colocando ni la 't' ni la 'b' (que como vimos, equivale por default a colocar una 't')). Si se abre en modo texto un archivo pensado como binario (por ejemplo, un archivo .exe o un .jpg o un .pdf), estos cambios podrían corromper el contenido del archivo, perdiendo datos o haciendo imposible abrirlo en forma normal. Si el archivo *no es de texto*, entonces, debería abrirse colocando la 'b' en el modo de apertura, lo cual *desactiva* el reemplazo de caracteres de salto de línea [2].

## 2.] Uso de archivos de texto en Python.

La forma abrir y usar un archivo de texto en Python es esencialmente la misma que la explicada para archivos binarios. Un archivo de texto también debe abrirse para poder operar con él, y la apertura se hace con la misma función *open()* ya analizada para los archivos binarios. La diferencia, es que al indicar el modo de apertura deberá reemplazarse la letra 'b' que se agregaba para los archivos binarios, por una 't', o bien, no escribir ninguna de las dos letras (lo que por default implica que el archivo es de texto) [1]. El significado de los modos de apertura es exactamente el mismo que el de los equivalentes para archivos binarios (consulte la tabla de modos de apertura de la *Ficha 22*). La siguiente instrucción abre el archivo '*datos.txt*' como archivo de texto, en modo de grabación y lectura:

```
m = open('datos.txt', 'w+t')
```

y como se indicó, es lo mismo que:

```
m = open('datos.txt', 'w+')
```

Una vez que el archivo ha sido abierto, la variable *file object* (en nuestro caso, *m*) que representa al archivo dispone de varios métodos (tales como *write()*, *read()*, *readline()* y *readlines()*) que permiten grabar o leer cadenas de caracteres con sencillez desde ese archivo.

En ese sentido, surge aquí una diferencia práctica entre lo que hacíamos para grabar y leer registros en un archivo binario y lo que haremos para grabar o leer cadenas en un archivo de texto: para grabar o leer registros empleábamos el mecanismo de *serialización*, a través de las funciones *pickle.dump()* y *pickle.load()* para simplificar el proceso, ya que los métodos *read()* y *write()* provistos por las variables *file object* requerían un poco de trabajo extra para poder operar con registros o variables de estructura compleja.

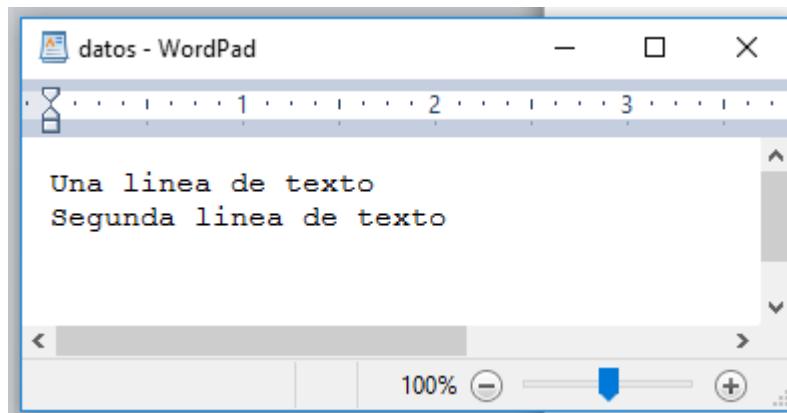
Sin embargo, esos mismos métodos permiten la grabación o la lectura de una cadena de caracteres en un archivo de texto en forma muy directa y muy simple. El método *write()* graba una cadena de caracteres en el archivo, y retorna la cantidad de caracteres que efectivamente grabó [2] [3] (en la función del ejemplo siguiente, el valor returnedo por

`write()` está siendo ignorado). Note que el archivo debe estar abierto en un modo que permita grabaciones, y que este método **no agrega** un salto de línea al final de la cadena, por lo cual el programador debe incluir un carácter '`\n`' explícitamente si lo desea:

```
def grabar_lineas():
    m = open('datos.txt', 'wt')
    m.write('Una linea de texto\n')
    m.write('Segunda linea de texto')
    m.close()
```

El archivo *datos.txt* generado al invocar la función *grabar\_lineas()* que acabamos de mostrar, contendrá dos líneas de texto, tal como se ve a continuación:

Figura 1: Contenido del archivo *datos.txt* creado con la función *grabar\_lineas()* (*con* salto de línea).

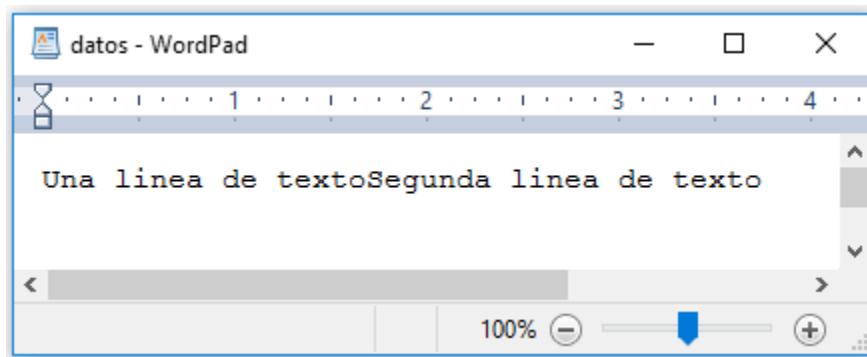


Como se indicó, el método *write()* no agrega un carácter de salto de línea al final de la cadena cuando la graba: es responsabilidad del programador agregar ese carácter cuando lo requiera. Si el ejemplo anterior se hubiese planteado así (sin el carácter '`\n`' al final de la primera cadena):

```
def grabar_lineas():
    m = open('datos.txt', 'wt')
    m.write('Una linea de texto')
    m.write('Segunda linea de texto')
    m.close()
```

entonces el contenido del archivo generado se vería así:

Figura 2: Contenido del archivo *datos.txt* creado con la función *grabar\_lineas()* (*sin* salto de línea).



Igual que con los archivos binarios, El método `close()` se usa para cerrar la conexión con el archivo y liberar cualquier recurso que estuviese asociado a él. Luego de invocar a `close()`, la variable que representaba al archivo queda indefinida. Recuerde, además, que en Python los archivos son cerrados automáticamente cuando la variable usada para accederlos sale del ámbito en que fue definida. Por lo tanto, en la función anterior no es estrictamente necesario invocar a `close()`.

Por otra parte, el método `read()` (invocado sin parámetros) permite leer el *contenido completo del archivo*, retornándolo como una sola y única cadena de caracteres. Se puede pasar como parámetro un número entero positivo, en cuyo caso `read(n)` leerá y retornará exactamente *n* bytes desde el archivo. Los caracteres '\n' que el archivo pudiera tener se preservan y se copian en la cadena retornada:

```
def leer_todo():
    m = open('datos.txt', 'r')
    todo = m.read()
    print('Contenido completo:')
    print(todo)
    m.close()
```

Si el archivo *datos.txt* fue generado con un salto de línea entre la primera cadena y la segunda (como se muestra en la *Figura 1*), entonces la salida en pantalla producida por la invocación de la función `leer_todo()` que se acaba de mostrar será la que sigue:

```
Contenido completo:
Una linea de texto
Segunda linea de texto
```

Otro método útil para procesar archivos de texto es el método `readline()`, que permite leer *una línea simple del archivo*, retornándola como una cadena de caracteres. La lectura se iniciará desde donde se encuentre el *file pointer* en ese momento, y terminará en el primer salto de línea que encuentre o al llegar al final del archivo. Si no puede leer una nueva cadena desde el archivo, por ejemplo por haber llegado ya al final del mismo, el método `readline()` retorna una cadena vacía ("") [2] [3]:

```
def leer_lineas():
    m = open('datos.txt')
    l1 = m.readline()
    l2 = m.readline()
    print('\nContenido linea por linea:')
    print(l1, end='')
    print(l2)
    m.close()
```

Note que `readline()` mantiene al final de la cadena retornada el carácter '\n', el cual sólo es omitido en la línea final del archivo (si el mismo no termina a su vez con '\n'). Es por este motivo que en la función anterior hemos agregado el parámetro `end=""` en la segunda invocación a `print()`, para evitar que aparezca una línea en blanco innecesaria en la salida por consola.

Como el archivo *datos.txt* contiene dos líneas de texto separadas por un salto de línea, la función `leer_lineas()` invoca dos veces a `readline()` para poder recuperar las dos líneas: la primera invocación recupera la cadena 'Una línea de texto' y sólo esa cadena, ya que la misma termina con un '\n'. El *file pointer* queda entonces apuntando al primer carácter de la segunda cadena, la cual es recuperada por la segunda invocación a `readline()`. La cadena

leída será entonces '*Segunda línea de texto*', la cual será retornada sin un '\n' al final, ya que esa cadena finaliza en el mismo lugar donde termina el archivo (y sin '\n').

En el ejemplo anterior, el archivo *datos.txt* contenía sólo dos líneas, y para leerlo línea por línea en forma completa bastaban dos invocaciones a *readline()*. Pero si el archivo contuviese una cantidad mucho mayor de líneas y/o no se conociese esa cantidad, entonces una forma alternativa y simple de leer el archivo completo, línea por línea, consiste en usar un ciclo *for que itere sobre el contenido del file object*:

```
def leer_con_iterador():
    m = open('datos.txt')
    print('\nContenido con iterador:')
    for line in m:
        print(line, end=' ')
    m.close()
```

Lo anterior es simple y cómodo, pero a veces se requiere mayor control sobre el flujo de lectura. Se puede implementar un ciclo que vaya leyendo el archivo con *readline()*, verificando si esa función retornó o no la cadena vacía (indicador de que se ha llegado al final del archivo). En la función *lectura\_controlada()* que sigue mostramos cómo hacer esto, y agregamos además un pequeño y útil proceso sobre la cadena leída, para eliminar el carácter de salto línea del final mediante un corte de índices [2] [3]:

```
def lectura_controlada():
    m = open('datos.txt')
    print('\nContenido con control de lectura:')
    while True:
        # intentar leer una linea...
        line = m.readline()

        # si se obtuvo una cadena vacia... cortar el ciclo y terminar...
        if line == '':
            break

        # si lo tenía, eliminar el carácter de salto de línea...
        if line[-1] == '\n':
            line = line[:-1]

        # procesar la cadena resultante...
        print(line)

    # cerrar antes de irnos...
    m.close()
```

Si la variable *line* es una cadena de caracteres, la expresión *if not line:* equivale a la expresión *if line == ""*: con lo cual la función mostrada está detectando el final del archivo y cortando el ciclo con un *break*. La expresión *line = line[:-1]* está tomando *todo el contenido de la cadena line salvo el último carácter*, y asignando la cadena cortada nuevamente en *line*. El resultado de esto es que se elimina el carácter '\n' que *line* tenía al final.

Otro método disponible para leer desde un archivo de texto es *readlines()* (con una *s* al final...). Si es invocado sin parámetros, retorna *una lista* (un arreglo) que contiene a todas las líneas del archivo (**incluidos** los '\n' al final de cada una). Se puede invocar al método pasándole como parámetro un valor numérico, en cuyo caso *readlines(n)* intentará recuperar hasta *n bytes* del archivo. En este caso, solo incluirá en la lista retornada las cadenas *completas* que haya logrado leer:

```

def list_lines():
    m = open('datos.txt', 'r')
    lista = m.readlines()
    print('\nLista de lineas contenidas:')
    print(lista)
    m.close()

```

Puede ver la implementación de todos los ejemplos y modelos que se han analizado hasta aquí en el modelo *test01.py* del proyecto [F25] Archivos de Texto que acompaña a esta Ficha.

### 3.] Reposicionamiento del file pointer en un archivo de texto en Python.

Para finalizar con el manejo de archivos de texto en Python, digamos que en forma similar a lo que ya hemos estudiado para los archivos binarios en Python pueden emplearse los mismos métodos *tell()* y *seek()* para consultar y cambiar el valor del *file pointer* de un archivo de texto, de forma de poder eventualmente realizar operaciones de lectura y/o grabación por acceso directo al byte en el cual se quiere comenzar (vea los ejemplos de uso y aplicación en el modelo *test02.py* en el proyecto [F25] Archivos de Texto que acompaña a esta Ficha).

Como sabemos, el método *tell()* retorna un número entero con el valor que en ese momento tenga el *file pointer* del archivo, medido en la cantidad de bytes desde el inicio del mismo y considerando al primer byte como en la posición 0(cero). Recuerde que si el *file pointer* está ubicado al final de un archivo (el primer byte inmediatamente luego del último que pertenece al archivo), entonces el valor retornado por *tell()* indica el tamaño en bytes del archivo (al igual que lo que ya vimos para archivos binarios) [1]:

```

m = open('prueba', 'w')
m.write('Universidad')
pos = m.tell()

# muestra: 11
print('Posicion del file pointer / Tamaño del archivo:', pos)

```

No confundir *tamaño del archivo en bytes*, con *cantidad de caracteres del archivo*. En el formato *Unicode – esquema UTF-8*, un carácter puede llegar a ser representado con 1, 2, 3 o 4 bytes dependiendo de cuál sea ese carácter, por lo que no necesariamente un tamaño de *k* bytes es equivalente a *k* caracteres en un archivo de texto.

El ya conocido método *seek()* permite cambiar el valor del *file pointer* también en un archivo de texto, aunque ahora existen algunas restricciones menores que distinguen su uso respecto de los archivos binarios.

Vimos que *seek()* en general recibe dos parámetros: el primero indica cuántos bytes debe moverse el *file pointer*, y el segundo indica desde donde se hace ese salto (el valor *io.SEEK\_SET = 0* indica saltar desde el principio del archivo, el valor *io.SEEK\_CUR = 1* indica saltar desde donde está el *file pointer* en ese momento y el valor *io.SEEK\_END = 2* indica saltar desde el final). El valor por default del segundo parámetro es *io.SEEK\_SET = 0*, por lo cual por omisión se asume que los saltos son desde el inicio del archivo (revise la *Ficha 22* para repasar estos conceptos si lo necesita).

Pero note que si se está trabajando con *archivos de texto*, en general *solo pueden hacerse saltos desde el principio del archivo (si el primer parámetro es un valor diferente de cero, entonces el segundo parámetro obligatoriamente debe ser io.SEEK\_SET)*. Si el segundo

parámetro enviado a `seek()` es `io.SEEK_END` o `io.SEEK_CUR`, entonces el primer parámetro debe ser obligatoriamente igual a 0. Si el archivo fue abierto como archivo de texto, cualquier otra combinación de valores para el primer parámetro y el segundo parámetro de `seek()` provocará un error de tiempo de ejecución y el programa se interrumpirá [2]:

```
def pruebas():
    m = open('prueba', 'w')
    m.write('Universidad')
    pos = m.tell()
    print('Posicion del file pointer / Tamaño del archivo:', pos)

    # repositionamiento correcto:
    m.seek(4, io.SEEK_SET)

    pos = m.tell()
    print('Posicion del file pointer ahora:', pos)

    # repositionamiento correcto:
    pos = m.seek(0, io.SEEK_END)

    print("Posicion del file pointer ahora: ", pos)

    # lo siguiente provoca un error...
    m.seek(3, io.SEEK_CUR)
```

Si la función anterior fuese ejecutada tal como está, el resultado será algo como:

```
Traceback (most recent call last):
  File "C:/Documentos/test02.py", line 31, in <module>, line 25, in main
    m.seek(3, io.SEEK_CUR)
io.UnsupportedOperation: can't do nonzero cur-relative seeks
```

Y si se ejecutase la siguiente instrucción en el mismo programa:

```
# lo siguiente también provoca un error...
m.seek(10, io.SEEK_END)
```

también obtendremos una situación error como la que sigue:

```
Traceback (most recent call last):
  File "C:/Documentos/test02.py", line 19, in main
    m.seek(10, io.SEEK_END)
io.UnsupportedOperation: can't do nonzero end-relative seeks
```

Está claro que si se invoca a `seek()` en esta forma:

```
m.seek(0, io.SEEK_END)
```

el efecto será mover el *file pointer* al final del archivo, y eso es válido tanto en archivos de texto como binarios. Y si se hace algo como:

```
m.seek(0, io.SEEK_CUR)
```

entonces el *file pointer* simplemente mantendrá el valor que ya tenía (tanto si el archivo es de texto como si es binario), sin cambios. Esto puede parecer poco útil, pero al menos no provoca un error si el archivo es de texto.

Existen otros métodos (y algunos atributos) incluidos en una variable/objeto de tipo *file object* en Python, **aplicables tanto para archivos de texto como para archivos binarios**.

Algunos han sido presentados en fichas anteriores pero mostramos los más comunes en la tabla siguiente:

**Tabla 1: Algunos métodos / atributos comunes incluidos en objetos tipo file object en Python.**

Método o Atributo	Aplicación
closed	Es un <u>atributo o campo</u> (no un método) cuyo valor es <i>True</i> si el archivo está cerrado.
flush()	Vuelca al archivo los buffers de grabación, si corresponde. Aplicable sólo para archivos abiertos para grabar (no hace nada en caso contrario).
readable()	Retorna <i>True</i> si el archivo está disponible para ser leído.
truncate(size=None)	Trunca el contenido del archivo a una cantidad igual a <i>size</i> bytes. El archivo puede <i>aumentar o disminuir</i> su tamaño.
writable()	Retorna <i>True</i> si el archivo está disponible para ser grabado.
writelines(lines)	Graba el contenido de la lista <i>lines</i> en el archivo, por defecto <i>sin</i> incluir separadores de línea (por lo cual, el programador debe indicar ese separador al final de cada elemento de la lista <i>lines</i> si quiere los separadores).

El siguiente problema servirá para mostrar la forma general de aplicar técnicas esenciales de manejo de archivos de texto:

**Problema 58.) Desarrollar un programa con menú de opciones que permita abrir un archivo que se sabe es de texto, y proceda a operar con él a partir de las siguientes opciones:**

- a.) *Cargar por teclado el nombre y la extensión de un archivo. Almacenar ese nombre en una variable fd de uso general para el resto de las opciones, y regresar al menú.*
- b.) *Crear el archivo fd (si ya existía, limpiar su contenido y abrirlo vacío) y grabar en él un texto cargado por teclado por el usuario.*
- c.) *Abrir el archivo fd, mostrar su contenido completo y permitir que el usuario cargue nuevo texto desde el teclado y se agregue al final del archivo. Si no existe el archivo, crearlo y también permitir que se grabe texto nuevo en el archivo, cargando ese texto por teclado.*
- d.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y mostrar su contenido completo. Si no existe, informar con un mensaje y retornar al menú.*
- e.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y determinar cuántas veces ese archivo contenía una palabra x que se carga por teclado antes de la consulta.*
- f.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y duplicar su contenido: agregar al final del archivo (sin eliminar el contenido anterior) una copia completa de su contenido actual.*
- g.) *Comprobar si existe el archivo fd, y en ese caso abrirlo y truncar su contenido para que finalmente sólo contenga tantas líneas como indique la variable cl que se carga por teclado (de las que ya tenía el archivo), eliminando todo el resto. Si el archivo tenía menos de cl líneas de texto, no hacer nada y regresar al menú.*

**Discusión y solución:** El proyecto [F25] Archivos de Texto que acompaña a esta Ficha contiene un modelo *test03.py* con el programa completo que resuelve este caso de análisis.

La función que hemos designado como *main()* será la función o punto de entrada del programa. Contiene el menú de opciones y la declaración inicial de la variable *fd* en la que se almacenará el nombre físico del archivo. Si nunca se selecciona la opción 1, esa variable

quedará asignada por defecto con la cadena '*'default.txt'*' (y ese será el nombre del archivo a manejar en ese caso):

```
def main():
    # nombre físico por default del archivo...
    fd = 'default.txt'

    op = 0
    while op != 8:
        print('Opciones para gestión del archivo de texto:', fd)
        print(' 1. Cargar nombre físico del archivo')
        print(' 2. Crear archivo nuevo y grabar texto')
        print(' 3. Abrir archivo y agregar texto al final')
        print(' 4. Mostrar contenido completo del archivo')
        print(' 5. Buscar y contar una palabra o cadena en el archivo')
        print(' 6. Duplicar contenido del archivo (en el mismo archivo)')
        print(' 7. Truncar contenido del archivo a dos líneas')
        print(' 8. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
        print()

        if op == 1:
            fd = cargar_nombre()

        elif op == 2:
            crear_archivo(fd)

        elif op == 3:
            abrir_archivo(fd)

        elif op == 4:
            listado_completo(fd)

        elif op == 5:
            buscar_cadena(fd)

        elif op == 6:
            duplicar_contenido(fd)

        elif op == 7:
            truncar_archivo(fd)

        elif op == 8:
            pass

# script principal...
if __name__ == '__main__':
    main()
```

La función *cargar\_nombre()* invocada al seleccionar la opción 1, es simple y directa: carga por teclado el nombre físico que se quiera utilizar desde ese momento, y lo retorna (con lo que a partir de ese momento, el valor de fd en la función *main()* cambia y con eso cambia también el nombre del archivo a utilizar en el resto del programa):

```
def cargar_nombre():
    fd = input('Ingrese nombre del archivo (con extensión si lo desea): ')
    print('Ok... nombre registrado...')
    print()
    return fd
```

La función *crear\_archivo(fd)* toma como parámetro el nombre del archivo que debe utilizar, y procede a crearlo si no existía, o a abrirlo limpiar su contenido si ya existía. En ambos casos, a continuación usa un ciclo para cargar por teclado una secuencia de cadenas de

caracteres, y graba una por una esas cadenas en el archivo. Como la función *input()* que se usa para la carga por teclado de una cadena, elimina el salto de línea del final, entonces al grabar la cadena en el archivo con *m.write()* hemos agregado el carácter '\n' en forma explícita:

```
def crear_archivo(fd):
    print('Archivo a crear:', fd)
    m = open(fd, 'wt')

    print('Ingrese líneas de texto, presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres "-." al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el último guión...
    line = line[:-1]

    # grabar la última linea...
    m.write(line + '\n')

m.close()
print()
print('Archivo creado y guardado...')
print()
```

Como la carga de datos se da por terminada cuando el usuario ingrese una cadena terminada con los caracteres '-.', entonces el ciclo de carga debe controlar que la cadena recién ingresada no termine con esos caracteres. El método *endswith(cad)* que viene incluido en toda variable de tipo cadena de caracteres permite chequear si la cadena con la que se invoca al método termina o no con los caracteres de la cadena *cad* tomada como parámetro, y ese método es el que se usa en el ciclo para controlar si debe continuar o detenerse.

Cuando esos caracteres aparecen en la última cadena *line* leída, el ciclo se detiene, pero esa última cadena debe todavía ser grabada en el archivo (el ciclo cortó antes de poder grabarla...) Para eso, se elimina el guión que la cadena tenía al final (*line = line[:-1]*), y se invoca a *write()* para grabarla, añadiendo el consabido '\n' al final.

La siguiente función es *abrir\_archivo(fd)*, la cual abre el archivo cuyo nombre toma como parámetro, pero en modo 'a+' para crearlo si no existía pero preservar su contenido si existía. Lo primero que esta función hace es leer línea por línea el contenido del archivo por medio de un **for iterador**. Note que antes de comenzar la lectura, se movió el *file pointer* al inicio del archivo (*m.seek(0, io.SEEK\_SET)*) ya que al abrir el archivo en modo 'a+' el *file pointer* queda ubicado al final, y eso haría que el ciclo de lectura no llegue a leer ninguna línea.

El resto de la función es similar a lo dicho para *crear\_archivo()*: se cargan por teclado nuevas líneas y se graban en el archivo. Al hacerlo se agregarán al final de ese archivo debido a que está abierto en modo 'a':

```
def abrir_archivo(fd):
    print('Archivo a abrir:', fd)
    m = open(fd, 'a+t')

    m.seek(0, io.SEEK_SET)
```

```

print('Contenido actual del archivo:')
print()
for line in m:
    print(line, end='')

print()
print()
print('Ingrese nuevas líneas de texto, para agregar al archivo,')
print('presionando <Enter> al final de cada una.')
print('Para terminar la carga, incluya los caracteres ".-" al final')
print()

line = input()
while not line.endswith('.-'):
    m.write(line + '\n')
    line = input()

# eliminar el guión del final...
line = line[:-1]

# grabar la última linea...
m.write(line + '\n')

m.close()
print()
print('Archivo modificado y guardado...')
print()

```

Sigue la función *listado\_completo(fd)*, que abre el archivo en modo de sólo lectura, y simplemente lo lee con un *for iterador* mostrando su contenido línea por línea. Como cada línea es recuperada del archivo **incluyendo** el carácter de salto de línea que tenía en ese archivo, entonces al mostrar cada cadena se agrega el parámetro *end=" "* en la invocación a *print()*, evitando con esto que se hagan dos saltos de línea en lugar de uno al visualizar la cadena:

```

def listado_completo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a mostrar:', fd)
    print()
    m = open(fd, 'rt')

    print('Contenido del archivo:')
    print()
    for line in m:
        print(line, end='')

    m.close()
    print()
    print()
    print('Archivo visualizado...')
    print()

```

La función *buscar\_cadena(fd)* abre el archivo y también lo lee línea por línea con un *for iterador*, pero ahora en lugar de mostrar cada cadena, invoca al método *count()* que esas cadenas contienen. Este método toma como parámetro otra cadena (aquí llamada *x*), y retorna la cantidad de veces que esa cadena *x* está contenida dentro de la cadena original con la cual se invoca al método. Los valores retornados por *count()* en cada vuelta del ciclo

se van acumulando en la variable *c*, y al finalizar el ciclo se muestra su valor (que será entonces la cantidad total de veces que *x* estaba en el archivo):

```
def buscar_cadena(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a controlar:', fd)
    m = open(fd, 'rt')

    c = 0
    x = input('Ingrese la cadena a buscar: ')
    for line in m:
        c += line.count(x)

    m.close()
    print()
    print('La cadena', x, 'fue encontrada', c, 'veces en el archivo')
    print()
```

La función *duplicar\_contenido(fd)* es realmente simple: abre el archivo, lee todo su contenido de una sola vez mediante el método *read()*, y la cadena así obtenida la graba de inmediato al final del archivo... con lo cual el archivo duplica su tamaño y contiene ahora una copia replicada de su contenido anterior. Como el archivo debe ser leído y también grabado, el modo de apertura elegido es '*r+*': al abrirlo, el *file pointer* apunta al inicio, el método *read()* lee desde allí todo el contenido y deja el *file pointer* al final, con lo que luego el método *write()* graba la cadena al final (no fueron necesarias en este caso, invocaciones a *seek()*):

```
def duplicar_contenido(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a expandir:', fd)
    m = open(fd, 'r+t')

    # leer el contenido completo...
    todo = m.read()

    # grabar ese contenido al final...
    m.write(todo)

    m.close()
    print()
    print('Se duplicó y grabó el contenido del archivo')
    print()
```

La última función de este programa se llama *truncar\_archivo(fd)*, y es la que lleva a cabo el requerimiento de *truncar* el contenido del archivo a una cantidad de líneas *cl* que se carga por teclado. La función abre el archivo en modo '*r+*', y luego usa un ciclo *while* para leer el archivo con la función *readline()*, controlando si se cargó la cadena vacía o no.

Por lo que hemos ya indicado, si *readline()* carga una cadena vacía, significa que se ha llegado al final del archivo. Además, con cada lectura se incrementa en 1 el contador *c*, de forma que si el valor de *c* llegase a ser igual a *cl*, se habrá leído entonces la cantidad de líneas que se quiere que el archivo finalmente contenga. En ese momento se corta el ciclo con *break*, y la función continúa controlando si efectivamente el ciclo cortó porque se leyeron *cl*

líneas: en ese caso, el *file pointer* está apuntando al byte donde específicamente queremos que termine el archivo. Ese valor (recuperado con *m.tell()*) se pasa como parámetro al método ***truncate()***, el cual corta el contenido del archivo haciendo que desde ese momento en adelante termine justamente en ese byte.

```
def truncar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a truncar:', fd)
    m = open(fd, 'r+t')

    c = 0
    cl = int(input('En cuántas líneas quiere truncar el archivo?: '))
    line = m.readline()
    while line != '':
        c += 1
        if c == cl:
            break
        line = m.readline()

    if c == cl:
        m.truncate(m.tell())
        print('Se truncó el contenido del archivo a', cl, 'lineas')
    else:
        print('No se truncó el archivo... no hay suficientes líneas...')

    m.close()
    print()
```

El programa completo se muestra a continuación:

```
import io
import os.path

__author__ = 'Cátedra de AED'

def cargar_nombre():
    fd = input('Ingrese el nombre del archivo (incluya si quiere una extensión): ')
    print('Ok... nombre registrado...')
    print()
    return fd

def crear_archivo(fd):
    print('Archivo a crear:', fd)
    m = open(fd, 'wt')

    print('Ingrese líneas de texto, presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres "-" al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el último guión...
    line = line[:-1]

    # grabar la última linea...
    m.write(line + '\n')
```

```
m.close()
print()
print('Archivo creado y guardado...')
print()

def abrir_archivo(fd):
    print('Archivo a abrir:', fd)
    m = open(fd, 'a+t')

    m.seek(0, io.SEEK_SET)
    print('Contenido actual del archivo:')
    print()

    for line in m:
        print(line, end='')

    print()
    print()
    print('Ingrese nuevas líneas de texto, para agregar al archivo,')
    print('presionando <Enter> al final de cada una.')
    print('Para terminar la carga, incluya los caracteres "-." al final')
    print()

    line = input()
    while not line.endswith('.-'):
        m.write(line + '\n')
        line = input()

    # eliminar el guión del final...
    line = line[:-1]

    # grabar la última linea...
    m.write(line + '\n')

m.close()
print()
print('Archivo modificado y guardado...')
print()

def listado_completo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a mostrar:', fd)
    print()
    m = open(fd, 'rt')

    print('Contenido del archivo:')
    print()
    for line in m:
        print(line, end='')

    m.close()
    print()
    print()
    print('Archivo visualizado...')
    print()

def buscar_cadena(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return
```

```
print('Archivo a controlar:', fd)
m = open(fd, 'rt')

c = 0
x = input('Ingrese la cadena a buscar: ')
for line in m:
    c += line.count(x)

m.close()
print()
print('La cadena', x, 'fue encontrada', c, 'veces en el archivo')
print()

def duplicar_contenido(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a expandir:', fd)
    m = open(fd, 'r+t')

    # leer el contenido completo...
    todo = m.read()

    # grabar ese contenido al final...
    m.write(todo)

    m.close()
    print()
    print('Se duplicó y grabó el contenido del archivo')
    print()

def truncar_archivo(fd):
    if not os.path.exists(fd):
        print('El archivo no existe... use las opciones 2 o 3 para crearlo...')
        print()
        return

    print('Archivo a truncar:', fd)
    m = open(fd, 'r+t')

    c = 0
    cl = int(input('En cuántas líneas quiere truncar el archivo?: '))
    line = m.readline()
    while line != '':
        c += 1
        if c == cl:
            break
        line = m.readline()

    if c == cl:
        m.truncate(m.tell())
        print('Se truncó el contenido del archivo a', cl, 'lineas')
    else:
        print('No se truncó el archivo... no hay suficientes líneas...')

    m.close()
    print()

def main():
    # nombre físico por default del archivo...
    fd = 'default.txt'

    op = 0
```

```

while op != 8:
    print('Opciones para gestión del archivo de texto:', fd)
    print(' 1. Cargar nombre físico del archivo')
    print(' 2. Crear archivo nuevo y grabar texto')
    print(' 3. Abrir archivo y agregar texto al final')
    print(' 4. Mostrar contenido completo del archivo')
    print(' 5. Buscar y contar una palabra o cadena en el archivo')
    print(' 6. Duplicar contenido del archivo (en el mismo archivo)')
    print(' 7. Truncar contenido del archivo a dos líneas')
    print(' 8. Salir')
    op = int(input('\t\tIngrese número de la opción elegida: '))
    print()

    if op == 1:
        fd = cargar_nombre()

    elif op == 2:
        crear_archivo(fd)

    elif op == 3:
        abrir_archivo(fd)

    elif op == 4:
        listado_completo(fd)

    elif op == 5:
        buscar_cadena(fd)

    elif op == 6:
        duplicar_contenido(fd)

    elif op == 7:
        truncar_archivo(fd)

    elif op == 8:
        pass

# script principal...
if __name__ == '__main__':
    main()

```

#### 4.] Procesamiento de archivos de texto que contienen secuencias numéricas.

En muchas ocasiones se trabaja con archivos de texto que contienen series de números expresados como cadenas. Típicamente en estos casos, cada línea contiene un "número" y el salto de línea separa a un número del siguiente. Normalmente, cuando se necesita procesar un archivo así, la idea es tomar las cadenas que representan números en el archivo, convertirlas a números en el formato correcto (*int* o *float*), almacenar esos números en un arreglo, y finalmente retornar el arreglo (u operar con él).

La razón por la cual alguien guardaría números como cadenas en un archivo, es la de evitar *problemas de incompatibilidad de formatos numéricos*, si se prevé que distintos programadores usen lenguajes diferentes para procesar el mismo archivo. El formato de texto también tiene diversas variantes, pero es siempre más sencillo compatibilizar formatos de texto que numéricos, sobre todo si el archivo de texto fue generado usando un juego de caracteres estándar (ASCII, UTF-8, etc.)

En el proyecto [F25] *Archivos de Texto* ya citado, se incluye el modelo *test04.py* en el cual aparecen dos funciones designadas como *save\_numbers(n, f)* y *read\_numbers(f)*. La primera toma como parámetro un número entero positivo *n* y el nombre *f* de un archivo, y graba en

el archivo  $f$  un total de  $n+1$  líneas de texto que representan números, pero como cadenas. La primera línea del archivo  $f$  es el propio valor de  $n$ , convertido a cadena de caracteres, a modo de indicador de la cantidad de números que contiene el archivo. Y las restantes líneas, son números aleatorios que pueden estar repetidos, pero de forma que todos pertenecen al intervalo  $[1, n]$  (ambos incluidos), y convertidos a cadenas de caracteres:

```
import random

def save_numbers(n, f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # crear el archivo de texto, en modo grabacion...
    m = open(f, 'w')

    # primera linea: la cantidad de numeros que tendrá el archivo...
    m.write(str(n) + '\n')

    # luego, n lineas: una por numero, en forma aleatoria en [1, n]...
    for i in range(n):
        # pedir un entero aleatorio entre 1 y n (incluidos ambos)
        x = random.randint(1, n)

        # grabar el numero como string, con salto de linea final...
        m.write(str(x) + '\n')

    m.close()
```

La segunda función (`read_numbers(f)`) abre el archivo  $f$  como archivo de texto en modo de solo lectura ( $f$  tiene por default el nombre '`lista.txt`'), lee la primera línea para saber cuántos números trae el archivo (y por añadidura, en qué rango vienen esos números...) y luego lee el resto de las líneas con un ciclo, elimina el salto de línea que cada una tenga al final, convierte la cadena resultante a un entero válido y agrega ese entero a un arreglo, por orden de llegada. Al finalizar el ciclo, la función retorna el arreglo:

```
def read_numbers(f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # el arreglo a retornar, inicialmente vacio...
    lista = []

    # abrir el archivo de texto en modo de solo lectura...
    m = open(f)

    # la primera linea del archivo nos da el valor de n...
    # ...sirva o no, leerla y eliminar el '\n' del final...
    ln = m.readline()
    ln = ln[:-1]

    # ...convertirla a int...
    n = int(ln)

    # leer el resto del archivo...
    while True:
        # leer el siguiente numero...
        xs = m.readline()
```

```

# controlar fin de archivo...
if xs == '':
    break

# si lo tenía, eliminar el caracter de salto de linea...
# ... y convertir a entero...
if xs[-1] == '\n':
    xs = xs[:-1]
x = int(xs)

# agregar el entero al arreglo en orden de llegada...
lista.append(x)

# cerrar el archivo y retornar el arreglo...
m.close()
return lista

```

El siguiente programa completo (modelo *test04.py*) contiene estas dos funciones y una simple función *test()* que las invoca, para mostrar finalmente la forma general de uso de ambas. La misma función *test()* es la que actúa como punto de entrada del programa:

```

import random

__author__ = 'Cátedra de AED'

def save_numbers(n, f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # crear el archivo de texto, en modo grabacion...
    m = open(f, 'w')

    # primera linea: grabar como string la cantidad de numeros que del archivo...
    m.write(str(n) + '\n')

    # luego n lineas: una por cada numero, generado en forma aleatoria en [1, n]...
    for i in range(n):
        # pedir un entero aleatorio entre 1 y n (incluidos ambos)
        x = random.randint(1, n)

        # grabar el numero convertido a string, con salto de linea al final...
        m.write(str(x) + '\n')

    m.close()

def read_numbers(f='lista.txt'):
    # asegurar el nombre por default del archivo...
    if not isinstance(f, str):
        f = 'lista.txt'

    # el arreglo a retornar, inicialmente vacio...
    lista = []

    # abrir el archivo de texto en modo de solo lectura...
    m = open(f)

    # la primera linea del archivo nos da el valor de n...
    # ...sirva o no, leerla y eliminar el '\n' del final...
    ln = m.readline()
    ln = ln[:-1]

```

```
# ...convertirla a int...
n = int(ln)

# leer el resto del archivo...
while True:
    # leer el siguiente numero...
    xs = m.readline()

    # controlar fin de archivo...
    if xs == '':
        break

    # si lo tenia, eliminar el caracter de salto de linea...
    # ... y convertir a entero...
    if xs[-1] == '\n':
        xs = xs[:-1]
    x = int(xs)

    # agregar el entero al arreglo en orden de llegada...
    lista.append(x)

# cerrar el archivo y retornar el arreglo...
m.close()
return lista

def test():
    save_numbers(40)
    lis = read_numbers()
    print('La lista leida es:')
    print(lis)
    print('Cantidad de elementos:', len(lis))

if __name__ == '__main__':
    test()
```

---

## Bibliografía

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

# Ficha 26

## Estrategias de Resolución de Problemas: Recursividad

### 1.] Introducción.

Cuando en la práctica se habla de *recursividad*, se está haciendo referencia a una muy particular forma de expresar la definición de un objeto o un concepto. Esencialmente, una definición se dice *recursiva* si el objeto o concepto que está siendo definido *aparece a su vez en la propia definición*. Consideremos por ejemplo, la siguiente definición:

*Una frase es un conjunto de palabras que puede estar vacío, o bien puede contener una palabra seguida a su vez de otra frase.*

Estamos frente a una definición recursiva, puesto que el objeto definido (*la frase*) se está usando en la misma definición, al indicar que una frase puede ser *una palabra seguida a su vez de otra frase*.

Al plantear una definición recursiva debe tenerse cuidado con los siguientes elementos [1]:

1. Una definición recursiva correctamente planteada, exige que la definición *agregue conocimiento* respecto del concepto u objeto que se define. No basta con que el objeto definido aparezca en la definición, pues si sólo nos limitamos a esa regla podrían producirse definiciones obviamente verdaderas, pero sin aportar conocimiento alguno. Por ejemplo, si decimos que:

*Una frase es una frase.*

no cabe duda en cuanto a que esa afirmación es recursiva, pero de ninguna manera estamos definiendo lo que *es* una frase. En todo caso, lo que tenemos es una *identidad*, y no una *definición*. En cambio en la definición original que dimos de la idea de *frase*, encontramos elementos que nos permiten construir paso a paso el concepto de *frase*, a partir de la noción de *frase vacía* y la noción de *palabra*, y esos elementos son los que agregan conocimiento al concepto.

2. Por otra parte, una definición recursiva correctamente planteada debe evitar la *recursión infinita* que se produce cuando en la definición no existen elementos que permitan cerrarla lógicamente. Se cae así en una definición que, aunque agrega conocimiento, no termina nunca de referirse a sí misma. Por ejemplo, si la definición original de *frase* fuera planteada así:

*Una frase es un conjunto que consta de una palabra seguida a su vez de una frase.*

tenemos que esta nueva definición es recursiva y aparentemente está bien planteada, por cuanto agrega conocimiento al indicar que una frase consta de palabras y frases. Pero al no indicar que una frase puede estar vacía, la noción de frase se torna en un concepto *sin fin*: cada vez que decimos *frase*, pensamos en una palabra, y en otra *frase*, lo cual a su vez lleva a otra palabra y a una nueva *frase*, y así sucesivamente, sin solución de continuidad.

Las definiciones recursivas no son muy comunes en la vida cotidiana porque en principio, siempre existe y siempre resulta más simple pensar y entender una definición directa, sin la vuelta hacia atrás que supone la recursividad. La misma noción de frase, puede definirse sin recursividad de la forma siguiente:

*Una frase es una sucesión finita de palabras, que puede estar vacía.*

y esta definición resulta más natural y obvia (incluso podría no indicarse que la frase puede estar vacía, y la definición seguiría teniendo sentido).

Sin embargo, en ciertas disciplinas como la Matemática, la recursividad se usa con mucha frecuencia debido a que existen problemas cuya descripción simbólica es más compacta y consistente con recursividad que sin ella<sup>1</sup>. Consideremos el típico y ya conocido caso del *factorial* de un número  $n$ . Como sabemos, si  $n$  es un entero positivo o cero, entonces el *factorial* de  $n$  (denotado como  $n!$ ), se puede definir sin recursividad, como sigue:

$$n! = \begin{cases} \text{si } n = 0 \Rightarrow 0! = 1 \\ \text{si } n > 0 \Rightarrow n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1 \end{cases}$$

Es decir: si  $n$  es cero, su factorial vale uno. Pero si  $n$  es mayor a cero, su factorial es el producto de  $n$  por todos los enteros positivos anteriores a  $n$ , hasta el uno. De esta forma, el factorial de 4 sería:

$$\begin{aligned} 4! &= 4 * 3 * 2 * 1 \\ 4! &= 24 \end{aligned}$$

La fórmula dada para el cálculo del factorial es sencilla, pero incluye una serie de puntos suspensivos que no siempre son bien recibidos, pues se presupone que quien lee la fórmula será capaz de deducir por sí mismo la forma de llenar el espacio de los puntos suspensivos. Eso podría no ser tan simple si la fórmula fuese más compleja.

La recursividad ofrece una alternativa de notación más compacta, evitando los puntos suspensivos. Por ejemplo, si se observa la definición para  $n > 0$ , tenemos:

$$n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$$

En esta fórmula es claramente visible que la expresión  $(n - 1) * (n - 2) * \dots * 3 * 2 * 1$  es equivalente al factorial del número  $(n - 1)$ : se está multiplicando a  $(n - 1)$  por *todos los números enteros anteriores a él, hasta llegar al uno*. Por lo tanto, la definición original podría escribirse así:

$$n! = n * (n - 1)!$$

---

<sup>1</sup> La idea de un concepto o un proceso que recurre a sí mismo para completarse puede aplicarse a la extraña situación de un sueño dentro de un sueño, que fue utilizada ya por Edgar Allan Poe en 1849 en su poema "A Dream Within a Dream". Este poema a su vez, inspiró en 1976 una composición musical instrumental de la banda The Alan Parsons Project también llamada "A Dream Within a Dream". Y en el cine, la referencia es obvia: la película *Inception* (conocida como *El Origen* en Hispanoamérica) del año 2010, dirigida por Christopher Nolan y protagonizada por Leonardo DiCaprio, narra una atrapante historia de un equipo de espías industriales que navega en tres niveles de sueños recursivos!

En otras palabras: si  $n > 0$ , entonces el *factorial* de  $n$  es igual a  $n$  multiplicado por el *factorial* de  $(n - 1)$ . Si reunimos todo en una definición global, tenemos:

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

y llegamos así a una *definición recursiva*: para definir el *factorial* de  $n$ , en la misma definición se recurre al *factorial* de  $(n - 1)$ . Observemos que la condición

$$\text{si } n = 0 \text{ entonces } 0! = 1$$

es la que evita la *recursión infinita*, y que la condición

$$\text{si } n > 0 \text{ entonces } n! = n * (n - 1)!$$

provoca el paso recursivo, pero agregando conocimiento al mismo: un *factorial* es en última instancia un *producto* de un número por el *factorial del número precedente*.

## 2.] Programación recursiva.

La noción de definición recursiva vista hasta aquí sirve como punto de partida para plantear *algoritmos* en forma recursiva (de hecho, la definición recursiva del factorial que se estudió, no es más que un *algoritmo recursivo* para calcular ese factorial).

Prácticamente todos los lenguajes de programación modernos soportan la recursividad, a través del planteo de *subrutinas recursivas*. En Python, la idea es que un algoritmo recursivo puede implementarse a través de *funciones de comportamiento recursivo* [2]. En términos muy básicos, una *función recursiva* es una función que incluye en su bloque de acciones *una o más invocaciones a sí misma*.

En otras palabras, una función recursiva es aquella que se invoca a sí misma una o más veces. En esencia, entonces, la siguiente función es recursiva (aunque aclaramos que está *mal planteada*):

```
def procesar():
    procesar()
```

Si la condición para ser recursiva es invocarse a sí misma, entonces la función anterior cumple el requisito: de hecho, *lo único* que hace es invocarse a sí misma. Sin embargo, como ya se dijo, está mal planteada: aún sin saber mucho respecto de cómo trabaja la recursividad en Python, un breve análisis inmediatamente permite deducir que una vez invocada esta función provoca una *cascada infinita* de auto-invocaciones, sin realizar ninguna tarea útil. Como ya veremos, este *proceso recursivo infinito* provocará tarde o temprano una falla de ejecución por falta de memoria, y el programa se interrumpirá.

¿Por qué está mal planteada esta función? En realidad ya conocemos la respuesta: cuando se introdujo a nivel teórico el tema de las definiciones recursivas bien planteadas en la sección anterior, se indicó que estas deberían *agregar conocimiento* respecto del concepto definido, y evitar la *recursión infinita* incluyendo una condición que corte el proceso recursivo. Y bien: la función *procesar()* aquí planteada no incluye ninguno de los dos elementos. por lo que no corresponde a una definición recursiva bien planteada [2].

Para poner un ejemplo conocido, supongamos que se desea plantear una función recursiva para calcular el factorial de un número  $n$  que entra como parámetro. Si planteáramos una función *factorial(n)* para calcular *recursivamente* el factorial de  $n$ , pero lo hiciéramos a la manera incorrecta que vimos antes para *procesar()*, quedaría:

```
def factorial(n):
    return factorial(n)
```

La función *factorial* así planteada, correspondería a "definir" el factorial así:

*El factorial de n es igual al factorial de n.*

Y como se ve, ni la definición ni la función agregan conocimiento al concepto y son por lo tanto, incorrectas. Un segundo intento, sería:

```
def factorial(n):
    return n * factorial(n - 1)
```

En este caso, la función mostrada correspondería a definir al factorial de la siguiente forma:

$$n! = n \cdot (n - 1)!$$

lo cual agrega conocimiento pero cae en *recursión infinita*, pues la invocación *factorial(n-1)* provoca una nueva llamada, y esta a su vez otra, y así en forma continua, sin definir en qué momento detener el proceso.

La *recursión infinita* se evita considerando que si  $n == 0$ , entonces el factorial de  $n$  es 1. Una simple condición en la función y queda la versión final, ahora correcta:

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

### 3.] Seguimiento de la recursividad.

En el punto anterior vimos que una función recursiva bien planteada debe contener una *condición de corte* para evitar la recursión infinita, e incluir una o mas invocaciones a sí misma pero agregando algún tipo de explicitación del proceso. Ahora bien... ¿Cómo funciona todo esto? ¿Cómo hace un lenguaje de programación para ejecutar una función recursiva? Una persona poco entrenada en programación recursiva podría creer que la función *factorial()* que planteamos antes está incompleta (porque no incluye ningún ciclo para calcular el factorial), o asumir que es correcta pero no comprender nada en absoluto respecto de su funcionamiento.

Para entender como trabaja una función recursiva, lo mejor es intentar un seguimiento a partir de un esquema gráfico [1]. Supongamos que se invoca a la función *factorial()* para calcular el factorial del número 4. O sea, supongamos la siguiente invocación:

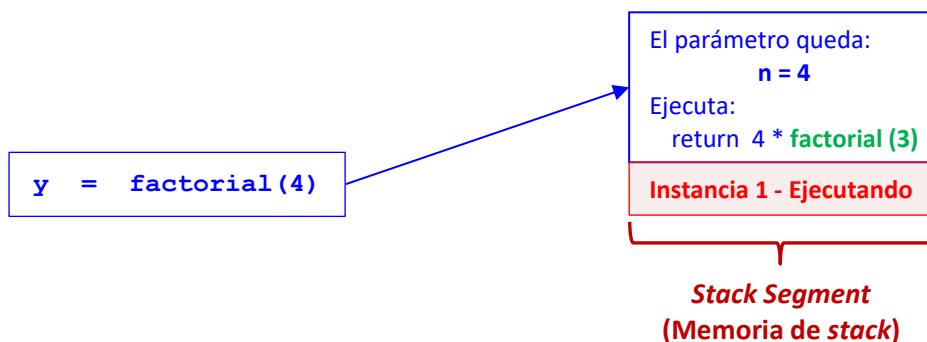
$$y = \text{factorial}(4)$$

Cuando una función es invocada (sea o no recursiva) automáticamente se asigna para ella un bloque de memoria en el segmento de memoria conocido como *Stack Segment* (o *Segmento de Pila*). Dentro de ese bloque, *la función crea y aloja sus parámetros formales y*

variables locales y luego comienza a ejecutarse. Se dice que se está ejecutando una *instancia* de la función (y en este caso, es la *primera instancia*).

Recordando que la función `factorial()` toma como parámetro una variable *n*, en la siguiente figura representamos con un rectángulo al bloque de memoria asignado a la función en esta *primera instancia de ejecución*:

Figura 1: Primera invocación a la función `factorial()`.



El valor 4 enviado como parámetro actual, es asignado en el parámetro formal *n*. Luego, la función verifica si *n* vale cero. En este caso, la condición sale por falso y ejecuta la rama *else*, que expresa:

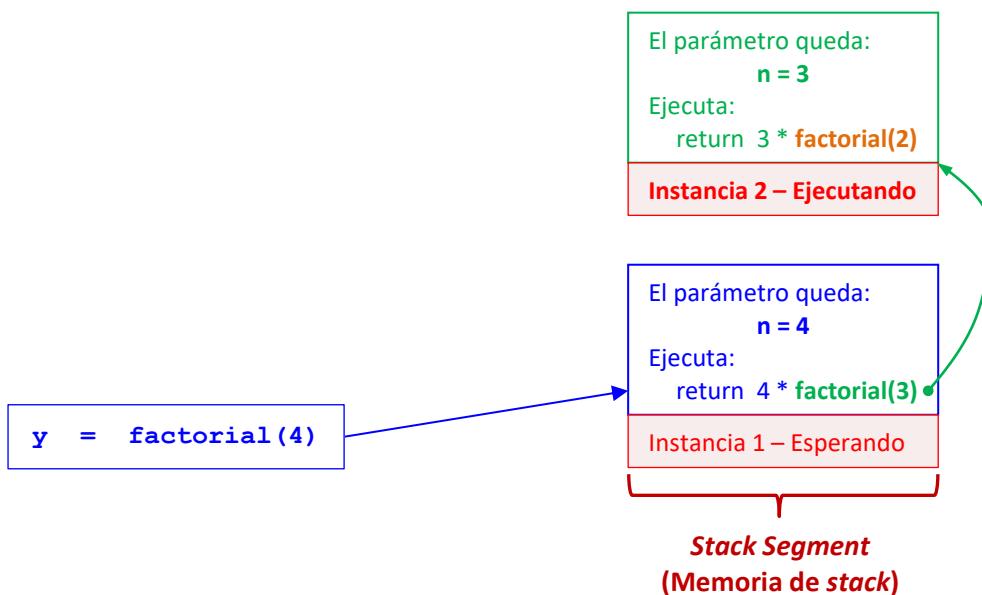
```
return n * factorial(n - 1)
```

Como *n* vale 4, la expresión se evalúa como

```
return 4 * factorial(3)
```

Ahora bien: en esta última expresión se está invocando nuevamente a la función `factorial()`, pero ahora para calcular el factorial de 3. *Aquí se produjo una llamada recursiva* y el lenguaje Python actúa frente a ella de manera sencilla: trata a esa invocación recursiva como trataría a cualquier invocación normal de función: simplemente, le asigna a esa *segunda instancia* de ejecución una *nueva área* de memoria de *stack*, para que a su vez esta segunda instancia aloje en ella sus variables locales y parámetros formales. Nuevamente, mostramos un rectángulo para representar a la segunda instancia:

Figura 2: Segunda invocación (recursiva) a la función `factorial()`.



Lo importante aquí es entender que al asignar memoria para la *segunda instancia de ejecución*, la función *vuelve a crear* sus variables locales y parámetros formales, sin interferir con los ya creados para la *primera instancia*. El área de memoria de stack asignada a la *primera instancia* sigue ocupada, pero momentáneamente inactiva (la *primera instancia* de ejecución está esperando a que la *segunda* finalice).

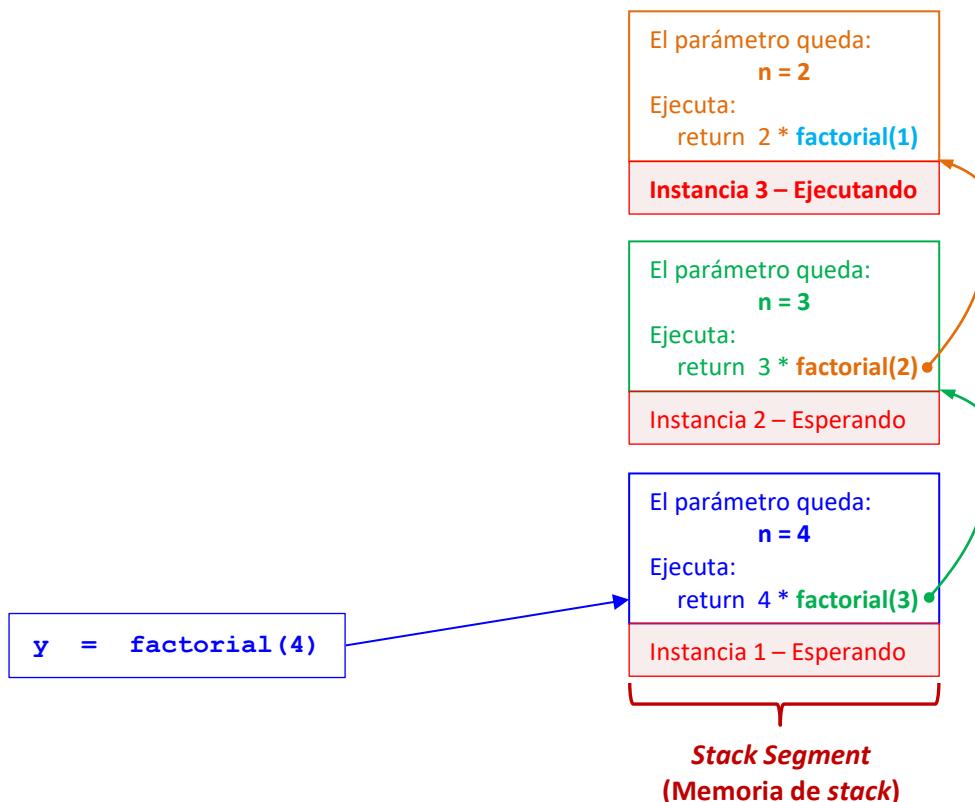
En otras palabras: la *segunda instancia* se comporta (y de hecho, *es*) como una función *diferente* de la primera, y cada una de ellas tiene *sus propias variables*. Aún cuando estas variables tienen los mismos nombres o identificadores en ambas instancias de ejecución, son variables diferentes *porque ocupan lugares distintos de la memoria*. El bloque de memoria de stack de la *primera instancia*, contiene una variable *n* cuyo valor es *4*, y el bloque de memoria de stack de la *segunda instancia* contiene *otra* variable (también llamada *n*), pero con valor igual a *3*. Mientras está activa (o sea, mientras se está ejecutando) la *segunda instancia*, la variable que se usa es la *n* cuyo valor es *3*.

Cuando se ejecuta la *segunda instancia*, se repite el esquema que ocurrió en la *primera*, pero ahora con *n* valiendo *3*. La expresión:

```
return 3 * factorial(2)
```

provoca a su vez *otra llamada recursiva*, que es alojada en *otra* área de memoria de memoria de stack. Esta *tercera instancia de ejecución* produce además una nueva creación de variables locales:

**Figura 3: Tercera invocación (recursiva) a la función *factorial()*.**



La *tercera instancia* lanza una nueva llamada recursiva y otra asignación de memoria de stack para la *cuarta instancia*, al hacer:

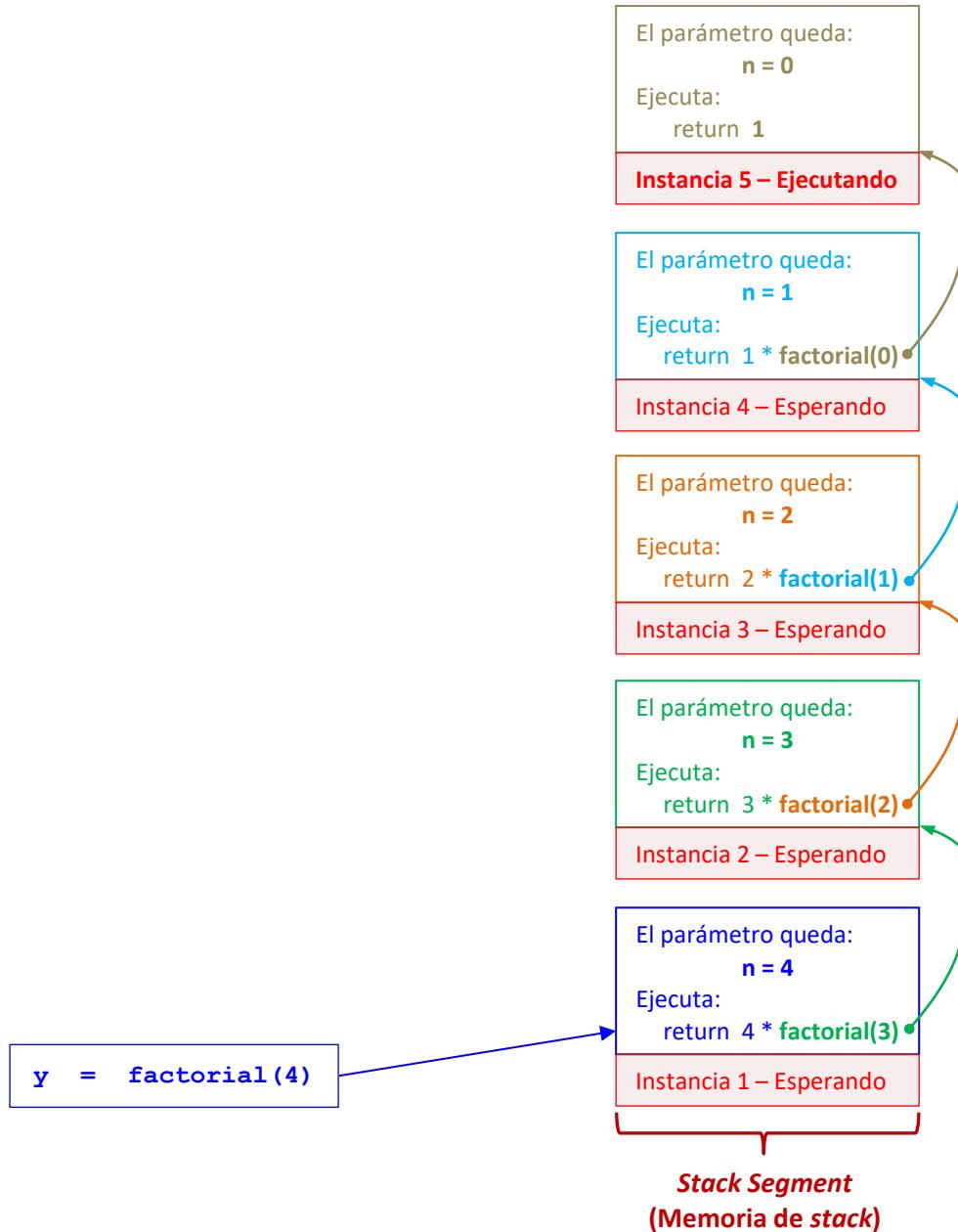
```
return 2 * factorial(1)
```

Y la *cuarta instancia* a su vez lanza una *quinta instancia* haciendo:

```
return 1 * factorial(0)
```

con lo que nuestro esquema de memoria de stack queda así:

**Figura 4: Memoria de stack luego de cinco invocaciones recursivas a la función `factorial()`.**



Observemos con mucha atención la *quinta instancia de ejecución*: en ella, el valor de *n* es *cero*, y por lo tanto en esta *quinta instancia* la condición:

```
if n == 0:
    return 1
```

es evaluada en *verdadero*, ejecutando entonces la instrucción *return 1*. Entiéndase bien: hasta aquí, había en memoria *cinco instancias* ejecutándose y/o esperando. Si bien se trataba de la misma función invocándose a sí misma varias veces, Python (y todo lenguaje

que soporte recursividad) trata a estas instancias como funciones diferentes que piden memoria por separado.

Ahora bien: cuando alguna de las funciones de esa *cascada recursiva* logra finalizar (como pasa con la *quinta instancia* de nuestro último gráfico), comienza a desarrollarse automáticamente un proceso conocido como *proceso de vuelta atrás* o *backtracking*.

La cuestión es simple, aunque un poco extensa de explicar: la función que logra finalizar libera su área de memoria de stack y retorna su valor hacia la instancia de ejecución que originalmente la había llamado. En nuestro caso, la *instancia 5* retorna el valor **1** (que es el valor del *factorial de 0*) hacia la *instancia 4* (que es la que había pedido el *factorial de 0*).

En la *instancia 4* ahora se conoce entonces cuánto vale el *factorial de 0*, por lo cual a su vez puede calcular el valor de  $1 * \text{factorial}(0)$  que es igual a **1**. Cuando la *instancia 4* calcula este valor (que es el *factorial de 1*), la *instancia 4* termina de ejecutarse retornando el valor **1** a la *instancia 3* (que fue la que pidió el *factorial de 1*).

En la *instancia 3* se calcula ahora el *factorial de 2*, haciendo  $2 * \text{factorial}(1)$  cuyo valor es **2**. Ese resultado se retorna a la *instancia 2*, en donde a su vez se calcula  $3 * \text{factorial}(2)$  que vale **6**.

Por fin, el valor **6** es retorna a la *instancia 1*, donde se calcula el valor  $4 * \text{factorial}(3)$  cuyo valor es **24** y es justamente el *factorial de 4* que se quería calcular originalmente. Ese resultado es devuelto al *punto original de llamada*, y la función *factorial()* termina (¡por fin!) de ejecutarse. Gráficamente, el *proceso de vuelta atrás* o *backtracking* completo puede verse en la *Figura 5* (página 521).

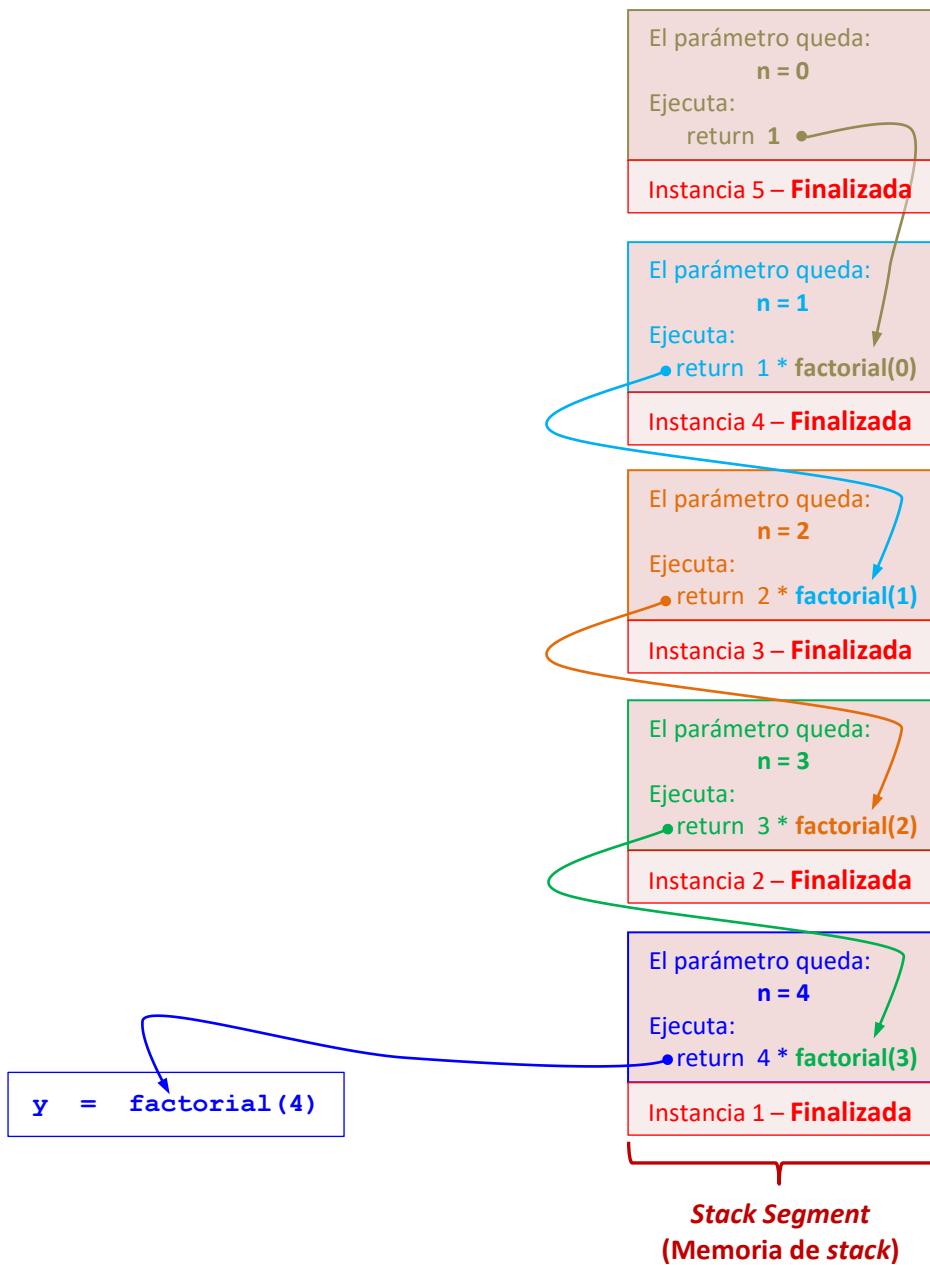
Lo importante de todo esto es que tanto el proceso de asignación de memoria de stack al comenzar el proceso recursivo hacia adelante (o *cascada recursiva*), como el proceso de *vuelta atrás* o *backtracking*, son gestionados en forma automática por el programa. Lo único que debe hacer el programador es *plantear en forma correcta la función*, lo cual como ya vimos, implica incluir una condición de corte y efectuar una o mas llamadas recursivas que de alguna forma vayan reduciendo poco a poco el proceso. En nuestro caso, al llamar a *factorial()* enviando como parámetro el valor  $n - 1$ , se va reduciendo el valor con el cual se trabaja hasta que en alguna instancia ese valor será cero y comenzará el proceso de *vuelta atrás*.

#### 4.] Aplicación elemental de la recursión.

El proceso recursivo implica una cascada de invocaciones a nuevas instancias de la misma función, y luego otro proceso de regreso o vuelta atrás mediante el cual se van cerrando los cálculos que estuviesen pendientes en instancias anteriores.

El hecho es que no siempre resulta evidente la forma en que ocurre ese proceso de vuelta atrás, sobre todo debido a que el mismo es automático e implícito. Pero si el programador tiene en claro el mecanismo, puede aprovecharlo para lograr resolver problemas que de otro modo parecerían muy complicados. Vemos un ejemplo simple pero muy ilustrativo: Supongamos que se desea implementar una función que simplemente tome como parámetro un número  $n$ , y muestre  $n$  mensajes en la consola de salida, pero numerados desde 1 hasta  $n$ . Supongamos también que se nos pide que la función lo haga en forma recursiva, sin usar un ciclo.

Figura 5: Proceso de vuelta atrás completo y finalización de la ejecución.



Un primer intento podría verse como sigue:

```
def mostrar01(n):
    if n > 0:
        print('Mensaje numero', n)
        mostrar01(n-1)
```

Evidentemente, si  $n$  es cero o menor que cero, la función no tiene nada que hacer (ya que se están pidiendo "cero mensajes"). Por lo tanto, esa es la *situación trivial o base* que define la condición de corte: la función sólo llevará a cabo algún proceso si  $n > 0$ . Si  $n$  es mayor que cero, se muestra el primer mensaje incluyendo en él el valor actual de  $n$ , **y luego se activa la recursión**: se vuelve a invocar a la función *mostrar01()* pero ahora pasándole como parámetro el valor  $n-1$ . Al reducir en 1 el valor del parámetro en cada invocación, se garantiza que en algún momento se alcanzará el valor 0 y la cascada recursiva finalizará.

Todo parecería correcto... pero el estudiante quizás ya notó que hay un problema: así planteada, la función mostrará efectivamente una secuencia de  $n$  mensajes, pero numerados en *forma descendente* (de  $n$  hacia 1) en lugar de hacerlo en *forma ascendente* (que era lo pedido). Si  $n$  es 5, por ejemplo, la salida producida por esta función será:

```
Mensaje numero 5
Mensaje numero 4
Mensaje numero 3
Mensaje numero 2
Mensaje numero 1
```

En efecto: cada instancia de ejecución de la función chequea si  $n$  es mayor a 0. Cuando  $n$  vale 5 la condición es cierta, y en la rama verdadera la primera instrucción muestra el valor actual de  $n$ , **antes** de la invocación recursiva. Por lo tanto, era de esperar que el primer mensaje mostrado se numere como 5 y no como 1. En cada una de las otras instancias de ejecución ocurre lo mismo: primero se muestra el valor del parámetro  $n$ , y **luego** de lanza la recursión para los mensajes que quedan.

¿Cómo se podría solucionar el problema y hacer que los mensajes salgan numerados de menor a mayor? Sólo debemos recordar que cada vez que se invoca a la función, se almacena en la memoria de stack el valor del parámetro  $n$ . No es obligatorio mostrar el valor de  $n$  inmediatamente al entrar en la rama verdadera de la condición: **se puede hacer primero la llamada recursiva**, ir almacenando con eso en el stack la sucesión de valores (sin mostrarlos), y mostrarlos sólo cuando finaliza la cascada recursiva y a medida que se regresa con el proceso de vuelta atrás:

```
def mostrar02(n):
    if n > 0:
        mostrar02(n-1)
        print('Mensaje numero', n)
```

Por curioso que parezca, esta nueva función hace lo pedido y muestra los mensajes numerados en forma ascendente. Lo único que tuvimos que hacer fue *invertir el orden de las instrucciones de la rama verdadera*. Como cada vez que se invoca a la función se almacena en la memoria de stack el valor actual de  $n$ , pero estos valores se almacenan de forma que el último en llegar es el que queda disponible en la cima del stack, entonces queda en la cima el valor 1 (puede ver la *Figura 4* para recordar la forma en que se van generando las instancias de ejecución).

Como en nuestra segunda versión las llamadas recursivas se hacen antes que las visualizaciones, el resultado es que **ninguna** visualización se hará hasta que todas las llamadas recursivas queden almacenadas en el stack... y cuando eso ocurra, la primera instrucción *print()* en ejecutarse será la que corresponda a la cima del stack que es la que contiene al valor  $n = 1$ . El proceso de vuelta atrás hará que se vayan mostrando los valores desde el stack en orden inverso al de su llenado, y eso es lo que hace que los valores se muestren en orden ascendente.

La primera función *mostrar01()* que hemos analizado, tenía una estructura conocida como *procesamiento en pre-orden* o *procesamiento en orden previo*, que se caracteriza por el hecho de que el proceso que debe realizar la función (en este caso, el *print()*), **se hace antes que la invocación recursiva**. Por otra parte, la función *mostrar02()* tiene una estructura que se conoce como *procesamiento en post-orden* o *procesamiento en orden posterior*: el

proceso a realizar (nuevamente la función *print()*) se hace *después de hacer la invocación recursiva*.

Para finalizar, un detalle: el segmento de memoria que hemos llamado *Stack Segment* se designa con ese nombre por una razón de peso: la palabra en inglés *stack* se traduce como *pila* en español y ese nombre se debe a que en ese segmento los valores que se almacenan van *apilándose* uno sobre otro a medida que se van invocando funciones (con recursión o no) (vuelva a ver la *Figura 4* para observar este efecto). Y como ya vimos en una Ficha anterior, esta forma de almacenar datos se conoce como *esquema de Pila* o de tipo *LIFO* por sus iniciales en inglés: *Last In – First Out (Primero en llegar – Último en salir)*.

El *Stack Segment* es el segmento de soporte para todos los procesos de invocaciones a funciones de cualquier lenguaje de programación. Cada vez que se invoca a una función, se reserva para ella un espacio en la cima del stack y en ese espacio la función almacena la dirección a la que se debe volver cuando finalice su ejecución, y sus variables locales. Y cuando una función finaliza, se libera el espacio de stack que la misma tenía, quedando en la cima la función desde la cual se invocó a la que acaba de terminar.

Y es natural que funcione como un *apilamiento* de datos: cuando una función termina, el flujo de ejecución del programa debe regresar al punto desde donde fue invocada la que acaba de terminar, y es justamente esa la que en ese momento estará en la cima. El *Stack Segment* constituye así un mecanismo confiable para que el programa recuerde el camino de regreso que debe seguir a medida que las funciones que se invocaron vayan finalizando.

## 5.] Consideraciones generales.

Llegado a este punto quizás resulte obvio que la recursividad es un proceso algo complicado de entender para quienes recién se acercan a ella. Sin embargo es también cierto que después de un poco de práctica el proceso se asimila sin inconvenientes, sobre todo si en ese período de práctica el lector se toma el trabajo de hacer un seguimiento gráfico de las funciones recursivas que plantea.

Uno de los puntos que más trabajo parece costar a los recién iniciados es el *planteo de la condición de corte* de la función (es decir, la condición para evitar la *recursión infinita*). La clave de este paso es tener en cuenta que lo que se busca determinar con esa condición es si se ha presentado lo que se conoce como un *caso base* o un *caso trivial* para el problema: un caso en el que la recursión no es necesaria y puede resolverse en forma directa (o incluso no haciendo nada) [1].

La determinación de esa condición de corte puede hacerse sin mayores problemas, simplemente respondiendo a la siguiente pregunta:

*¿En qué caso o casos el problema que se quiere plantear se resuelve en forma trivial?*

La respuesta a esa pregunta, aplicada al problema particular que se está enfrentando evidencia la condición de corte que se buscaba. En el caso del cálculo del factorial, la pregunta sería: ¿en qué caso el factorial de  $n$  se resuelve en forma trivial, sin necesidad de cálculos ni procesos recursivos? Es obvio que ese caso se da cuando  $n$  vale cero, pues entonces el factorial vale directamente 1 (uno). Y en nuestra versión recursiva del factorial hemos incluido entonces una condición de corte que comprueba si  $n$  es 0.

Por otra parte, debe observarse que la recursividad es una herramienta para usar con cautela [3] [4]: si bien es cierto que una función recursiva es extremadamente compacta en cuanto a código fuente y permite además que una definición recursiva sea llevada a una función recursiva prácticamente sin cambios (obsérvese la gran similitud que existe entre la definición algebraica recursiva del factorial y el planteo de la función para calcular el factorial), también es cierto que *la recursividad insume más memoria que la que usaría un proceso cíclico común*.

Esto se debe a que cada *instancia recursiva pide memoria de stack* para esa instancia (como se vio en los gráficos anteriores). Si la *cascada recursiva* fuera muy larga (por ejemplo, para calcular el factorial de un número grande), podría llegar a provocarse un *rebalsamiento, desborde o sobreflujo (overflow)* de memoria de stack. Esto significa que alguna instancia de la cascada de invocaciones *podría no tener lugar en el Stack Segment* para ejecutarse, con lo cual se produciría la interrupción o cancelación del programa que invocó a esa función. La memoria de un computador es un recurso de tamaño limitado, y el *Stack Segment* es parte de la memoria.

Por otra parte, la ejecución de una función requiere *cierto tiempo de procesamiento* desde que esta comienza y hasta que finaliza, que debe contemplarse en la estimación del tiempo total de ejecución de un proceso recursivo completo. En el caso del factorial, la versión *iterativa* (no recursiva, basada en ciclos) tendrá un tiempo de ejecución en relación directa con el tiempo que demore el ciclo *for* en finalizar, y este ciclo depende a su vez del valor de  $n$ . Por otra parte, la versión *recursiva* hará tantas invocaciones recursivas como sea el valor de  $n$ , y el tiempo total dependerá del tiempo que lleve terminar de ejecutar cada una. En este caso, se puede asumir que en ambos escenarios se tendrá un *tiempo de ejecución que depende de  $n$  en forma directa*, y por lo tanto serían *igualmente aceptables* la versión recursiva y la versión iterativa.

Notemos además que el código fuente de la versión recursiva es simple, directo y compacto. La *complejidad aparente y la estructura del código fuente* de un programa o función es otro de los elementos que se tienen en cuenta para hacer un análisis comparativo entre dos o más soluciones propuestas para un mismo problema. Pero en el caso del factorial, la versión iterativa no es mucho más compleja que la recursiva, aunque es cierto que la recursiva es más directa para comprender.

Si se tienen en cuenta los tres factores generales de análisis comparativo entre algoritmos (*consumo de memoria, tiempo de ejecución y complejidad de código fuente*) entonces si un programador debe realizar al cálculo del factorial de  $n$ , debería usar la versión *iterativa* (y *no la recursiva*). La decisión final de usar la versión *iterativa*, *se basa en este caso en el uso más eficiente de la memoria por parte de la versión iterativa*. El *tiempo de ejecución final está en el mismo orden de magnitud en ambos casos* y *la complejidad del código fuente es aceptable también en ambos casos*.

Por estos motivos, en general *se suele* aconsejar usar la recursividad *sólo cuando sea absolutamente necesario por la naturaleza implícitamente recursiva del problema que se enfrenta*, como es el caso del recorrido de ciertas estructuras de datos como los *árboles* o los *grafos* (que escapan a los alcances de este curso), o la generación de gráficos de *naturaleza fractal* (figuras que se forman combinando versiones más sencillas de las mismas figuras...) que resultaría prácticamente imposible de programar sin recursión desde el punto de vista de la *complejidad del código fuente*.

Algunos programadores experimentados a veces plantean primero la solución recursiva de un problema (para darse una idea de la forma mínima de resolverlo desde el punto de vista de la *complejidad del código fuente*), y luego tratan de convertir ese planteo a una solución no recursiva basada en ciclos (aunque esto no siempre es sencillo de hacer...)

## 6.] Caso de análisis: La Sucesión de Fibonacci.

Para permitir un mayor dominio de las técnicas de programación recursiva, y sus ventajas y desventajas, analizaremos ahora un conocido problema: el cálculo del término *n*-ésimo de la *Sucesión de Fibonacci*.

**Problema 59.)** *La Sucesión de Fibonacci es una secuencia de valores naturales en la que cada término es igual a la suma de los dos anteriores, asumiendo que el primer y el segundo término valen 1. En algunas fuentes se parte de suponer que los dos primeros valen 0 y 1 respectivamente, pero eso no cambia el espíritu de la regla ni la explicación que sigue. Formalmente, el cálculo del término n-ésimo  $F(n)$  de la sucesión se puede entonces definir así (y note que la definición planteada es directamente recursiva):*

Término *n*-ésimo de Fibonacci

$$F(n) \quad \begin{cases} = 1 & \text{(si } n = 1 \text{ ó } n = 0\text{)} \\ = F(n-1) + F(n-2) & \text{(si } n > 1\text{)} \end{cases}$$

*Se pide desarrollar un programa que permita calcular el valor del término enésimo de esta sucesión, cargando *n* por teclado.*

**Discusión y solución:** Si bien el enunciado muestra la definición en forma directamente recursiva, podemos diseñar un par de funciones que calculen el valor del término *n*-ésimo de Fibonacci: una en forma iterativa y la otra en forma recursiva para luego poder comparar ambas soluciones (vea el proyecto *F[26] Recursión* que acompaña a esta ficha):

```
# versión iterativa
def fibonacci01(n):
    ant2 = ant1 = 1
    for i in range(2, n + 1):
        aux = ant1 + ant2
        ant2 = ant1
        ant1 = aux
    return ant1

# versión recursiva
def fibonacci02(n):
    if n <= 1:
        return 1
    return fibonacci02(n - 1) + fibonacci02(n - 2)
```

La *versión iterativa* inicializa las variables *ant2* y *ant1* con el valor 1, y usa un *for* para recorrer el intervalo  $[2, n]$ . En cada repetición suma los valores de *ant1* y *ant2* para obtener el siguiente término en forma progresiva, actualizando también los valores de *ant1* y *ant2*

para quedarse siempre con los dos últimos calculados. Cuando el *for* finaliza, el último número almacenado en *ant1* es el que corresponde al término  $n$ -ésimo pedido.

La *versión recursiva* es la aplicación directa de la fórmula dada en la definición: la condición de corte comprueba si  $n$  es menor o igual a 1. En caso afirmativo, se asume que  $n$  vale 0 o vale 1 y en ese caso, se retorna 1 como se pide en la definición. Si  $n$  fuese mayor a 1, se aplica la fórmula en forma recursiva y se retorna la suma de los dos anteriores a  $n$ .

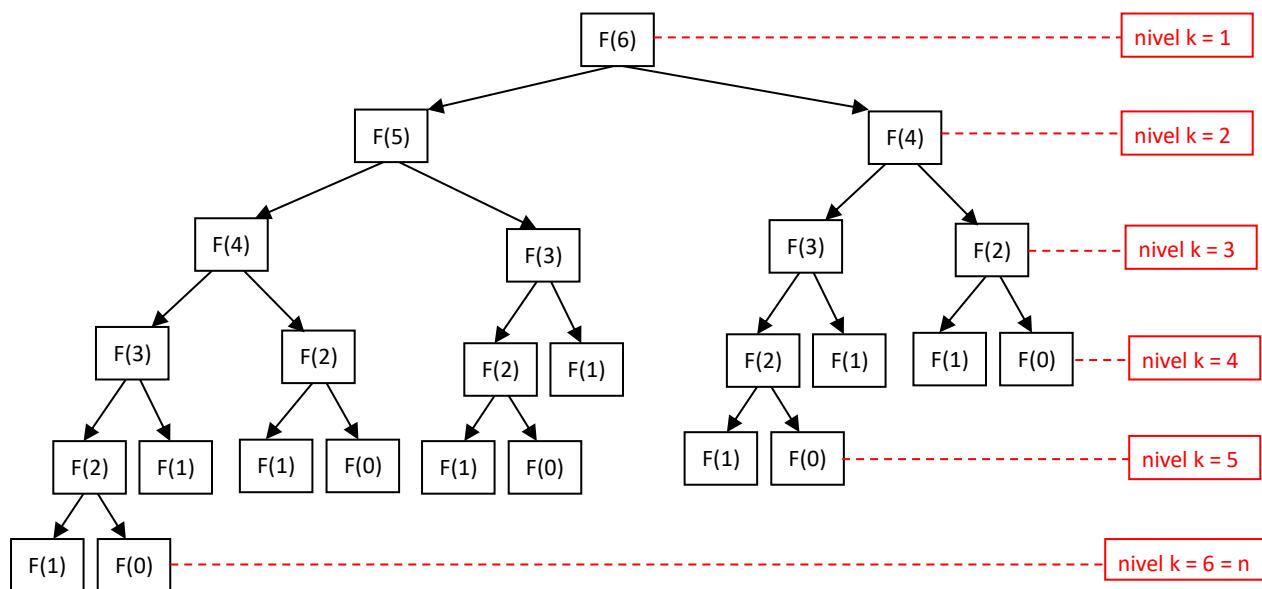
Está claro que la *versión iterativa* se ejecuta en un tiempo proporcional a  $n$ , en forma similar al cálculo del factorial: a medida que  $n$  aumenta, en la misma proporción aumenta el tiempo de ejecución ya que depende de cuántas repeticiones haga el ciclo *for*.

El espacio de memoria empleado por la *versión iterativa* es siempre el mismo: un puñado de variables locales, que no cambia aunque  $n$  sea mayor o menor.

Si se comienza un análisis comparativo, está muy claro que la *versión recursiva* es evidentemente más simple de entender y más compacta en cuanto a complejidad de código fuente que la *versión iterativa*. Pero el análisis del tiempo de ejecución (y en menor medida, del consumo de memoria) de la *versión recursiva* no es tan simple debido a la presencia de **dos** invocaciones recursivas.

Para intentar una aproximación intuitiva, podemos mostrar un gráfico de la estructura de la *cascada de llamadas* que implica la *versión recursiva*. Si suponemos que se desea calcular el valor de *fibonacci02(6)* (que abreviaremos como  $F(6)$  en el gráfico, para simplificar), se puede ver que se producirá un *árbol de llamadas* similar al siguiente (aunque comprenda, *no todo el árbol estará contenido en la memoria al mismo tiempo*):

**Figura 6: Esquema del árbol de llamadas recursivas para  $F(6)$ .**



Como se dijo, este es el *árbol completo* de invocaciones pero *nunca* estará totalmente contenido en memoria: de hecho, *sólo cada una de las ramas de este árbol* estará completamente en el *stack segment* en un momento dado, por lo que se puede ver que la ocupación de memoria del proceso en el peor caso, equivale a la cantidad de niveles de la rama más larga del árbol (en este caso, la de la extrema izquierda). Puede verse que para calcular el término  $F(6)$  la rama más larga tiene exactamente **6(seis) niveles**, por lo que es simple comprobar que  $F(n)$  requerirá  $n$  niveles, y de allí que el consumo de memoria será

proporcional a  $n$ : a medida que  $n$  aumente, lo hará también el espacio de memoria de stack usado, en proporción directa con  $n$  (esto en sí mismo ya es malo en comparación con la **versión iterativa**... cuyo espacio de memoria se mantenía constante sin importar qué tan grande sea  $n$ ...)

Pero los problemas verdaderamente comienzan cuando se analiza el *tiempo de ejecución*. En este caso, **sí** importa el *árbol completo*, pues debemos calcular *cuánto tiempo llevará hacer el proceso total*. De nuevo, el hecho técnico de invocar a una función lleva un tiempo constante, pero a ese tiempo debe sumarse el que lleve completar el proceso contenido en ella. En nuestra **versión recursiva**, el proceso consta básicamente de una suma que también ejecuta en tiempo constante... y por lo tanto, *todo queda reducido a saber cuántas veces se invoca la función a lo largo de toda la ejecución*.

Se puede sospechar que será un número elevado, ya que sólo para  $F(6)$  tenemos 25 invocaciones (el número de rectángulos o *nodos* del árbol anterior), pero además podemos ver que muchas de esas invocaciones se hacen para realizar un proceso que ya había sido completado antes: sin contar los cálculos triviales para  $F(1)$  y  $F(0)$ , vemos que dos veces se calcula  $F(4)$ , tres veces se calcula  $F(3)$  y cuatro veces se calcula  $F(2)$ . El árbol no sólo es denso de por sí, sino que además se pierde tiempo re-calculando valores que ya se tenían...

En definitiva: ¿cuánto tiempo llevará el proceso completo para  $F(n)$  en la **versión recursiva**? Intuitivamente, si se observa el árbol de llamadas recursivas para  $F(6)$ , puede verse con claridad que el número total de invocaciones a medida que se completa cada nivel aumenta en forma más o menos predecible:

Nivel	Invocaciones hasta ese nivel (incluido)
$k = 1$	$1 = 2^1 - 1$
$k = 2$	$3 = 2^2 - 1$
$k = 3$	$7 = 2^3 - 1$
$k = 4$	$15 = 2^4 - 1$
$k = 5$	23
$k = 6$	25

Al menos hasta el nivel  $k = 4$ , todos los niveles están completos y el número de llamadas acumulado hasta allí varía en *forma exponencial*. Y con ello podemos empezar a suponer que el tiempo de ejecución será algo de la forma  $b^n$  para alguna base  $b$  constante y mayor a 1.

Puede probarse que este resultado intuitivo es efectivamente correcto, y veremos que si un algoritmo/programa tiene un tiempo de ejecución exponencial (como en este caso) entonces estamos en serios problemas... Si el tiempo de ejecución de un algoritmo cambia en relación exponencial a medida que  $n$  crece, entonces ese algoritmo sólo será aplicable cuando  $n$  sea realmente muy pequeño.

Si  $n$  toma un valor de entre 30 y 35 la cantidad de pasos que el algoritmo supone es tan grande que incluso una computadora moderna muy veloz comenzará a verse en problemas para terminar el proceso y entregar el resultado en un tiempo prudente. Puede probar el programa *test01.py* del proyecto [F26] *Recursión* que acompaña a esta ficha, y cargar los valores 30, 31, 32, 33, 34 y 35 para darse una idea de lo que ocurre con pequeñas variaciones en el valor de  $n$ ...

Para valores de  $n$  mayores a 35 y hasta 40 el proceso se torna penosamente lento, de forma que ya para  $n = 40$  posiblemente tenga que detener el programa en forma manual (y todo mientras la *versión iterativa* finaliza en un instante).

Para  $n = 50$  o  $n = 60$  el *proceso recursivo es definitivamente inaceptable* en términos prácticos: la computadora podría estar miles o centenares de miles años haciendo el cálculo y buscando el resultado (que la *versión iterativa* obtiene en unos pocos milisegundos). Y si fuese  $n = 100$ , entonces será hora de rendirse: el *programa recursivo ejecutará durante todo el tiempo de vida que le queda al universo, y no alcanzará a calcular el valor de  $F(100)$* . Ejecute el programa *test01.py* ya mencionado, e inténtelo si no lo cree...

Veremos más adelante en este mismo curso, que si un problema *sólo admite soluciones que se ejecutan en tiempo exponencial*, entonces ese problema se dice **intratable** y constituye todo un desafío para las Ciencias de la Computación. Note que el cálculo del término  *$n$ -ésimo de Fibonacci no es un problema intratable*, ya que se conoce al menos un *algoritmo de tiempo de ejecución no exponencial* para resolverlo: nuestra ya conocida *versión iterativa*.

A modo de conclusión:

- La recursividad es una herramienta muy útil para el planteo de algoritmos, pero debe ser usada con cuidado y con conocimiento adecuado en cuanto a la forma de estimar el uso de recursos de tiempo y memoria.
- En general, desde el punto de vista de la *complejidad del código fuente*, la recursión permitirá escribir programas más compactos, más simples de comprender, y más consistentes con la definición formal del problema que en un planteo iterativo. Pero si se toma a la ligera el consumo de tiempo y memoria usada, el programa obtenido podría simplemente ser inaceptable en la práctica.
- Algunos programadores suelen ensayar soluciones recursivas para un problema, dado que en términos de complejidad de código fuente podría resultar más simple; y si luego descubren que esas soluciones son poco eficientes proceden a eliminar la recursión y convertir el programa en un proceso iterativo (aunque como dijimos, no siempre esto es simple de hacer)
- ¿¿¿Obtuvo ya el resultado de  $F(100)???$
- A la luz de todo lo dicho, podría alegarse que entonces no es conveniente aplicar recursividad *en ningún caso*. Sin embargo, esta postura pesimista está lejos de ser cierta: si la recursión se aplica en la forma apropiada, puede dar lugar a soluciones muy eficientes (al menos en cuanto a tiempo de ejecución) y sorprendentemente compactas en cuanto a complejidad de código fuente. Ciertas estrategias de resolución de problemas (como la que se conoce como *Divide y Vencerás* (*DyV*)) se basan en un esquema recursivo aplicado con ciertas restricciones, y de acuerdo a esas restricciones pueden resolverse distintos problemas en forma muy eficiente respecto del tiempo de ejecución.
- La estrategia conocida como *Backtracking* (o *Vuelta Atrás*) es otra estrategia de planteo de algoritmos que se monta sobre un proceso recursivo para ayudar a explorar diversas soluciones a un problema y quedarse con la mejor a medida que las instancias recursivas van finalizando. Muchos de los problemas que se resuelven con *Backtracking*, serían casi imposibles de resolver sin esa técnica.
- Sin embargo, el enfoque pesimista todavía podría alegar que las estrategias *DyV* y *Backtracking* son *técnicas diferentes* y con  *entidad propia* para el planteo de algoritmos. Se podría alegar que la *recursividad aplicada en forma directa* y sin las restricciones propias de

la estrategia *DyV* o el *Backtracking* no es práctica. Y de nuevo, la respuesta es que eso no es cierto. Como dijimos, muchas aplicaciones propias del recorrido de estructuras de datos no lineales (como *árboles binarios*, *árboles n-arios* y *grafos*) se resuelven en forma natural y eficiente usando *recursividad directa*, con el adicional de que esos procesos resultan muy compactos y simples de comprender en cuanto a código fuente.

- ¿¿¿Obtuvo ya el resultado de  $F(100)???$
- Y por supuesto, existen áreas específicas en las ciencias de la computación en las que el empleo de la recursión en forma directa o indirecta (la *recursión indirecta o mutua*, es aquella situación en la que una función  $A()$  invoca a otra  $B()$ , pero luego  $B()$  invoca a su vez a  $A()$ ) es prácticamente obligado, pues de otro modo los algoritmos serían absurdamente difíciles de plantear. Una de esas áreas es la de la generación de *figuras fractales*. En casos así, el factor de eficiencia preponderante es evidentemente, la *complejidad del código fuente*.
- Ya puede cancelar el programa del cálculo de  $F(100)$ . A menos que esté dispuesto a esperar hasta el fin de los tiempos....

---

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: <https://docs.python.org/3/>.
- [3] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [4] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

# Ficha 27

## Estrategias de Resolución de Problemas: Divide y Vencerás

### 1.] Aplicaciones de la recursividad: Introducción a los gráficos fractales.

Hemos indicado que la recursividad debe ser usada con cuidado ya que aplicada con criterio incorrecto puede llevar a situaciones de excesivo (e innecesario) consumo de memoria (como se vio para el caso del factorial), o bien a implementaciones cuyos tiempos de ejecución resultan inaceptables (como vimos para el caso de la sucesión de Fibonacci).

Pero también hemos indicado que ciertos casos generales la recursión es una excelente herramienta para planteo de soluciones a problemas que de otro modo serían extremadamente difíciles desde el punto de vista de la complejidad del código fuente. En este tipo de problemas, el consumo extra de memoria que hace la recursión está justificado y el tiempo de ejecución será aceptable.

Como ya se dijo, una de las áreas en que la recursión es recomendable es el procesamiento de *estructuras de datos no lineales* como los *árboles* o los *grafos*. Y otra de esas áreas es la *generación y tratamiento de figuras y gráficos fractales*, que es el tema sobre el que está centrado el desarrollo de esta sección. [1]

Una *figura fractal* es aquella que se compone de versiones más simples y pequeñas de la misma figura original, y curiosamente existen numerosos ejemplos de formaciones fractales en la naturaleza, como se ve en las imágenes de la figura que sigue:

Figura 1: Algunos ejemplos de formaciones *fractales naturales*.<sup>1</sup>



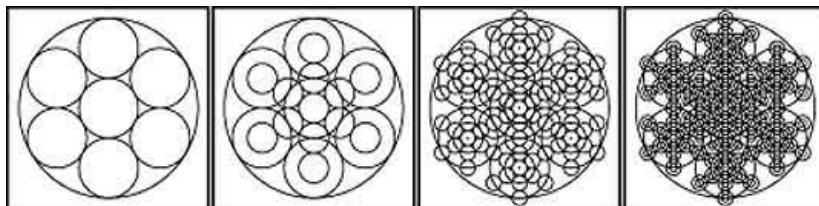
Muchas figuras fractales naturales pueden imitarse y reproducirse artificialmente mediante algoritmos recursivos. Pero también puede pensarse directamente en generar figuras y

<sup>1</sup> Las fuentes de las imágenes que se muestran en la *Figura 1* son las siguientes, tomadas de izquierda a derecha:

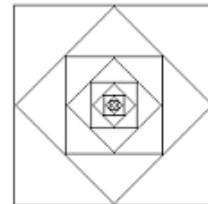
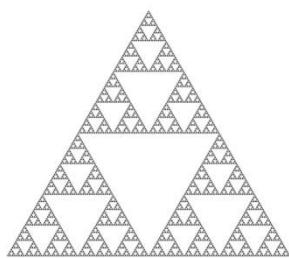
- ✓ <http://ztfnews.files.wordpress.com/2010/10/helechom.gif>
- ✓ [http://www.oddee.com/\\_media/imgs/articles/a302\\_f5.jpg](http://www.oddee.com/_media/imgs/articles/a302_f5.jpg)
- ✓ <https://cuquialcocer.files.wordpress.com/2012/04/fractal2.jpg>

curvas basadas en geometría fractal que no estén inspiradas en figuras naturales<sup>2</sup>. La *Figura 2* provee algunos ejemplos relativamente simples de visualizar:

**Figura 2:** Algunas gráficas *fractales artificiales* simples.<sup>3</sup>



a.) Composición fractal con círculos



b.) Composición fractal con cuadrados

c.) Composición fractal con triángulos

Todas la gráficas de la *Figura 2* están compuestas por la misma figura geométrica, repetida una y otra vez con diversos tamaños y posiciones siguiendo un patrón. Cada imagen remite de inmediato a una secuencia recursiva gráfica (y cada una puede entenderse como una definición recursiva intuitiva de la forma "*un círculo compuesto por círculos*" o un "*cuadrado compuesto por cuadrados*", etc.) Y como veremos, esta clase de figuras son especialmente aptas para dibujarse mediante programas recursivos (las gráficas **b** y **c** de la figura anterior están programadas en el proyecto [F27] *Piramide* que acompaña a esta Ficha [programas *caleidoscopio.py* y *triangulo.py*])

## 2.] Generación de gráficos fractales.

La misma idea que se expuso en el apartado anterior vale para la atractiva pirámide coloreada que se muestra en la *Figura 3* (página 532). La figura es una composición fractal formada exclusivamente por cuadrados, de manera que se vayan dibujando unos sobre otros para armar los cuatro soportes de una estructura piramidal vista desde arriba: El gran

<sup>2</sup> Las figuras fractales naturales también son comunes en el cuerpo humano y un ejemplo es el iris del ojo en cuanto la distribución del color y las formaciones internas que pueden observarse en él. Al respecto, una extraña película del año 2014, titulada *I Origins* (conocida en español como *Orígenes*), dirigida por *Mike Cahill* y protagonizada por *Michael Pitt*, *Àstrid Bergès-Frisbey* y *Brit Marling* narra la historia de un investigador de la evolución del ojo humano que hace un descubrimiento colateral sorprendente, justamente en relación al iris... Extraña pero digna de verse, prestándole atención a los detalles.

<sup>3</sup> Las fuentes de los gráficos que se muestran son las siguientes:

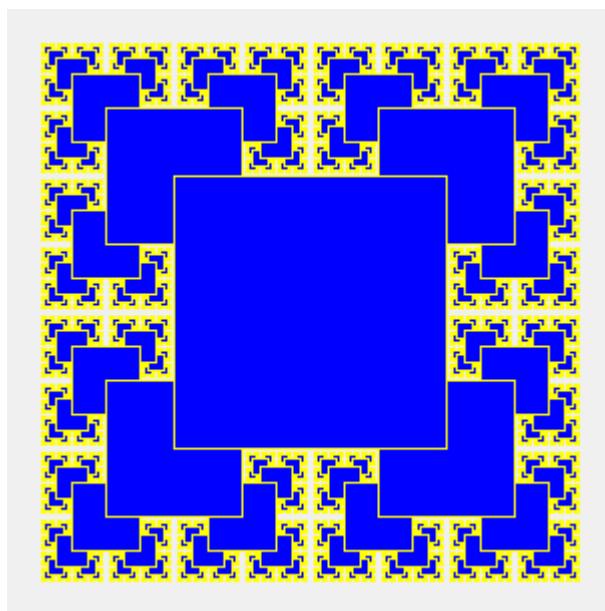
- a.) <http://www.thecamino.com.ar/imagf/spiral08.jpg>
- b.) [http://www.infinitefractal.com/movabletype/blogs/my\\_blog/cart/Diapositiva3.jpg](http://www.infinitefractal.com/movabletype/blogs/my_blog/cart/Diapositiva3.jpg)
- c.) <http://2.bp.blogspot.com/-AD-lxWAAzg/Ughm1btCCPI/AAAAAAA40/1NBDGQlGhO0/s1600/fractal+6.png>

cuadrado azul que se ve en primer plano constituye la cima de la pirámide, y los cuatro soportes se van estirando hacia los cuatro vértices de la gráfica. [1]

La figura no sólo es fractal por estar compuesta por cuadrados que van disminuyendo su tamaño: cada uno de los soportes es a su vez otra pirámide más pequeña, con exactamente la misma forma y proporciones que la pirámide completa.

Esto comienza a sugerir la idea de usar una función recursiva para lograr el dibujo entero. La parte superior de la pirámide muestra un gran cuadrado a modo de tapa, y otros cuatro cuadrados más pequeños (con lados la mitad de largos que el cuadrado mayor) sirviendo como apoyo en cada uno de los vértices. A su vez cada uno de esos cuatro cuadrados tiene otros cuatro más pequeños como soporte, y la idea se repite hasta la base de la pirámide.

Figura 3: *Pirámide fractal* formada por una composición de cuadrados coloreados.<sup>4</sup>



Python provee numerosos y potentes mecanismos para el diseño de ventanas y componentes gráficos. Si bien no forma parte de los objetivos de este curso hacer un estudio detallado respecto de esos mecanismos, el hecho es que resulta relativamente simple desarrollar un programa que genere mínimamente gráficos como el anterior. [2] [3]

Además, también es cierto que un curso de programación introductorio podría verse muy beneficiado si los estudiantes pudieran utilizar recursos visuales de alto impacto en el corto plazo: la experiencia nos ha mostrado que esos recursos incentivan la creatividad y constituyen un fuerte impulso de motivación: al fin y al cabo, resulta obvio que un programa resulta mucho más atractivo de usar (y de desarrollar...) si el mismo cuenta con ventanas y gráficos que hagan más sencillo su funcionamiento y más accesibles sus recursos. [4]

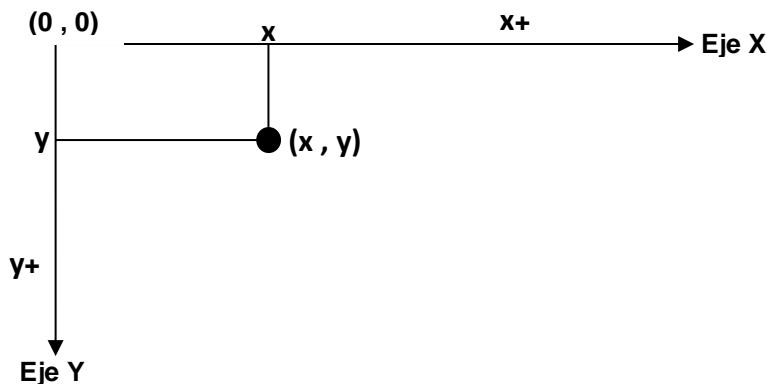
Por lo pronto, el primer elemento que debe quedar claro y dominarse rápidamente es el *esquema de coordenadas de pantalla* que Python (y todo otro lenguaje) supone para la

<sup>4</sup> La idea de la pirámide fractal expuesta en esta sección está inspirada en la presentación del tema en el libro "Estructuras de Datos en Java" de Mark Allen Weiss (página 179).

gestión de gráficos. El sistema de coordenadas de Python permite localizar a cada punto de luz o *pixel* de la pantalla en forma directa.

Por defecto, el punto superior izquierdo dentro del área de dibujo de una ventana o un lienzo gráfico (conocido como un *canvas*) es el *origen de coordenadas* de ese contenedor gráfico (o sea, el punto de coordenadas  $(0, 0)$  del contenedor). Si la distancia de un *pixel* al origen del sistema en el eje horizontal se designa como  $x$ , entonces  $x$  será la *coordenada de columna* de ese pixel. De forma similar, si la distancia del pixel al origen en el eje vertical se designa como  $y$ , entonces  $y$  será la *coordenada de fila* del pixel (ver Figura 4).

**Figura 4:** El sistema de coordenadas de pantalla.



Note que en este sistema de coordenadas, el *número de filas crece hacia abajo en la pantalla* (y no hacia arriba, como quizás el estudiante esperaría dada la costumbre de los ejes cartesianos que normalmente se usan en matemáticas). Esto quiere decir que una fila que se encuentre más cerca del borde superior de la pantalla o ventana, tendrá un número *menor* que otra fila que se encuentre más abajo. El crecimiento en el número de columnas sigue la misma idea que los gráficos cartesianos normales (es decir, los números de columna crecen hacia la derecha en la pantalla).

El programa completo para generar la pirámide de la Figura 3 es el que sigue (y viene acompañando a esta Ficha en el proyecto [F27] Piramide):

```
from tkinter import *
__author__ = 'Cátedra de AED'

def pyramid(canvas, x, y, r):
    if r > 0:
        # dibujar recursivamente el soporte inferior izquierdo...
        pyramid(canvas, x-r, y+r, r//2)

        # dibujar recursivamente el soporte inferior derecho...
        pyramid(canvas, x+r, y+r, r//2)

        # dibujar recursivamente el soporte superior izquierdo...
        pyramid(canvas, x-r, y-r, r//2)

        # dibujar recursivamente el soporte superior derecho...
        pyramid(canvas, x+r, y-r, r//2)

        # dibujar en (post-orden) la tapa o cima de la pirámide...
        square(canvas, x, y, r)

def square(canvas, x, y, r):
```

```

left = x - r
top = y - r

right = x + r
down = y + r

canvas.create_rectangle((left, top, right, down), outline='yellow', fill='blue')

def render():
    # configuración inicial de la ventana principal...
    root = Tk()
    root.title("Piramide Fractal")

    # calculo de resolucion en pixels de la pantalla...
    maxw = root.winfo_screenwidth()
    maxh = root.winfo_screenheight()

    # ajustar dimensiones y coordenadas de arranque de la ventana...
    root.geometry("%dx%d+%d+%d" % (maxw, maxh, 0, 0))

    # un lienzo de dibujo dentro de la ventana...
    canvas = Canvas(root, width=maxw, height=maxh)
    canvas.grid(column=0, row=0)

    # sea valor inicial de r igual a un duodécimo del ancho de la pantalla...
    r = maxw // 12

    # coordenadas del centro de la pantalla...
    cx = maxw // 2
    cy = maxh // 2

    # dibujar piramide centrada en (cx, cy) y el cuadrado mayor con lado = 2r.
    pyramid(canvas, cx, cy, r)

    # lanzar el ciclo principal de control de eventos de la ventana...
    root.mainloop()

if __name__ == '__main__':
    render()

```

El programa comienza con una instrucción *import* para dar acceso al contenido del módulo *tkinter*, que es el que básicamente contiene las declaraciones y funciones para manejar ventanas y gráficos:

```

from tkinter import *
__author__ = 'Cátedra de AED'

```

La función *pyramid()* es la que realiza *el gráfico completo de la pirámide* usando recursión, pero veremos primero la forma de dibujar un cuadrado simple.

La función *square(canvas, x, y, r)* es la encargada simplemente de dibujar un cuadrado de bordes amarillos y pintado por dentro de color azul.

```

def square(canvas, x, y, r):
    left = x - r
    top = y - r

    right = x + r
    down = y + r

    canvas.create_rectangle((left, top, right, down), outline='yellow', fill='blue')

```

El parámetro `canvas` recibido por la función, es el lienzo o área de dibujo en donde debe graficarse el cuadrado (por el momento, no es necesario saber de dónde sale ese lienzo: simplemente, la función `square()` lo toma como parámetro y dibuja en él el cuadrado pedido).

Los parámetros `x` e `y` son las coordenadas de columna y fila (respectivamente) del pixel que queremos que sea el *centro* del cuadrado dentro del `canvas` donde será dibujado. Y el parámetro `r` es el valor de la mitad de la longitud del lado del cuadrado a dibujar (en pixels), por lo que la longitud total del lado será igual a  $2 \cdot r$ .

Las cuatro primeras líneas de la función `square()` calculan entonces las coordenadas de pixel de las esquinas superior izquierda e inferior derecha del cuadrado (asignando esos valores en las variables (`left`, `top`) y (`right`, `down`)) (asegúrese de entender la naturaleza de estos cuatro cálculos).

Finalmente, en la última línea de la función `square()`, se invoca a la función `canvas.create_rectangle()` para dibujar el cuadrado. Un elemento u objeto de tipo *Canvas* dispone de funciones (o métodos) para dibujar distintos tipos de figuras básicas o primitivas. En este caso, el método `create_rectangle((left, top, right, down), outline='yellow', fill='blue')` recibe como primer parámetro una tupla conteniendo los valores de las coordenadas de los pixels superior izquierdo e inferior derecho del rectángulo a dibujar. El parámetro `outline` (accedido por palabra clave) indica el color con el que será dibujado el borde o perímetro del rectángulo (aquí lo asignamos en amarillo ('yellow')). Y el parámetro `fill` (también accedido por palabra clave) indica el color con el que será pintado por dentro el rectángulo (y aquí lo asignamos en azul ('blue')). [2]

Note que esta función ***no dibuja la pirámide***: sólo dibuja ***un*** cuadrado en las coordenadas centrales (`x`, `y`) que se le indiquen y con longitud de su lado igual a  $2r$ . Es la función `pyramid()` la que dibuja ***toda la pirámide***, aplicando recursión e invocando tantas veces como sea necesario a la función `square()`.

La *definición recursiva* del concepto de "pirámide formada por cuadrados" puede enunciarse intuitivamente así:

"Una ***pirámide formada por cuadrados de lado máximo  $2r$***  estará vacía si  $r$  es 0 (no se puede dibujar un lado de longitud 0), o bien contendrá un cuadrado de lado  $2r$  en la cima que estará apoyado en cuatro soportes. ***Pero estos cuatro soportes serán a su vez pirámides formadas por cuadrados cuyos lados serán de longitud máxima  $r$ .***"

Claramente esta definición es recursiva: el concepto de pirámide aparece en la propia definición. Pues bien: se trata de dibujar ***una pirámide formada por cuatro pirámides menores (los soportes) y un cuadrado en la cima***. Y eso es exactamente lo que hace la función `pyramid(canvas, x, y, r)` (los parámetros tienen el mismo significado que en la función `square()`):

```
def pyramid(canvas, x, y, r):
    if r > 0:
        # dibujar recursivamente el soporte inferior izquierdo...
        pyramid(canvas, x-r, y+r, r//2)

        # dibujar recursivamente el soporte inferior derecho...
        pyramid(canvas, x+r, y+r, r//2)

        # dibujar recursivamente el soporte superior izquierdo...
        pyramid(canvas, x-r, y-r, r//2)
```

```

# dibujar recursivamente el soporte superior derecho...
pyramid(canvas, x+r, y-r, r//2)

# dibujar en (post-orden) la tapa o cima de la pirámide...
square(canvas, x, y, r)

```

La situación trivial se da cuando  $r$  es 0: en ese caso la función simplemente termina sin hacer nada. Pero si  $r$  es mayor a 0 (la longitud del lado del cuadrado es por lo menos  $2*1 = 2$ ), entonces se procede al dibujado "desde abajo y hacia la cima", con cuatro invocaciones recursivas (una para cada soporte). Usamos recursión pues tenemos que dibujar *cuatro pirámides que sostengan a la pirámide mayor...*

Los valores  $x$ ,  $y$ ,  $r$  enviados como parámetro a cada instancia recursiva de la función ***pyramid()*** se calculan de forma que cada nueva pirámide se dibuje con diferentes coordenadas del centro y una longitud  $2r$  para el lado del cuadrado en la cima progresivamente menor, dividiendo por 2 a  $r$  en cada invocación. Esta sucesión de divisiones en algún momento hará que se llegue a un valor de  $r$  que será igual a 0, y en ese momento se detendrá la recursión.

Observe un detalle: las cuatro invocaciones recursivas se hacen ***antes*** que la invocación a la función ***square()***... lo cual tiene sentido, ya que primero deben construirse las paredes antes de colocar el techo! Formalmente, como el proceso de dibujar el cuadrado se hace después que terminan las invocaciones recursivas, entonces tenemos un proceso de dibujado en ***post-orden()*** (*u orden posterior*) [5] como se indicó en la *Ficha 26*.

La última función que queda en el programa es ***render()***, y es la encargada de preparar el contexto visual y gráfico:

```

def render():
    # configuracion inicial de la ventana principal...
    root = Tk()
    root.title("Piramide Fractal")

    # calculo de resolucion en pixels de la pantalla...
    maxw = root.winfo_screenwidth()
    maxh = root.winfo_screenheight()

    # ajustar dimensiones y coordenadas de arranque de la ventana...
    root.geometry("%dx%d+%d+%d" % (maxw, maxh, 0, 0))

    # un lienzo de dibujo dentro de la ventana...
    canvas = Canvas(root, width=maxw, height=maxh)
    canvas.grid(column=0, row=0)

    # sea valor inicial de r igual a un duodécimo del ancho de la pantalla...
    r = maxw // 12

    # coordenadas del centro de la pantalla...
    cx = maxw // 2
    cy = maxh // 2

    # dibujar piramide centrada en (cx, cy) y el cuadrado mayor con lado = 2r.
    pyramid(canvas, cx, cy, r)

    # lanzar el ciclo principal de control de eventos de la ventana...
    root.mainloop()

```

Prácticamente todas las novedades técnicas para la creación y control de la ventana principal y el *canvas* están en esta función [2]. La primera instrucción crea una variable que hemos llamado *root* con la función *Tk()*. Una variable de tipo *Tk* representa una ventana

básica, con bordes definidos, barra de título y controles de minimización y cierre de la ventana. La variable `root` será entonces la que nos permitirá controlar lo que ocurre con la ventana donde se despliega el gráfico. Y la segunda instrucción simplemente asigna la cadena de caracteres que será visualizada en la barra de título de esa ventana:

```
# configuracion inicial de la ventana principal...
root = Tk()
root.title('Piramide Fractal')
```

Las siguientes dos líneas permiten básicamente averiguar la resolución (en pixels) de la pantalla en el momento de ejecutar el programa, para poder luego calcular en forma correcta las dimensiones de la ventana y, sobre todo, las coordenadas del pixel central:

```
# calculo de resolucion en pixels de la pantalla...
maxw = root.winfo_screenwidth()
maxh = root.winfo_screenheight()
```

La función `winfo_screenwidth()` retorna el ancho máximo en pixels admitido por la resolución actual de la pantalla, y la función `winfo_screeheight()` hace lo mismo pero con la altura. Al momento de ejecutar este programa para preparar esta Ficha, la resolución era de 1366 pixels de ancho por 768 pixels de alto, y esos dos valores (o los que correspondan) se almacenan en las variables `maxw` y `maxh`.

La quinta línea del programa ajusta lo que se conoce como la *geometría de la ventana principal* (que como vimos, tenemos representada con la variable `root`):

```
# ajustar dimensiones y coordenadas de arranque de la ventana...
root.geometry('%dx%d+%d+%d' % (maxw, maxh, 0, 0))
```

La función `geometry()` permite ajustar el ancho y el alto de la ventana en cualquier momento, y también las coordenadas del pixel superior izquierdo desde donde debe mostrarse esa ventana. La función toma como parámetro una *cadena de caracteres* de la forma '`WxH+x+y`' en la que `W` es el ancho en pixels que queremos que tenga la ventana al mostrarse, `H` es la altura en pixels para la ventana, `x` es la coordenada de columna del pixel superior izquierdo donde queremos que aparezca, e `y` es la coordenada de fila de ese mismo pixel. A modo de ejemplo simple, si hubiésemos hecho algo como:

```
root.geometry('400x300+50+50')
```

entonces la ventana representada por la variable `root` tomaría un ancho de 400 pixels, una altura de 300 pixels, y sería visualizada con su esquina superior izquierda en el pixel de coordenadas (50, 50) de la pantalla.

El problema es que en muchos casos los valores que se quieren asignar están contenidos en variables y no vienen como constantes directas. En nuestro programa, por ejemplo, queremos que la ventana aparezca con un ancho tan grande como el que se tenga disponible en la resolución actual (y como vimos, tenemos ese valor almacenado en la variable `maxw`) y lo mismo vale para la altura (que tenemos en `maxh`). Si quisieramos esos valores ancho y alto, y hacer aparecer la ventana desde el pixel (0, 0), entonces una invocación de la forma:

```
root.geometry('maxw x maxh + 0+0')
```

simplemente provocaría un error al ejecutarse: la cadena contiene letras donde Python esperaba encontrar dígitos...

La forma de tomar esos valores desde variables, consiste en usar *caracteres de reemplazo* (también llamados *caracteres de formato*) en la cadena. El par de caracteres `%d` se toma como si fuese un único carácter, y se interpreta como que en ese lugar debe ir el valor de una variable de tipo entero (`int`) que se informa más adelante en la propia lista de parámetros de la función. En nuestro programa, la función se está invocando así:

```
root.geometry('%dx%d+%d+%d' % (maxw, maxh, 0, 0))
```

lo cual significa que el primer carácter `%d` (marcado en rojo en el ejemplo) será reemplazado por el valor de la variable `maxw` que es a su vez la primera en la tupla que aparece luego del signo `%` (de color negro). En forma similar, el segundo `%d` (azul) será reemplazado por el valor de la variable `maxh`, que es la segunda en la tupla, y así hasta el final.

Las dos instrucciones siguientes asignan una variable que hemos llamado *canvas* con un elemento u objeto de tipo *Canvas* para hacer allí los gráficos que necesitamos:

```
# un lienzo de dibujo dentro de la ventana...
canvas = Canvas(root, width=maxw, height=maxh)
canvas.grid(column=0, row=0)
```

La función *Canvas()* crea un objeto que representa un lienzo de dibujo. Esa función toma varios parámetros, y en este caso el primero es la variable `root` que representa nuestra ventana principal. Al enviar la ventana como parámetro, le estamos indicando al programa que el *Canvas* que estamos creando (variable `canvas`) estará contenido en esa ventana (variable `root`) y por eso todo lo que se dibuje en `canvas`, aparecerá en la ventana `root`.

Los otros dos parámetros que estamos enviando a la función *Canvas()* se están accediendo por palabra clave, e indican el ancho y el alto en pixels que debe tener el *canvas* dentro de la ventana. En este caso, simplemente hemos asumido que el *canvas* será el único elemento contenido en la ventana `root`, y le hemos fijado un ancho y un alto máximo (los valores de nuestras variables `maxw` y `maxh`). Por supuesto, esto puede ajustarse a voluntad del programador.

Cuando una ventana es creada, todos los elementos que se incluyan dentro de ella serán distribuidos en filas y columnas, como si el interior de la ventana fuese una tabla. La invocación a la función *grid()* que sigue a la creación del canvas, le dice al programa que el *canvas* debe ser ubicado dentro de la ventana en la "casilla" indicada por los parámetros `column` y `row` (en este caso, la casilla `[0, 0]`).

La instrucción que sigue sólo calcula el valor inicial para `r`, la mitad de la longitud del lado del cuadrado mayor que irá en la cima de la pirámide:

```
# sea un valor inicial de r igual a un duodécimo del ancho del area de dibujo...
r = maxw // 12
```

En este caso, se está tomando un valor igual a la duodécima parte del valor del ancho máximo para la ventana. El estudiante puede cambiar el 12 por otro valor y explorar los efectos que eso produce.

Las dos instrucciones que siguen simplemente calculan las coordenadas del centro de la pantalla:

```
# coordenadas del centro de la pantalla...
cx = maxw // 2
cy = maxh // 2
```

Estas coordenadas son importantes para nuestro programa, ya que le indican en qué posición del *canvas* de la ventana deberá estar centrada la vista de la pirámide, y a partir de esa posición la pirámide se irá construyendo hacia afuera.

La anteúltima instrucción invoca a la función *pyramid()* para dibujar la pirámide completa, cuyo funcionamiento recursivo ya hemos explicado más arriba:

```
# dibujar piramide centrada en (cx, cy) y el cuadrado mayor con lado = 2r.
pyramid(canvas, cx, cy, r)
```

Sólo note que el parámetro *canvas* es el lienzo de dibujo que hemos creado dentro de la ventana, y que la función *pyramid()* lo recibe justamente para poder desarrollar en él nuestro gráfico.

Finalmente, la última instrucción es la que efectivamente pone todo a funcionar:

```
# lanzar el ciclo principal de control de eventos de la ventana...
root.mainloop()
```

La ventana representada por la variable *root* se usa para invocar al método *mainloop()*, el cual hace que la ventana se muestre y quede a disposición del usuario. Toda acción que ese usuario aplique sobre la ventana (pasar el mouse sobre ella, minimizarla, presionar el botón de cierre, etc.) genera en el programa lo que se conoce como un *evento*, y el método *mainloop()* se encarga de tomar todos los eventos producidos por la ventana *root* y procesarlos adecuadamente. Por caso, es el método *mainloop()* el que cierra la ventana y da por terminado el programa cuando el usuario presione el botón de cierre ubicado arriba y a la derecha de la ventana.

### 3.] Pruebas y aplicaciones.

El programa que hemos mostrado en la sección anterior puede ser modificado de distintas maneras para producir diferentes (y muy interesantes) resultados gráficos. Nos permitimos mostrar más abajo algunas gráficas que hemos generado con algunas de esas variantes, y sugerimos al estudiante a que explore y aprenda en qué formas podría generar las mismas figuras efectuando distintos cambios en el código fuente original. Diviértase...

**Figura 5:** Gráficas producidas con modificaciones simples del programa [F27] Piramide [Parte 1]

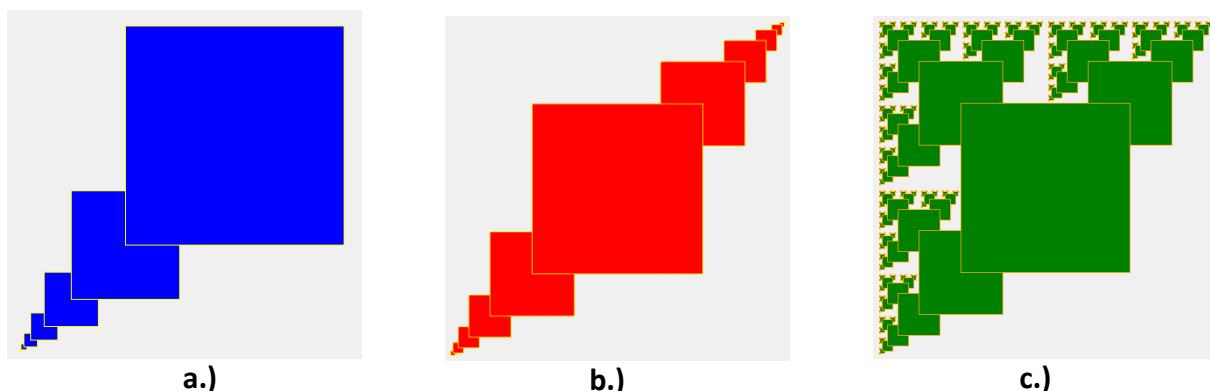
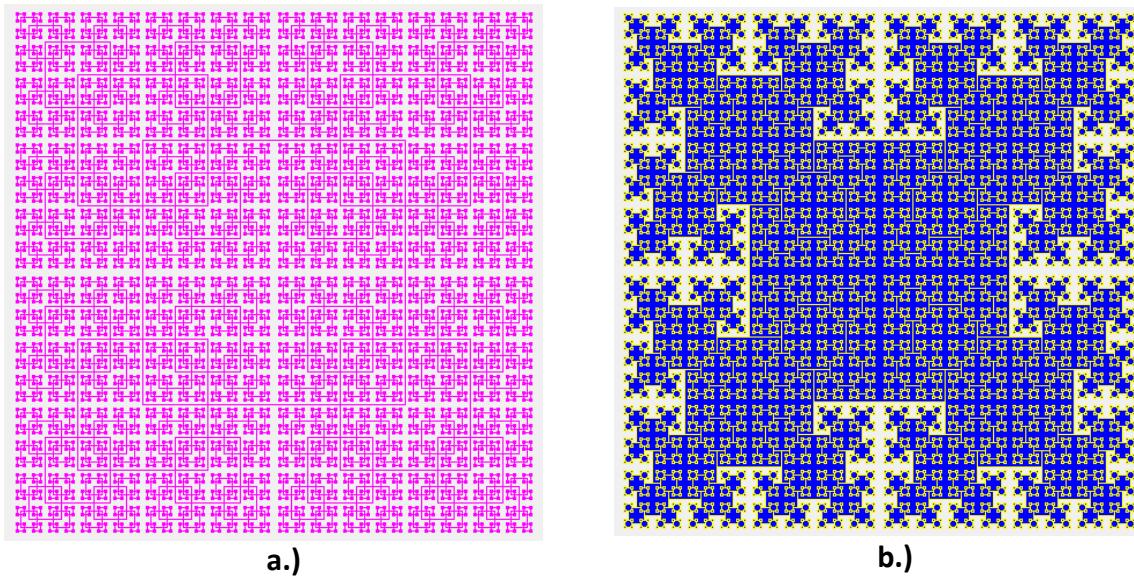
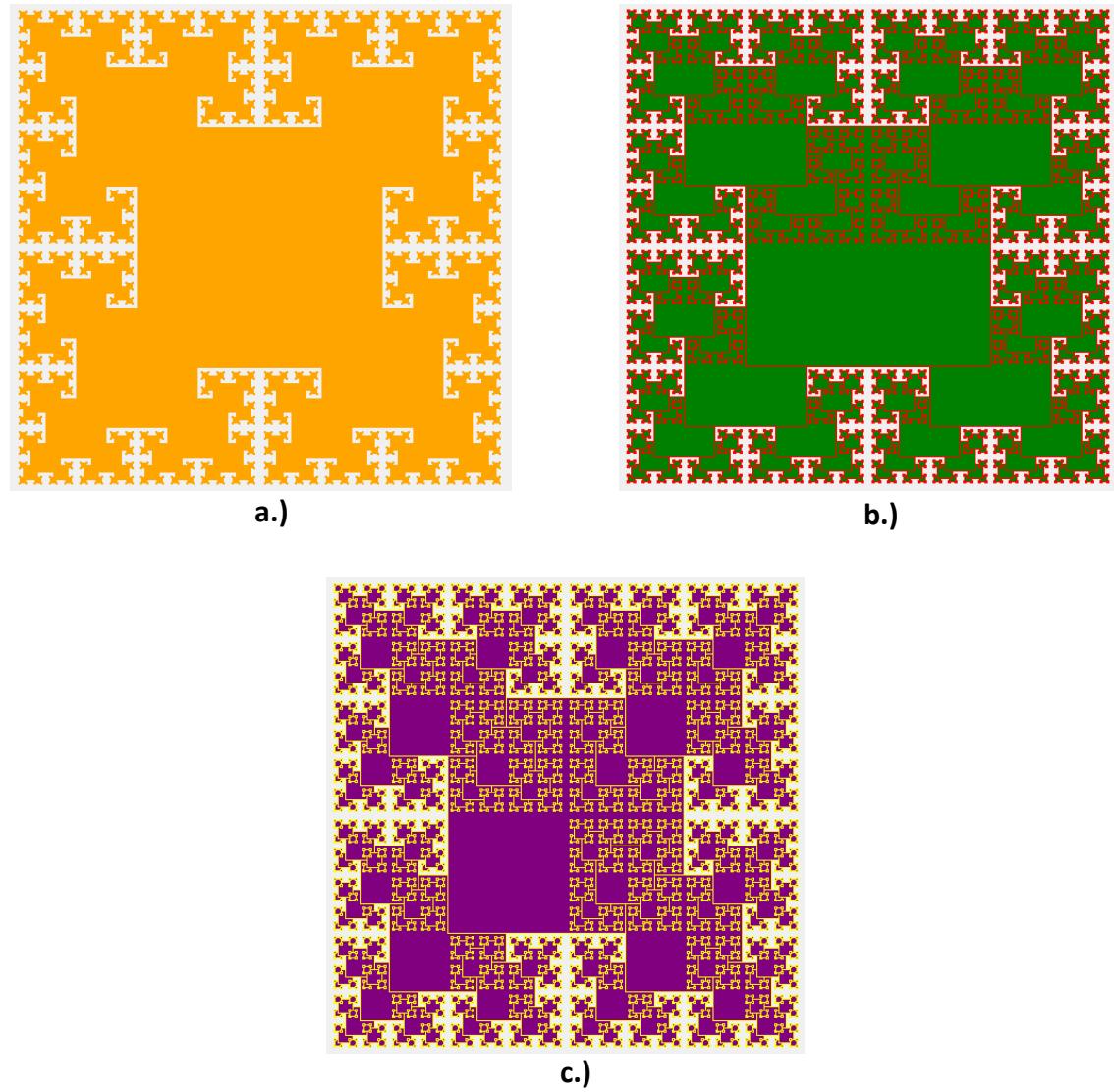


Figura 6: Gráficas producidas con modificaciones *no tan simples* del programa [F27] Piramide [Parte 2]Figura 7: Gráficas producidas con modificaciones *no tan simples* del programa [F27] Piramide [Parte 3]

#### 4.] Aplicaciones de la recursividad: El algoritmo Quicksort.

Si se observa el método de *Intercambio Directo* o *Bubblesort* (ver *Ficha 17*), se verá que algunos elementos (los mayores o "más pesados") tienden a viajar más rápidamente que otros (los menores o "más livianos") hacia su posición final en el arreglo. El proceso de "burbujeo" que lleva a cabo ese algoritmo directo es evidentemente *asimétrico*, pues los elementos más grandes se comparan más veces que los más chicos en la misma pasada, haciendo que en esa misma pasada cambien de lugar también más veces que los valores pequeños. Se suele llamar *liebres* y *tortugas* a estos elementos, en obvia alusión a su velocidad de traslado por el vector [5]. El resultado es que el algoritmo necesita muchas más pasadas hasta que se acomode el último de los menores que viaja desde la derecha...

En 1960, un estudiante de ciencias de la computación llamado *Charles Hoare* se haría famoso al presentar en *Communications of the ACM* una versión mejorada del *bubble sort*, al cual llamó simplemente *Ordenamiento Rápido* o *Quicksort*... y desde entonces ese método se ha convertido en el más estudiado de la historia de la programación, entre otras cosas por ser hasta ahora en promedio el algoritmo más rápido (aunque hay situaciones en que se comporta bastante mal, esas situaciones de peor caso son raras y poco probables y de todos modos se puede refinar el algoritmo original para que "se proteja" de esos casos y mantenga su buen rendimiento) [6] [1].

La idea es recorrer el arreglo desde los dos extremos. Se toma un elemento *pivot* (que suele ser el valor ubicado al medio del arreglo pero puede ser cualquier otro). Luego, se recorre el arreglo desde la izquierda buscando algún valor que sea *mayor que el pivot*. Al encontrarlo, se comienza una búsqueda similar pero desde la derecha, ahora buscando un *valor menor al pivot*. Cuando ambos hayan sido encontrados, se intercambian entre ellos y se sigue buscando otro par de valores en forma similar, hasta que ambas secuencias de búsqueda se crucen entre ellas. De esta forma, se favorece que tanto los valores mayores ubicados muy a la izquierda como los menores ubicados muy a la derecha, viajen rápido hacia el otro extremo... y todos se vuelven liebres.

Al terminar esta pasada, se puede ver que el arreglo no queda necesariamente ordenado, pero queda claramente *dividido en dos subarreglos*: *el de la izquierda contiene elementos que son todos menores o iguales al pivot*, y *el de la derecha contiene elementos mayores o iguales al pivot*. Por lo tanto, ahora se puede aplicar exactamente el mismo proceso a cada subarreglo, usando *recursividad*. El mismo método que partió en dos el arreglo original, se invoca a sí mismo dos veces más, para partir en otros subarreglos a los dos que se obtuvieron recién. Con esto se generan cuatro subarreglos, y con más recursión se procede igual con ellos, hasta que sólo se obtengan particiones de tamaño uno. En ese momento, el arreglo quedará ordenado. El algoritmo procede como se ve en la *Figura 8* (ver página 542):

En el proyecto [F27] *Divide y Vencerás* que viene con esta Ficha, el módulo *ordenamiento.py* incluye la siguiente función *quicksort()* que implementa el algoritmo (de hecho, el módulo *ordenamiento.py* es básicamente el mismo que ya venía con la *Ficha 17* sobre el tema *Ordenamiento*) [5]:

```
def quick_sort(v):
    # ordenamiento Quick Sort
    quick(v, 0, len(v) - 1)

def quick(v, izq, der):
```

```

x = get_pivot(v, izq, der)

i, j = izq, der
while i <= j:
    while v[i] < x and i < der:
        i += 1
    while x < v[j] and j > izq:
        j -= 1
    if i <= j:
        v[i], v[j] = v[j], v[i]
        i += 1
        j -= 1

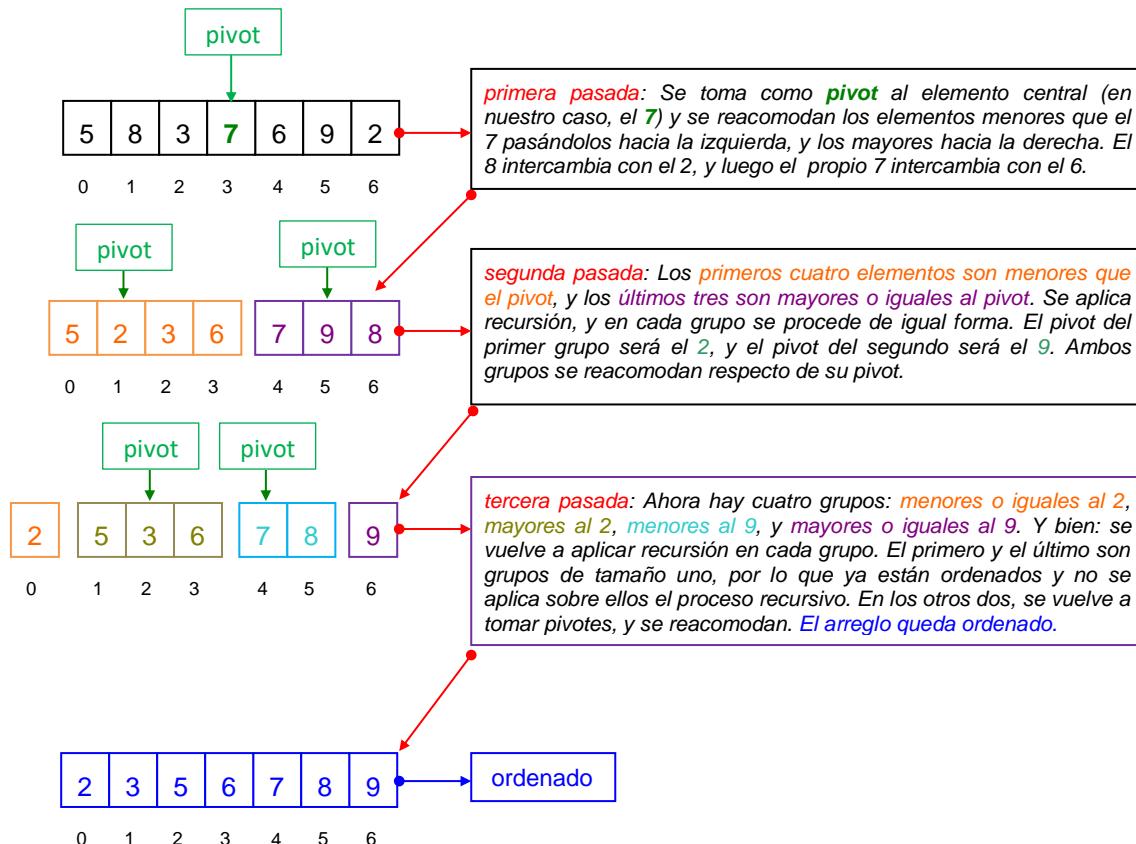
if izq < j:
    quick(v, izq, j)

if i < der:
    quick(v, i, der)

def get_pivot(v, izq, der):
    # calculo del pivot: elemento central de la partición [izq, der]
    x = v[int((izq + der) / 2)]
    return x

```

Figura 8: Esquema general de funcionamiento del algoritmo Quicksort.



El mismo proyecto contiene el programa *main.py*, que a través de un menú de opciones permite crear y ordenar un arreglo seleccionando el algoritmo de ordenamiento a aplicar (incluyendo al *Quicksort* que acabamos de mostrar). Otra vez, el programa *main.py* es esencialmente el mismo que ya se presentó con la *Ficha 17*.

## 5.] La estrategia Divide y Vencerás para resolución de problemas.

El algoritmo *Quicksort* que hemos analizado en la sección anterior es un ejemplo de aplicación de una *estrategia de resolución de problemas* muy conocida y estudiada, que se designa como estrategia *Divide y Vencerás*, por la cual los  $n$  datos del problema estudiado se dividen en subconjuntos de aproximadamente el mismo tamaño ( $n/2$  por ejemplo), luego se procesa cada uno de esos subconjuntos en *forma recursiva* y finalmente se "unen las partes" que se acaban de procesar para lograr el resultado final. Muchas veces, y dependiendo de factores tales como el *tamaño de cada subconjunto* o la *cantidad de invocaciones recursivas* que se hagan para procesar esos subconjuntos, esta estrategia favorece el diseño de algoritmos muy eficientes en cuanto al tiempo de ejecución [6].

Como toda estrategia general de resolución de problemas, la técnica *divide y vencerás* puede aplicarse en ciertos problemas (cuya estructura interna admite la división en subconjuntos de tamaños similares) para lograr soluciones con mejor o mucho mejor tiempo de ejecución en comparación a soluciones intuitivas de *fuerza bruta*, que suelen consistir en explorar todas y cada una de las posibles combinaciones de procesamiento de los datos de entrada, pero de tal forma que la cantidad de operaciones a realizar aumenta de manera dramática a medida que crece la cantidad  $n$  de datos [1].

En el algoritmo *Quicksort* la estrategia *Divide y Vencerás* es clara: se trata de dividir el arreglo en dos mitades tales que una de ellas contenga elementos menores al pivot y la otra contenga elementos mayores a ese pivot. Mediante recursión cada uno de esos subarreglos se vuelve a dividir, aplicando en cada uno el mismo proceso de intercambio con respecto al pivot. Finalmente, cuando ya no sea posible volver a particionar por haber llegado a subarreglos de tamaño 1, terminar el proceso dejando los subarreglos ordenados de forma que el arreglo final quede ordenado a su vez.

La división del arreglo en dos para volver a tomar pivotes en cada subarreglo, puede hacerse recursivamente: en cada mitad se calcula un nuevo pivot, se vuelven a trasladar los menores y los mayores; y recursivamente se vuelve a dividir, una y otra vez, obteniendo particiones de tamaños  $n/2$ ,  $n/4$ ,  $n/8$ , etc., aplicando recursión sobre ellas hasta que la partición a procesar tenga un solo elemento.

¿Qué tan bueno es este algoritmo en tiempo de ejecución? Podemos ver que la función *quick()* comienza haciendo primero el cálculo del pivot y el traslado de los menores y mayores con respecto a ese pivot. Si se analiza con cuidado ese proceso, puede verse que se ejecuta en tiempo lineal ( $O(n)$ ): los dos ciclos más internos mueven los índices  $i$  y  $j$  una vez por cada iteración, y aun cuando ambos están dentro de otro ciclo, este ciclo externo solo controla que  $i$  y  $j$  no se crucen, pero ***no fuerza*** a los ciclos internos a comenzar nuevamente desde cero (lo que provocaría un rendimiento cuadrático).

Luego de terminar el *proceso de pivoteo*, la función *quick()* hace *dos llamadas recursivas* tomando cada una el subarreglo que corresponda (el primero, delimitado por los índices *izq* y *j*, y el segundo delimitado por *i* y *der*). Para analizar el comportamiento de la función *quick()* en cuanto a su tiempo de ejecución total, veámosla en un esquema de pseudocódigo simplificado:

```
quick(subarreglo actual):
    x = pivot del subarreglo actual
    pivotear: trasladar menores y mayores a x [==> tiempo adicional: O(n)]
    quick(mitad izquierda)
    quick(mitad derecha)
```

Podemos ver que el proceso completo consta de dos llamadas recursivas (cuyo tiempo de ejecución o costo ignoramos ahora) más la ejecución del proceso **pivotear** que en este caso sabemos que es de **tiempo lineal ( $O(n)$ )**. Esto significa que el tiempo total empleado por la función *quick()* completa, dependerá del tiempo que lleve la ejecución de las dos llamadas recursivas. Necesitamos entonces averiguar cuántas invocaciones recursivas serán realizadas y cuánto tiempo empleará cada una.

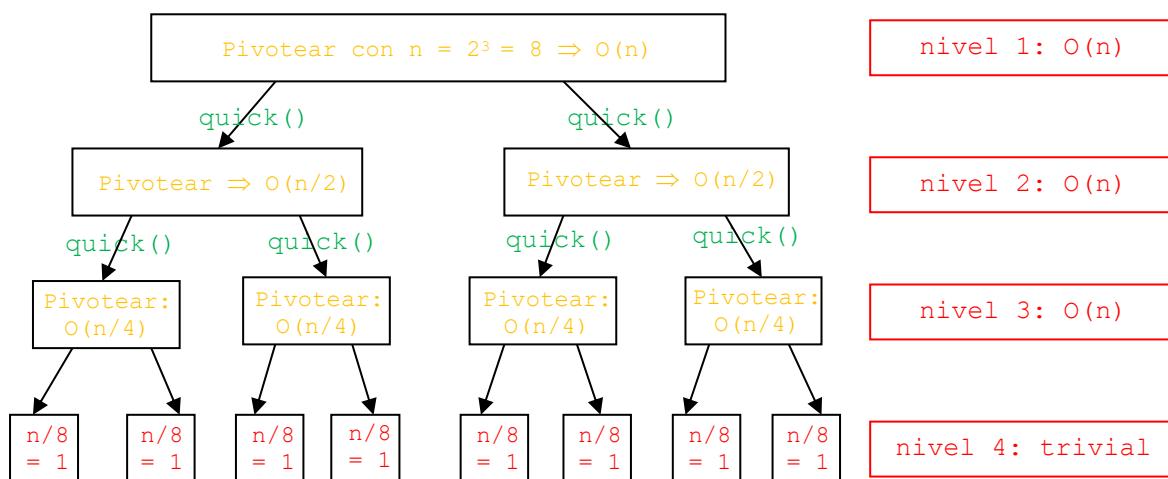
Para ello ayudará un ejemplo. Supongamos que la cantidad de elementos del arreglo es  $n = 8$  (para facilitar la explicación suponemos que  $n$  es una potencia de 2, pero el análisis no cambia para cualquier otro valor). Al ser invocada por primera vez, la función *quick()* hace el primer proceso de pivoteo sobre el arreglo completo (tiempo:  $O(n)$  para cumplir este paso), y luego dos llamadas recursivas (*nivel 1* del gráfico siguiente)

Cada una de las dos instancias recursivas tiene que volver a aplicar el proceso de pivoteo sobre dos subarreglos de tamaño promedio  $n/2 = 4$ . En cada uno demora  $O(n/2)$ , por lo que el tiempo conjunto de ambos lleva a  $O(n/2) + O(n/2) = O(n)$  en el *nivel 2* del gráfico.

Luego en ese mismo nivel 2 se hacen cuatro llamadas recursivas a *quick()* para otros cuatro subarreglos de tamaño promedio  $n/4$ , por lo cual el proceso de pivoteo completo para ese nivel es  $O(n/4) + O(n/4) + O(n/4) + O(n/4) = O(n)$  también en el *nivel 3*.

En cada nivel de llamadas recursivas, el tamaño promedio de cada subarreglo se divide por 2, y aunque es cierto que el proceso de pivoteo trabaja entonces cada vez menos, el hecho es que ese pivoteo se hace varias veces por nivel, con un tiempo de ejecución combinado que siempre es  $O(n)$ . ¿Cuántos *niveles* de llamadas a *quick()* tiene el árbol? La respuesta es:  $k = 3 = \log_2(8) = \log_2(n)$  [o sea: tantos como se pueda dividir sucesivamente a  $n$  por 2 hasta llegar a un cociente de 1 (lo que finalmente es igual a  $\log_2(n)$ )]. En notación *Big O* se puede prescindir de la base del logaritmo, por lo que la cantidad de niveles con llamadas a *quick()* es  $O(\log(n))$ .

**Figura 9:** La estrategia *Divide y Vencerás* aplicada en el algoritmo *Quicksort*.

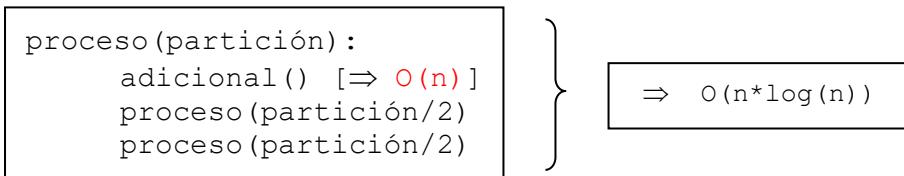


En la gráfica anterior se muestra el nivel 4 del árbol, aunque en realidad ese nivel no llega a generarse: cuando un subarreglo tiene tamaño 1, no hay invocación recursiva para él ya que teniendo un único componente está ya ordenado. Por lo tanto, si la cantidad total de niveles de invocación a *quick()* es  $O(\log(n))$  y cada nivel se procesa en un tiempo de ejecución combinado  $O(n)$  entonces el tiempo total de ejecución es  $O(n * \log(n))$  lo cual es

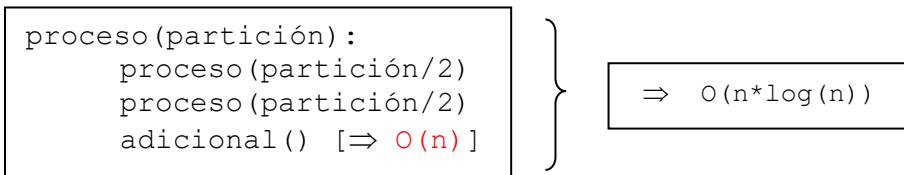
*subcuadrático*: puede verse que  $n * \log(n) < n * n = n^2$  por lo que  $O(n * \log(n)) < O(n^2)$ . Esto prueba que *en promedio*, el algoritmo *Quicksort* ejecuta en tiempo  $O(n * \log(n))$  (decimos *en promedio*, porque el tamaño de cada subarreglo no es constantemente igual a  $n/2$  en cada nivel, pero si los datos vienen aleatoriamente dispuestos se puede esperar que los subarreglos tengan tamaños que *en promedio* se asemejen a  $n/2$ , compensando las diferencias).

¿Qué tan general es este resultado? La estrategia *divide y vencerás* se aplicó para diseñar un algoritmo de ordenamiento y al analizarlo hemos obtenido un tiempo de ejecución promedio de  $O(n * \log(n))$  ¿Puede inferirse que siempre será así? ¿Toda vez que se aplique la estrategia *divide y vencerás* se obtendrá una solución con tiempo de ejecución  $O(n * \log(n))$  para el caso promedio o para el peor caso? Se puede probar que la respuesta es **sí siempre y cuando el algoritmo analizado tenga la misma estructura que el algoritmo quick()**: dos llamadas recursivas aplicadas sobre lotes de datos de tamaño aproximadamente igual a la mitad del lote actual, más *un proceso adicional de tiempo de ejecución lineal ( $O(n)$ )* (como nuestro proceso de *pivoteo*):

Figura 10: Esquema esencial de un algoritmo *Divide y Vencerás* con tiempo de ejecución  $O(n * \log(n))$ .



O bien:



## 6.] Implementación del Quicksort mediante selección del pivot por *Mediana de Tres*.

Hemos visto que el algoritmo *Quicksort* ejecuta con tiempo promedio de  $O(n * \log(n))$ . Aún cuando ese comportamiento es muy bueno, existen casos de configuraciones de entrada que pueden llevar al *Quicksort* a un *peor caso* muy malo en cuanto a tiempo de ejecución, dependiendo de la forma en que se tome el pivot: puede probarse que si los elementos del arreglo están distribuidos de forma tal que cada vez que se toma el pivot su valor es el *menor* (o el *mayor*) en el subarreglo considerado, entonces *Quicksort* ejecuta en tiempo  $O(n^2)$  (*cuadrático...*) [1].

Intuitivamente puede verse que esto es cierto: si recurrentemente se toma el *menor* o el *mayor* del subarreglo a pivotear, entonces se producirán dos particiones pero una de ellas tendrá un solo elemento y la otra tendrá  $n-1$  elementos... y al repetirse este patrón, el árbol de llamadas recursivas tendrá  $n$  niveles en lugar de  $\log(n)$  niveles, por lo que surge el orden cuadrático: si cada nivel se pivotea a un costo de  $O(n)$ , y tenemos  $O(n)$  niveles, entonces el tiempo total combinado será  $O(n^2)$ .

Las distribuciones de datos que pueden causar que *Quicksort* se degrade a un rendimiento cuadrático dependen de la forma en que se selecciona el pivot. Algunos programadores suelen elegir como pivot al *primer elemento del subarreglo analizado* (o bien suelen elegir el *último*). Sin embargo, estas dos formas simples de selección del pivot son justamente las que *maximizan la posibilidad de caer en un rendimiento cuadrático*: si el arreglo estuviese ya ordenado, el algoritmo demoraría un tiempo de orden cuadrático en no hacer nada más que comparaciones, sin intercambio alguno. Por lo tanto, es una muy mala idea tomar como pivot a alguno de los elementos de los extremos en cada partición [\[6\]](#) [\[1\]](#).

La variante que hemos mostrado implementada en las secciones anteriores de esta Ficha, toma como pivot al *elemento central del subarreglo analizado*, y esta es una estrategia muy estable incluso si el arreglo ya estuviese ordenado (en este caso, se invierte un tiempo de  $O(n * \log(n))$  en recorrer el arreglo, sin hacer intercambios). Sin embargo, aún puede ocurrir que aparezca alguna distribución de datos en la cual el elemento central elegido siempre sea el menor o el mayor. Hemos dicho que esa distribución es altamente improbable, pero aún así muestra que la estrategia de selección del pivot podría mejorarse todavía más.

La estrategia más usada se conoce como *selección del pivot por mediana de tres*, y permite eliminar por completo la posibilidad de caer en el peor caso, incluso para distribuciones de datos muy malas o adversas. Esta variante consiste en seleccionar como pivot al *valor mediano entre el primero, el último y el central* del subarreglo analizado (la *mediana* de un conjunto de  $n$  elementos, es el valor  $x$  tal que la mitad de los elementos del conjunto son menores a  $x$ , y la otra mitad son mayores). La forma obvia de obtener el valor mediano de un conjunto es ordenar ese conjunto y luego limitarse a tomar el valor que haya quedado en el centro [\[1\]](#).

En la implementación del *Quicksort por Mediana de Tres*, en cada partición se toma el valor del extremo izquierdo, el valor del extremo derecho y el valor central de la partición, se los ordena entre ellos, y se toma como pivot al que finalmente haya quedado en la casilla central. De esta forma, se garantiza que nunca se tomará como pivot al mayor o al menor, evitando el peor caso antes mencionado. También se suele aconsejar que la *mediana de tres* se tome entre tres elementos seleccionados aleatoriamente dentro de cada partición, lo cual efectivamente lleva a tiempos de ejecución ligeramente mejores.

Por otra parte, se han ensayado otras variantes tomando como pivot al *valor mediano entre cinco* o más componentes del subarreglo, pero las pruebas de rendimiento no han mostrado una mejora significativa en el algoritmo (además de ser más complicadas de programar). Por lo tanto, la estrategia de *selección de pivot por mediana de tres* se ha consolidado como una de las más usadas en la implementación del *Quicksort* (tomando o no los tres elementos en forma aleatoria).

Para finalizar este breve análisis, mostramos la implementación del algoritmo *Quicksort por Mediana de Tres*, tomando la mediana entre el primero, el central y el último de cada partición. Dejamos para los alumnos la implementación con la variante de tomar los tres elementos en forma aleatoria:

**Problema 60.)** *Implementar el algoritmo Quicksort, pero de forma que la selección del pivot en cada partición se realice por la técnica de la Mediana de Tres.*

**Discusión y solución:** Lo único que debemos definir es la forma en que puede obtenerse la mediana entre los tres elementos seleccionados (el primero, el central y el último) de cada

partición. El resto del algoritmo es exactamente el mismo que ya hemos mostrado en esta misma Ficha.

Si el arreglo a ordenar es  $v$ , y la partición en la que se debe tomar el pivot está delimitada por los índices  $izq$  y  $der$ , entonces la siguiente función simple reordena los tres elementos  $v[izq]$ ,  $v[der]$  y  $v[central]$ , dejando en  $v[central]$  al valor mediano (que finalmente será retornado):

```
def get_pivot_m3(v, izq, der):
    # calculo del pivot: mediana de tres...
    central = (izq + der) // 2
    if v[der] < v[izq]:
        v[der], v[izq] = v[izq], v[der]
    if v[central] < v[izq]:
        v[central], v[izq] = v[izq], v[central]
    if v[central] > v[der]:
        v[central], v[der] = v[der], v[central]
    return v[central]
```

Los detalles de funcionamiento de esta función son simples y directos, por lo que dejamos su análisis para el estudiante. El algoritmo completo se implementa con las siguientes funciones (incluida la anterior) que forman parte del proyecto [F27] *Divide y Vencerás* que acompaña a esta Ficha:

```
def quick_sort_m3(v):
    # ordenamiento Quick Sort
    quick_m3(v, 0, len(v) - 1)

def quick_m3(v, izq, der):
    x = get_pivot_m3(v, izq, der)

    i, j = izq, der
    while i <= j:

        while v[i] < x and i < der:
            i += 1

        while x < v[j] and j > izq:
            j -= 1

        if i <= j:
            v[i], v[j] = v[j], v[i]
            i += 1
            j -= 1

        if izq < j:
            quick(v, izq, j)

        if i < der:
            quick(v, i, der)

def get_pivot_m3(v, izq, der):
    # calculo del pivot: mediana de tres...
    central = int((izq + der) / 2)

    if v[der] < v[izq]:
        v[der], v[izq] = v[izq], v[der]
```

```

if v[central] < v[izq]:
    v[central], v[izq] = v[izq], v[central]

if v[central] > v[der]:
    v[central], v[der] = v[der], v[central]

return v[central]

```

El programa *main()* del proyecto [F27] *Divide y Vencerás* es esencialmente el mismo que ya se mostró en la Ficha 17: un menú que permite crear un arreglo de tamaño  $n$  con valores aleatorios, y numerosas opciones para ordenar el arreglo con cualquiera de los algoritmos que hemos visto. La diferencia es que ahora se incorpora una opción más para lanzar el *Quicksort por Mediana de Tres*.

## 7.] La Función de Ackermann.

La *Función de Ackermann* es una función recursiva matemática que debe su nombre a quien la postuló en 1926: *Wilhelm Ackermann* [7]. La *Función de Ackermann* se basa en aplicar un operador simple (la suma) combinado con numerosas invocaciones recursivas, con la intención de lograr *un crecimiento muy rápido* en la magnitud de los valores calculados. Toda la explicación necesaria para comprender su uso y la forma de programarla, se expone en el problema que sigue:<sup>5</sup>

**Problema 61.)** *La Función de Ackermann originalmente planteada por Wilhelm Ackermann en 1926 ha sido ajustada a lo largo de los años hasta terminar siendo definida en la forma que se muestra aquí:*

<b>Función de Ackermann</b>
-----------------------------

$$\begin{aligned}
 A(m, n) &= \begin{cases} n + 1 & (\text{si } m = 0) \\ A(m - 1, 1) & (\text{si } m > 0 \text{ y } n = 0) \\ A(m - 1, A(m, n - 1)) & (\text{si } m > 0 \text{ y } n > 0) \end{cases} \\
 &\quad (m, n \text{ enteros} \\
 &\quad m \geq 0, n \geq 0)
 \end{aligned}$$

*Se pide desarrollar un programa que permita calcular el valor de la Función de Ackermann para distintos valores de  $m$  y  $n$ .*

**Discusión y solución:** La función originalmente planteada por *Ackermann* tenía tres parámetros en lugar de dos. Como dijimos, a lo largo de los años se han ido sugiriendo variantes y modificaciones que terminaron conformando una *familia o serie de funciones de Ackermann*, basadas en estructuras similares. La definición que mostramos en el enunciado es una variante que fue propuesta por *Rózsa Peter* y *Raphael Robinson*, aunque es común que se la cite como si fuese la original de *Ackermann* [8]. En algunos contextos, la variante de *Peter* y *Robinson* se conoce también como la *Función de Ackermann-Peter*.

---

<sup>5</sup> La fuente esencial de la explicación que sigue sobre la *Función de Ackermann*, es la *Wikipedia* (especialmente la versión en inglés): [https://en.wikipedia.org/wiki/Ackermann\\_function](https://en.wikipedia.org/wiki/Ackermann_function). También hemos usado una referencia básica contenida en el libro "Estructuras de Datos con C y C++" de *Yedidyah Langsam* (y otros), así como alguna cita del libro "Estructuras de Datos en Java" del ya citado *Mark Allen Weiss*.

La función está definida directamente en forma recursiva, por lo que su programación también es directa y no ofrece mayores problemas (ver el proyecto F[27] Ackermann que acompaña a esta Ficha):

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

Técnicamente hablando, el problema está resuelto. Sin embargo, conviene tomarse un tiempo para estudiar esta función y sus propiedades. Lo primero que se observa es que sólo la primera parte de la definición es *no recursiva*: si  $m = 0$  entonces  $A(0, n) = n + 1$ . Esto significa que lo que sea que termine calculando, será el resultado de aplicar ese único operador directo (la suma), para añadir 1 al valor de  $n$ .

El siguiente programa (incluido en el mismo proyecto F[27] Ackermann) calcula diversos valores para la función haciendo que  $m$  tome todo valor posible en el intervalo  $[0, 3]$  y  $n$  tome valores del intervalo  $[0, 6]$ :

```
def test():
    for m in range(4):
        for n in range(7):
            ack1 = ackermann(m, n)
            print('Ackermann(', m, ', ', n, ') = ', ack1, '\t-\t', sep='', end='')
    print()

# script principal...
test()
```

Podría parecer entonces que la función entregará resultados que no serán de valor muy alto, ya que todos los cálculos se hacen en base a sumar 1 al valor actual de  $n$  en cada instancia recursiva. Pero la realidad es que para valores combinados de  $m$  y  $n$  relativamente bajos, la función llega a resultados asombrosamente elevados. Mientras el valor de  $m$  se mantenga menor a 4 y el valor de  $n$  se mantenga menor a 7, la función devuelve valores de magnitud aceptable, como se puede ver en la siguiente salida generada por el programa anterior:

```
A(0,0): 1   A(0,1): 2   A(0,2): 3   A(0,3): 4   A(0,4): 5   A(0,5): 6   A(0,6): 7
A(1,0): 2   A(1,1): 3   A(1,2): 4   A(1,3): 5   A(1,4): 6   A(1,5): 7   A(1,6): 8
A(2,0): 3   A(2,1): 5   A(2,2): 7   A(2,3): 9   A(2,4): 11  A(2,5): 13  A(2,6): 15
A(3,0): 5   A(3,1): 13  A(3,2): 29  A(3,3): 61  A(3,4): 125 A(3,5): 253 A(3,6): 509
```

Si el valor inicial de  $m$  es 0, la función directamente retorna  $n + 1$ , y eso produce que la primera fila de la tabla anterior simplemente contenga la progresión de los números naturales para los resultados. Las dos filas que siguen se mantienen con valores de salida pequeños. Pero la cuarta fila ya comienza a producir resultados bastante mayores. Para  $A(3,5)$  el resultado ya es 253 y para  $A(3, 6)$  se obtiene un 509.

Evidentemente, cuando los valores combinados de  $m$  y  $n$  comienzan a crecer, la función realiza una cantidad muy alta de invocaciones recursivas, y eso hace que el valor de salida sea muy alto también (aunque se obtenga sólo con la acción de sumar 1...) Si intentamos calcular  $A(3,7)$  el programa se interrumpe por *overflow de stack* sin llegar a mostrarnos el

resultado: la cantidad de instancias recursivas es tan grande que no puede ser soportada por el *Stack Segment*.

Lo anterior lleva a que nos preguntemos si la función está efectivamente bien planteada, o si por el contrario, pudiera estar entrando en una situación de recursión infinita que sea la responsable del overflow de stack. Al fin y al cabo, no parece evidente que  $A(m,n)$  llegue a terminar de calcularse alguna vez para valores de  $m$  y  $n$  bastante mayores que 0, ya que la segunda parte de la definición de la función hace una llamada recursiva, pero la tercera hace algo más complejo aún: *una invocación recursiva compuesta con otra* (la primera toma como parámetro al resultado de la segunda, haciendo una *composición de funciones*).

Sin embargo, un análisis detallado del comportamiento de la función nos lleva a concluir que la función está bien planteada y no produce una regresión infinita: si  $m$  y  $n$  son diferentes de cero, ambos son restados en 1 en algún momento durante la cascada recursiva, pero se comienza restando 1 a  $n$ . Cuando el valor de  $n$  llega a ser 0, comienza desde allí a restarse 1 a  $m$ , lo que eventualmente llevará a  $m = 0$  y a la detención de la cascada recursiva. El tema es que hasta que eso ocurra, se creará una inmensa cantidad de instancias recursivas que hará que los valores retornados sean cada vez mayores, y provocando también el rebalsamiento de stack.

Para terminar de comprender la magnitud de los valores que llega a calcular la función, note que  $A(4,0)$  vale 13. El valor  $A(4,1)$  ya pasa a ser 65333 (aunque nuestra función posiblemente ya no pueda calcularlo por producirse un overflow de stack). Y el valor  $A(4,2)$  es tan inmenso que no puede escribirse sino en forma exponencial: ese valor es igual a  $2^{65536} - 3$  (de hecho, este número resulta ser mayor que  $u^{200}$  donde  $u$  es la cantidad total de partículas contenidas en el universo). Y se pone mucho peor: los valores de la función para  $m > 4$  y  $n > 1$  son tan extravagantemente enormes que no podrían escribirse nunca en forma normal, ya que la secuencia de dígitos que conforma a cada uno de esos números *no cabe en el universo conocido* (y por supuesto, ni nuestro programa ni ningún otro podría siquiera comenzar a calcular esos valores, ya que mucho antes de llegar al final se produciría un overflow de stack).

Para finalizar, digamos que la *Función de Ackermann* tiene ciertos usos prácticos: por lo pronto, debido a su naturaleza profundamente recursiva, puede utilizarse para testear la habilidad de un compilador en optimizar un proceso recursivo. En general, cuando se compila un programa recursivo, el compilador intenta eliminar la recursión y producir una versión compilada iterativa, que será más eficiente en cuanto a uso de memoria y posiblemente en velocidad de ejecución. La *Función de Ackermann* puede ser entonces uno de los desafíos que se le imponen a un compilador cuando esté en etapa de prueba (o *benchmarking*).

Además, dado que la *Función de Ackermann tomada para m = n* (*o sea: A(n,n)*, que a su vez puede simplificarse como  $A(n)$ ) crece de forma tan violenta, entonces su inversa (es decir, la función que a cada número y calculado por  $A(n)$  le asigna el mismo valor  $n$  de partida) normalmente designada con la *letra griega alfa*:  $\alpha(n) = A^{-1}(n)$  aumenta de manera sorprendentemente lenta. De hecho, la función  $\alpha(n)$  retorna valores menores que 5 para prácticamente cualquier valor de  $n$  que se tome como parámetro. Esta propiedad de la *inversa de Ackermann* de crecer tan lentamente, suele usarse en el contexto del *análisis de algunos algoritmos* que se sabe que tienen un tiempo de ejecución que crece de forma muy lenta a medida que aumenta el número de datos.

## Bibliografía

---

- [1] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.
- [2] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: <https://docs.python.org/3/>. [Accessed 24 February 2016].
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [4] V. Frittelli, R. Teicher, M. Tartabini, J. Fernández and G. F. Bett, "Gestión de Gráficos (en Java) para Asignaturas de Programación Introductiva," in *Libro de Artículos Presentados en I Jornada de Enseñanza de la Ingeniería - JEIN 2011*, Buenos Aires, 2011.
- [5] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [6] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [7] Wikipedia, "Ackermann function" 2015. [Online]. Available: [https://en.wikipedia.org/wiki/Ackermann\\_function](https://en.wikipedia.org/wiki/Ackermann_function).
- [8] Y. Langsam, M. Augenstein and A. Tenenbaum, Estructura de Datos con C y C++, México: Prentice Hall, 1997.

# Ficha 28

## Estrategias de Resolución de Problemas: Algoritmos Ávidos, Programación Dinámica y Backtracking

### 1.] Introducción.

La tarea de encontrar y plantear un algoritmo que resuelva un problema dado (cosa que debe hacer casi todo el tiempo un programador) requiere conocimientos en el campo de ese problema, pero también paciencia, creatividad, ingenio, disciplina para realizar pruebas, capacidad de autocrítica... y aún así en muchos casos encontrar una solución no es un proceso simple ni obvio. A veces un programador encuentra una solución que parece correcta pero luego descubre fallas o demuestra que la misma no es correcta para todos los casos. Otras veces se encuentra una solución, pero luego se comprueba que la misma es poco eficiente (ya sea porque su tiempo de ejecución es alto o porque ocupa demasiada memoria, por ejemplo). Y muchas veces la solución simplemente no aparece [1].

En este sentido, a lo largo de los años los investigadores y los especialistas en el área de los algoritmos han estudiado y propuesto diversas técnicas básicas que ayudan a plantear algoritmos para situaciones específicas. Estas técnicas *no implican una garantía de éxito*, sino simplemente un punto de partida para intentar encontrar una solución frente a un problema particular. Si el programador conoce lo suficiente acerca del problema, y domina los principios de aplicación de cada técnica, *posiblemente* podrá entonces plantear alguna de ellas para intentar dar con una solución al problema. Nada garantiza que lo logre, y si lo logra, nada garantiza que la solución encontrada sea eficiente, pero incluso así el programador habrá tenido elementos para comenzar a trabajar y al menos poder descartar algunos caminos.

Tradicionalmente, las técnicas o estrategias de planteo de algoritmos que se sugieren como básicas son las siguientes:

- **Fuerza Bruta:** Consiste en explorar y aplicar sistemáticamente, una por una, todas y cada una de las posibles combinaciones de solución para el problema dado. A modo de ejemplo, el algoritmo de *ordenamiento de Selección Directa* es esencialmente un algoritmo de *fuerza bruta*. Por lo general, los algoritmos obtenidos por *fuerza bruta* resultan intuitivamente simples de comprender e implementar (de hecho, un algoritmo de fuerza bruta suele ser lo primero que se le ocurre a un programador) pero por otra parte suelen ser muy ineficientes (o bien son muy lentos si el tamaño de la entrada es grande, o bien ocupan demasiados recursos de memoria) ya que la esencia del planteo consiste justamente en no ahorrar pasos.

En la medida de lo posible, si se tiene un algoritmo de *fuerza bruta* para un problema, el programador debería esforzarse en refinarlo, eliminar elementos de redundancia lógica o agregar elementos que permitan ahorrar trabajo y tratar de obtener una solución mejor. Sin

embargo, es notable que para muchos problemas muy estudiados y comunes, sólo se conocen soluciones generales de *fuerza bruta*, aunque para casos especiales de esos problemas se aplican estrategias de optimización que permiten mejorar el rendimiento de la solución. Un conocido ejemplo de esta situación, es el famoso *Problema del Viajante* (o del *Vendedor Ambulante*): dadas  $n$  ciudades y las distancias entre ellas, establecer un recorrido que permita pasar una sola vez por todas ellas pero recorriendo la menor distancia total. La solución trivial de fuerza bruta enumera todos los posibles recorridos y se queda con el más corto, pero eso lleva a un tiempo de ejecución  $O(n!)$ , que lo vuelve inaplicable si el número de ciudades se hace mayor a 20. Hasta hoy, una de las mejores soluciones se basa en técnicas de *Programación Dinámica* con tiempos de ejecución  $O(n^2 2^n)$ . No se conoce una solución general más eficiente para todos los casos de este problema, pero se pueden aplicar estrategias que mejoren la situación si se sabe que el número de ciudades es pequeño, o *estrategias subóptimas* que obtienen soluciones aproximadas aunque no necesariamente óptimas.

- **Recursión:** Como sabemos, la *recursión* o *recursividad* es la propiedad que permite que un proceso se invoque a sí mismo una o más veces como parte de la solución. Si se establece que un problema puede ser entendido en base a versiones más pequeñas y manejables de si mismo, entonces un *planteo recursivo* puede ayudar a lograr un algoritmo que normalmente será muy claro para entender e implementar, al costo de utilizar cierta cantidad de memoria adicional en el segmento de stack del computador (y algo de tiempo extra para gestionar el apilamiento en ese stack). En muchos casos, este costo no es aceptable y es preferible plantear un *algoritmo no recursivo* que resuelva el mismo problema. Muchos ejemplos conocidos que se usan para explicar el funcionamiento de la *recursividad* (cálculo del factorial de un número, cálculo de un término de la secuencia de Fibonacci, etc.) son justamente casos en los que la *recursión no* es aconsejable.

A pesar de los costos extra en uso de memoria y tiempo de ejecución, para algunos problemas de lógica compleja la recursión constituye una herramienta que posibilita el planteo lógico en forma clara y concisa, frente a planteos no recursivos extensos, intrincados y sumamente difíciles de mantener frente a cambios en los requerimientos. Casos así, por ejemplo, se dan en algoritmos (que hemos visto) para generar *gráficas fractales* o en situaciones más conocidas como la implementación de *algoritmos de inserción y borrado en árboles de búsqueda equilibrados*.

- **Vuelta Atrás (Backtracking):** Se trata de una técnica que permite explorar en forma incremental un conjunto de potenciales soluciones (*soluciones parciales*) a un problema, de manera que si se detecta que una *solución parcial no puede* ser una solución válida, se la descarta junto a todas las candidatas que podrían haberse propuesto a partir de ella. Típicamente, se implementa mediante *recursión* generando un árbol de invocaciones recursivas en el que cada nodo constituye una solución parcial. Como cada nueva invocación recursiva agrega un nodo en el nivel siguiente del árbol por la inclusión de un nuevo paso simple en el proceso, entonces es fácil ver que si se llega a un punto en el cual se obtiene una solución no válida, puede anularse toda la rama que llevó a esa solución y continuar el análisis en ramas vecinas. Cuando es aplicable, el *backtracking* suele ser más eficiente que la enumeración por *fuerza bruta* de todas las soluciones, ya que con *backtracking* pueden eliminarse muchas soluciones sin tener que analizarlas.

La estrategia de *backtracking* se usa en problemas que admiten la idea de solución parcial, siempre y cuando se pueda comprobar en forma aceptablemente rápida si una solución parcial es válida o no. Ejemplos muy conocidos de aplicación se dan con el problema de las *Ocho Reinas* (tratar de colocar ocho reinas en un tablero de ajedrez, sin que se ataquen entre ellas), o en el planteo de soluciones para juegos de *Palabras Cruzadas*, o en el planteo de tableros de *Sudoku* o en general, en problemas de *optimización combinatoria*.

- **Algoritmos Ávidos o Devoradores (Greedy Algorithms):** Un *algoritmo ávido* es aquel que aplica una regla intuitivamente válida (una *heurística*) en cada paso local del proceso, con la *esperanza* de obtener finalmente una solución global óptima que resuelva el problema original. Para muchos problemas esta estrategia no produce una solución óptima, pero suele servir de todos modos para intentar plantear soluciones óptimas locales, que luego puedan aproximar una solución óptima final en un tiempo aceptable.

La ventaja de un *algoritmo ávido* (cuando es correcto) es que por lo general lleva a una solución simple de entender, muy directa de implementar y razonablemente eficiente. Pero la desventaja es que como no siempre lleva a una solución correcta, se debe realizar una *demostración de la validez* del algoritmo que podría no ser sencilla de hacer. En general, si se puede demostrar que una *estrategia ávida* es válida para un problema dado, entonces la misma suele producir resultados más eficientes que otras estrategias como la *programación dinámica*.

Algunos ejemplos de aplicación conocidos en los cuales un *algoritmo greedy* se aplica con éxito son los algoritmos de *Prim* y de *Kruskal* para obtener el *árbol de expansión mínimo de un grafo*, el algoritmo de *Dijkstra* para obtener el *camino más corto entre nodos de un grafo*, y el algoritmo de *Huffman* para construir árboles que permitan obtener la *codificación binaria óptima para una tabla de símbolos*, de forma de reducir el espacio ocupado por un mensaje armado con esos símbolos.

- **Divide y Vencerás:** Consiste en tratar de dividir el lote de datos en dos o más subconjuntos de tamaños aproximadamente iguales, procesar cada subconjunto por separado y finalmente unir los resultados para obtener la solución final. Normalmente, se aplica *recursión* para procesar a cada subconjunto y el proceso total podrá ser más o menos eficiente en cuanto a tiempo de ejecución dependiendo de tres factores: la *cantidad de invocaciones recursivas* que se hagan, el *factor de achicamiento del lote de datos* (por cuanto se divide al lote en cada pasada) y el *tiempo que lleve procesar en forma separada a un subconjunto*. Ejemplos muy conocidos de aplicación exitosa de la técnica de *divide y vencerás*, son los algoritmos *Quicksort* y *Mergesort* para ordenamiento de arreglos.
- **Programación Dinámica:** Esta técnica sugiere almacenar en una tabla las soluciones obtenidas previamente para los subproblemas que pudiera tener un problema mayor, de forma que cada subproblema se resuelva sólo una vez y luego simplemente se obtengan sus soluciones consultando la tabla si esos subproblemas volvieran a presentarse. Esto tiene mucho sentido: en muchos planteos originalmente basados (por ejemplo) en *divide y vencerás*, suele ocurrir que al dividir un problema en subproblemas se observe que varios de estos últimos se repiten más de una vez, con la consecuente pérdida de tiempo que implicaría el tener que volver a resolverlos.

La idea de la *programación dinámica* es entonces resolver primero los subproblemas más simples, y luego ir usando esas soluciones hacia arriba para combinarlas y resolver problemas con entradas mayores. Esto es especialmente útil en problemas en los que el número de subproblemas que se repiten crece en forma exponencial. Se aplica en *problemas de optimización*: encontrar la mejor solución posible entre varias alternativas presentes. Algunos ejemplos en los que la programación dinámica se aplica con éxito son el algoritmo de *Bellman-Ford* y el algoritmo de *Floyd-Warshall* para encontrar el *camino más corto entre dos vértices de un grafo*, o los diversos algoritmos conocidos para *alineación de secuencias*, cuyo objetivo es encontrar la *mínima cantidad de cambios que deben hacerse en una secuencia de entrada para convertirla en otra*.

- **Randomización (Algoritmos de Base Aleatoria):** Las estrategias para planteo de algoritmos que hemos enumerado hasta aquí no son las únicas posibles, pero son quizás las más conocidas. Todas ellas se basan en el intento de arribar a un algoritmo (más o menos

eficiente) que resuelva un problema, y en todas la idea es que ese algoritmo sea *determinista*. La característica de un algoritmo *determinista* es que ante una misma entrada, produce siempre la misma salida. Cada paso que el algoritmo aplica es una consecuencia lógica del estado en que se encontraba en el paso anterior y por lo tanto, un algoritmo determinista correctamente planteado siempre produce la solución correcta al problema analizado.

Está claro que siempre queremos que un algoritmo sea correcto, pero también esperaríamos que un algoritmo sea *eficiente* al menos en su tiempo de ejecución. Sin embargo, muchos problemas no son tan simples. Existen problemas que recurrentemente aparecen y requieren soluciones prácticas, pero no siempre esas soluciones prácticas son eficientes. Si la cantidad de posibles soluciones fuese exponencial (por ejemplo), un *algoritmo determinista de Fuerza Bruta* tendría un tiempo de ejecución exponencial ( $O(2^n)$ ) lo cual es inaplicable incluso para valores de  $n$  pequeños.

Frente a estos casos existe la alternativa de plantear una *estrategia de Randomización* (o de *Aleatorización*). La idea es intentar plantear algoritmos que ya *no sean deterministas*, sino de *base aleatoria*: ahora, ante la misma entrada, el algoritmo podría producir salidas diferentes ya que el siguiente paso a aplicar surge de algún tipo de *selección aleatoria*. Y está claro que si interviene el azar, entonces es posible que el algoritmo no llegue eventualmente a una solución correcta.

Lo anterior implica que si se piensa seriamente en aplicar un *algoritmo randomizado* para un problema dado, ese algoritmo debe ser cuidadosamente analizado desde varios aspectos:

- Por lo pronto, debe quedar claro que el algoritmo valga la pena en cuanto a tiempo de ejecución esperado (por ejemplo, si obtiene un tiempo de ejecución polinómico o que combine una expresión polinómica con otra logarítmica) Si se aplica un *algoritmo randomizado* que de todos modos tendrá un tiempo de ejecución exponencial, no habrá ningún beneficio y todavía se tendrá la desventaja de eventualmente no obtener la solución correcta.
- Si bien se sabe que un *algoritmo randomizado* podría no obtener la solución correcta, se espera poder conocer con precisión cuál es la *probabilidad de que el algoritmo falle*, y diseñarlo de tal modo que esa *probabilidad sea realmente baja o muy baja*.
- Y finalmente, en la medida de lo posible, se esperaría que el algoritmo planteado sea relativamente *simple de implementar* (cosa que suele ser el caso de los *algoritmos randomizados*).

## 2.] Conceptos básicos sobre Algoritmos Ávidos y Programación Dinámica.

Como dijimos, los investigadores y programadores han caracterizado algunas técnicas generales que a menudo llevan a algoritmos eficientes para la resolución de ciertos problemas. En fichas anteriores hemos aplicado con mayor o menor profundidad técnicas de *fuerza bruta* y de *recursividad* para resolver algunos problemas y situaciones prácticas (en general, son algoritmos de *fuerza bruta* los que hemos empleado para buscar el menor o el mayor en un arreglo y los algoritmos de ordenamiento simples, mientras que son algoritmos *recursivos* los que hemos visto para generar gráficas fractales) En esta ficha en particular, haremos una introducción a los principios básicos de los *algoritmos ávidos* y de la *programación dinámica*.

Sin embargo, es bueno recordar que existen algunos problemas para los cuales ni éstas ni otras técnicas conocidas han producido hasta ahora soluciones eficientes. Cuando se encuentra algún problema de este tipo suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta y más difícil de calcular.

Un conocido problema que suele usarse para mostrar la aplicación de los *algoritmos ávidos* y de la *programación dinámica*, es el *problema del cambio en monedas*, que enunciamos de la siguiente forma:

**Problema 62.) Problema del Cambio de Monedas:** Se trata de plantear un programa que pueda calcular la **mínima cantidad de monedas** en la que puede cambiarse una cierta cantidad  $x$  de centavos, conociendo los valores nominales de las monedas disponibles. Supongamos que disponemos de un conjunto de monedas **coins** de 1, 5, 10 y 25 centavos. Nos dan una cantidad  $x$  de centavos, y nos piden calcular la mínima cantidad de monedas de  $M$  en la que puede expresarse (o cambiarse) el valor  $x$ .

Así, si  $x = 25$  entonces la mínima cantidad de monedas de **coins** = {1, 5, 10, 25} que podemos usar es 1, ya que disponemos de una moneda de exactamente 25 centavos. Si  $x = 31$ , entonces la mínima cantidad de monedas es 3: una de 25, una de 5 y una más de 1 centavo.

Se pide explorar distintas posibilidades algorítmicas para este problema, y plantear los programas correspondientes.

**Discusión y solución:** Podemos dar diversas soluciones a este problema, basadas en distintas estrategias algorítmicas. Una forma muy intuitiva (*que no siempre funciona*), consiste en elegir sistemáticamente la moneda de mayor valor, devolver tantas de ella como se pueda, y luego continuar así con las monedas de mayor valor que siguen hasta cubrir el valor pedido. Así, si el conjunto de valores de monedas disponible es **coins** = [1, 5, 10, 25] y el valor a cambiar es  $x = 63$ , elegimos dos veces la moneda de 25, una vez la de 10 y tres la de 1, lo cual da un total de 6 monedas (que efectivamente, en el contexto planteado es la mínima cantidad de monedas para llegar a cubrir  $x = 63$ ).

Note que es exigible que la moneda de valor 1 exista para que el problema tenga solución, pues de otro modo habría valores de  $x$  que serían imposibles de cambiar. Si las monedas disponibles son las que hemos supuesto para el conjunto **coins**, se puede probar que esta forma de trabajar *resolverá siempre el problema en forma correcta*, y constituye un ejemplo de la estrategia de resolución de problemas conocida como *algoritmos ávidos* o *devoradores* (o *greedy algorithms*) [2].

En términos generales, un conjunto **coins** de valores monetarios que siempre produce una *solución óptima* para el problema del cambio de monedas (es decir, siempre logra encontrar la menor cantidad posible de monedas para cambiar cualquier valor  $x > 0$ ) aplicando la regla ávida que hemos indicado, se designa como un *sistema de monedas canónico*. Los sistemas monetarios como el de Argentina, Estados Unidos y Europa (por ejemplo) son *canónicos*: con todos ellos, el *algoritmo ávido* descripto en esta sección funcionará correctamente.

Como hemos indicado, un *algoritmo ávido* es aquel en el que en todo momento se toma la decisión que localmente parece la más apropiada u óptima para el caso, y se aplica esa regla una y otra vez sin importar ni analizar si esa decisión traerá malas consecuencias en el futuro, y sin posibilidad de volver atrás y cambiar la decisión en ningún momento.

También hemos visto que la ventaja de las estrategias ávidas es que suelen ser simples de plantear y de entender. Normalmente se ajustan en forma directa al modelo de solución mental que se tiene del problema, por lo que intuitivamente suelen ser las estrategias preferidas si pueden aplicarse. Lo malo, es que no siempre funcionan: si en el problema del cambio de monedas dispusiéramos también de una moneda de 21 centavos en el conjunto *coins*, podemos ver que la estrategia de *algoritmo ávido* que hemos empleado para el cambio de  $x = 63$  seguiría informando que el mínimo es de 6 monedas, cuando es obvio que ahora la solución óptima es 3. En este caso, el conjunto *coins* = [1, 5, 10, 21, 25] no es *canónico*, ya que nuestro *algoritmo ávido* no funcionará [2].

En todo caso, el problema con las *reglas ávidas* es que debemos demostrar que funcionan para toda combinación posible de entradas, y esa demostración no siempre es simple de hacer. Suponiendo que el conjunto *coins* de valores de monedas *es canónico*, entonces el modelo *avid.py* incluido en el proyecto [F25] *Avidos-Dinamica-Backtraking* que acompaña a esta Ficha resuelve el problema en forma óptima aplicando la regla ávida que hemos analizado. Se muestra a continuación:

```
__author__ = 'Cátedra de AED'

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tSe pidió mayor a', lim, '... cargue de nuevo...')
    return n

def insertion_sort(v):
    n = len(v)
    for j in range(1, n):
        y = v[j]
        k = j - 1
        while k >= 0 and y > v[k]:
            v[k+1] = v[k]
            k -= 1
        v[k+1] = y

def greedy_change(x, coins):
    # chequear si existe una moneda única que sea la solución...
    for value in coins:
        if value == x:
            return 1

    # si no existe, ordenar coins de MAYOR a MENOR...
    insertion_sort(coins)

    # ... y aplicar la regla ávida...
    count, amount = 0, 0
    for value in coins:
        while amount + value <= x:
            amount += value
            count += 1
        if amount == x:
            break
    return count

def change():
    print('Cambio de Monedas: Solución con Algoritmo Ávido')
```

```

coins = [1, 5, 10, 25]
print('Sistema monetario: - coins =', coins)
print('(Nota: el programa funcionará SÓLO si coins es canónico...)')
print()

print('Monto a cambiar...')
x = validar_mayor(0)

cant = greedy_change(x, coins)
print()
print('Cantidad mínima de monedas a devolver:', cant)

if __name__ == '__main__':
    change()

```

El programa anterior inicializa el arreglo *coins* con un conjunto que sabemos que es canónico. Luego carga por teclado el valor *x* a cambiar, validando que sea mayor a cero, e invoca a la función *greedy\_change()* que es la que aplica el algoritmo ávido.

La función *greedy\_change()* comienza chequeando con un *for iterador* [3] [4] el conjunto *coins*, para determinar si existe una moneda cuyo valor ya sea igual a *x*, en cuyo caso termina retornando un 1 (la solución óptima es devolver *x* con una sola moneda).

Pero si no existe una moneda con valor *x*, la función *greedy\_change()* procede a ordenar el arreglo *coins* de mayor a menor usando el algoritmo de *ordenamiento por inserción simple*. Este ordenamiento es necesario para el *algoritmo ávido* si *no* se supone que el arreglo *coins* vendrá ya ordenado. Y en este caso puntual, hemos usado un ordenamiento simple debido a que suponemos que *coins* tendrá muy pocos elementos y un algoritmo simple será entonces aceptable en cuanto a tiempo de ejecución. Si se espera que *coins* tenga una *gran cantidad de elementos*, sabemos que entonces debería aplicarse un método compuesto (*quicksort*, *heapsort*, *shellsort*, etc.) [5]

Luego de ordenar el arreglo, se lanza otro ciclo *for iterador* que va tomando uno por uno los valores del arreglo *coins* (en la variable *value*). Como *coins* está ordenado de mayor a menos, los valores de *value* serán tomados uno a uno de forma que el primer valor será el mayor. Se usa un acumulador *amount* para ir sumando los valores seleccionados en *value*, y un contador *count* para llevar la cuenta de la cantidad de valores/monedas que se acumularon.

Dentro del *for iterador*, otro ciclo (un *while* en este caso) prueba a sumar el valor actual *value* de la moneda seleccionada tantas veces como sea posible, mientras no se supere el valor de *x*. Si se supera el valor de *x*, el ciclo interno *while* corta, y el *for iterador* toma el siguiente elemento de *coins*, asignándolo nuevamente en *value*, repitiendo este esquema hasta que finalmente *amount* llegue a valer *x*. En ese momento el *while* se detendrá, y también lo hará el *for iterador* con el *break* ubicado en la condición debajo del *while* [3] [4]. La función retorna el valor final de *count*, que la cantidad de monedas a usar para cambiar *x*.

Ahora bien: si el número y designación de las monedas disponibles en *coins* es *arbitrario*, entonces podría *no ser canónico* y la solución anterior basada en un *algoritmo ávido* *no funcionará*. Nos preguntamos cómo plantear entonces un algoritmo que *siempre* funcione.

Ya hemos indicado que lo primero será fijar alguna precondición que *garantice* que la solución existe y sabemos que eso depende de que siempre exista la moneda de 1 centavo: cualquiera sea la cantidad a expresar, siempre se podrá plantear como suma de monedas de

1 centavo. Pero si el conjunto *coins* no es canónico (incluso garantizando que contenga el valor 1) el algoritmo ávido no necesariamente dará la solución óptima.

Una solución mejor consiste en usar una *tabla* (un arreglo) en la cual se almacenen los resultados para valores menores a  $x$ , recalculados, y luego se use esa tabla para calcular nuevos casos. Al fin y al cabo, si queremos saber cuántas monedas hacen falta para cambiar  $x = 63$  centavos, es posible que sirva saber cuántas monedas hacen falta para cambiar 30 o 60, y si esos cálculos ya fueron hechos con anterioridad sería útil poder reusar sus resultados sin tener que volver a calcular [2].

La idea: ir progresando en el cálculo del cambio óptimo para cada uno de los valores menores a  $x$ , (sin calcular dos o más veces para el mismo valor), y cada vez que se tenga un nuevo caso resuelto, se guarde su resultado en la tabla. Esta técnica o estrategia de solución basada en tablas de resultados previos, se conoce como *programación dinámica*.

En nuestro caso, dado el valor  $x$  a cambiar en monedas, queremos reutilizar los cálculos que hayamos realizado para cualquier valor menor que  $x$  durante el proceso, y para ello usaremos un arreglo *prev* que tendrá exactamente  $x + 1$  casilleros. De esta forma, el arreglo *prev* tendrá efectivamente su última casilla numerada con el valor  $x$ , y la idea es depositar en esa casilla el valor final: la *cantidad mínima de monedas* para cubrir el valor  $x$ .

Luego debemos ir llenando esa tabla *prev* con los resultados parciales de los subproblemas o valores *anteriores* al cálculo para  $x$ , para lo cual tendremos que plantear el mecanismo de llenado para cada casilla: este mecanismo se conoce como el planteo de la *recurrencia* a emplear para el llenado de la tabla. En todo proceso de *programación dinámica* se parte de uno o más *casos obvios*, llamados *casos base* de la recurrencia: en nuestro caso, el caso base es directo: si tenemos que cubrir un valor de  $x = 0$ , entonces necesitamos 0 monedas y sabemos entonces que la casilla *prev[0]* deberá valer 0.

A partir de allí, en cada casilla *prev[v]* (con  $1 \leq v \leq x$ ) deberemos almacenar la *mínima cantidad de monedas posible para llegar al valor v*, usando los valores de las monedas disponibles. El cálculo podría parecer complicado, pero un ejemplo mostrará en forma clara la ecuación de recurrencia a emplear:

Suponga (igual que en la solución ávida) que los  $n$  valores nominales de las monedas disponibles están almacenados en el arreglo *coins*. Sea *coins* = [1, 5, 10, 25, 50] con  $n = 5$  y sea  $v = 13$ : esto es, queremos llenar la casilla *prev[13]* de la tabla, suponiendo que ya hemos llenado **todas** las anteriores. Está claro entonces que los posibles valores para *prev[13]* saldrán del **mínimo** entre todas las siguientes combinaciones:

- ✓ 13 monedas de 1 centavo.
- ✓ 1 moneda de 1 centavo + el mínimo para cubrir 12 centavos = 1 + *prev[12]*
- ✓ 1 moneda de 5 centavos + el mínimo para cubrir 8 centavos = 1 + *prev[8]*
- ✓ 1 moneda de 10 centavos + el mínimo para cubrir 3 centavos = 1 + *prev[3]*
- ✓ Y no podemos usar monedas de 25 o 50 centavos, ya que ambas son mayores a  $v = 13$ .

De allí surge con sencillez la expresión completa de la recurrencia a usar:

$$\begin{aligned} \text{prev}[v] &= \text{mínimo}(v, \text{mínimo}(1 + \text{prev}[ui])) \\ (\text{donde } ui &= v - \text{coins}[j] \text{ con } j = 0, 1, 2, \dots n - 1) \end{aligned}$$

Finalmente, el planteo del algoritmo completo sería entonces:

- Entradas:

- Un arreglo *coins*, con los *n* valores nominales de las monedas disponibles (de forma que el valor nominal 1 exista).
- Un valor entero *x* > 0 que será el valor a cubrir con las monedas disponibles.

- Salidas:

- Un número entero que indique la mínima cantidad de monedas a devolver para cubrir el valor *amount*.

- Proceso (algoritmo básico – recurrencia a emplear):

```
dinamic_change(x, coins):
    1. Sea n = tamaño del arreglo coins (cantidad de valores de monedas)
    2. Sea la tabla prev con x + 1 casilleros, inicializados en 0
    3. Para todo v en [1..x] (incluyendo 1 y x):
        3.1. prev[v] = mínimo(v, mínimo(1 + prev[ui]))
              (donde ui = v - prev[j] (con j = 0, 1, 2, ..., n-1))
    4. Retornar prev[x]
```

Un replanteo del problema del cambio de monedas, ahora usando *programación dinámica*, se muestra en el mismo proyecto que acompaña a esta ficha, pero ahora en el siguiente modelo llamado *dinamico.py*:

```
__author__ = 'Cátedra de AED'

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
    if n <= lim:
        print('\t\tSe pidió mayor a', lim, '... cargue de nuevo...')
    return n

def dinamic_change(x, coins):
    # cantidad de valores de monedas disponibles...
    n = len(coins)

    # tabla para los resultados previos del cambio de los valores menores a x...
    # ... incluyendo una casilla prev[x] para el cambio del propio valor x...
    # ... y prev[0] = 0 por el caso base: hacen falta 0 monedas para x = 0...
    prev = [0] * (x + 1)

    # calcular el mínimo para cada posible valor de v con 1 <= v <= x
    for v in range(1, x + 1):
        # empezamos suponiendo que el mínimo es el propio v...
        minv = v

        # ... y verificamos si hay alguna combinación menor...
        for j in range(n):
            # si coins[j] se pasa, seguir el ciclo...
            if coins[j] > v:
                continue

            ui = v - coins[j]
            if prev[ui] + 1 < minv:
                minv = prev[ui] + 1

        # ...almacenar el mínimo encontrado para v en prev[v]...
        prev[v] = minv

    # retornar el mínimo para x, que está en la última casilla...
    return prev[x]
```

```

def change():
    print('Cambio de Monedas: Solución con Programación Dinámica')

    coins = [1, 5, 10, 21, 25]
    print('Sistema monetario: - coins =', coins)
    print('(Nota: vale cualquiera que tenga al 1, aún no canónico o desordenado)')
    print()

    print('Monto a cambiar...')
    x = validar_mayor(0)

    cant = dinamic_change(x, coins)
    print()
    print('Cantidad mínima de monedas a devolver:', cant)

if __name__ == '__main__':
    change()

```

La función *dinamic\_change()* aplica en forma directa el algoritmo basado en la recurrencia explicada más arriba. Se deja para el alumno el estudio de los detalles.

### 3.] Conceptos básicos sobre *Backtracking*.

Otra de las estrategias clásicas de resolución de problemas (o planteo de algoritmos) se conoce como *backtracking* (o *vuelta atrás*). Se trata de una técnica de base recursiva mediante la cual se exploran todos los posibles caminos que pudieran conducir a la solución de un problema, pero de forma tal que cuando se descubre que un camino no conduce a una solución válida, se aprovecha el retorno (o *vuelta atrás*, o *backtracking*) de las instancias recursivas que se hayan activado para "deshacer" los cambios que hayan llevado por ese camino incorrecto, e intentar la exploración de otro camino con nuevos valores [6] [2].

La técnica de *backtracking* es muy útil en problemas en los que las soluciones surgen de una o más combinaciones de los mismos datos de entrada, ya que el proceso de *prueba y error* que esencialmente permite realizar garantiza que el algoritmo no dejará afuera ninguna posible solución sin explorar. Sin embargo, el uso de la recursión para explorar todos los posibles caminos puede llevar a un número considerable o demasiado elevado de invocaciones o instancias recursivas, aumentando notablemente el tiempo de ejecución y/o la cantidad de memoria de stack requerida.

Para evitar estos inconvenientes, los programadores normalmente estudian cada problema en particular, tratando de identificar posibles formas de evitar trabajo extra y ahorrando así algunas tandas de invocaciones recursivas. En algunos casos, las técnicas empleadas para evitar esa sobrecarga de trabajo tienen nombres y características propias (como el proceso de *poda alfa-beta*), y en otros casos se trata directamente de variantes y trucos sutiles que dependen de cada problema en particular. Los detalles de estas técnicas escapan al alcance introductorio de estas notas, pero aún así el esquema general y básico de aplicación de la estrategia de *backtracking* puede analizarse sin mayores problemas.

Como hemos indicado en la introducción de esta misma Ficha de estudios, la técnica de *backtracking* suele usarse con frecuencia (y con éxito) en problemas de optimización combinatoria (encontrar la mejor forma de combinar ciertos elementos), y muchos de esos problemas tienen que ver con juegos de ingenio. Uno muy conocido, y muy usado para exponer los principios de aplicación del *backtracking*, es el *Problema de las Ocho Reinas*, que enunciamos a continuación [6]:

**Problema 63.) Problema de las Ocho Reinas:** Es un problema de ingenio derivado del juego del ajedrez, que como se sabe, consta de un tablero de  $8 * 8 = 64$  casilleros en el cual se ubican piezas de diversas características (rey, reina, alfil, caballo, torre y peón) y con diversas capacidades de movimiento. Entre esas piezas las más poderosa del juego es la **reina**, ya que puede moverse (y atacar) a lo largo de toda la fila, toda la columna y las dos diagonales que pasen por el casillero de su posición en el tablero. El Problema de las Ocho Reinas<sup>1</sup> se enuncia entonces de forma simple: en un tablero de ajedrez normal, tratar de ubicar ocho reinas pero de tal forma que **ninguna de ellas ataque a ninguna de las otras**.

**Discusión y solución:** Este problema de ingenio fue planteado por primera vez en 1848 por el ajedrecista *Max Bezzel*, y fue estudiado por diversos matemáticos (entre ellos: *Gauss*, *Cantor*, *Nauck* y *Glaisher*). El uso y aplicación de la computadora y las técnicas recursivas han aportado numerosos planteos algorítmicos que permiten arribar a una o a todas las soluciones posibles en forma relativamente sencilla<sup>2</sup>.

Un rápido análisis inicial muestra con claridad que cualquier solución del problema necesariamente tendrá que ubicar a cada una de las ocho reinas en ocho columnas diferentes y también en ocho filas diferentes del tablero (ya que de otro modo, estarían en posiciones de ataque mutuo).

Así, entonces, cada una de las reinas puede ser representada por un número *col* entre 0 y 7 que indique en qué *columna* del tablero estará ubicada, y la posición *fil* (o número de fila) de esa reina en la columna *col* puede almacenarse en un arreglo unidimensional de ocho componentes de tipo *int* que llamaremos *rc* (por *row on column = fila en la columna*). Cada casillero *rc[col]* se usará para guardar el número de fila dentro de la columna *col* en que estará ubicada la reina cuyo número es justamente *col*. Por ejemplo, si *rc* fuese:

```
rc = [2, 4, 7, 0, 5, 1, 3, 6]
```

entonces se estaría indicando que queremos a la primera reina (la reina de la columna 0) en la fila 2 del tablero; luego queremos a la segunda reina (la reina de la columna 1) en la fila 4 del tablero, y así sucesivamente.

Esta forma de notación es compacta y permite controlar rápidamente que las reinas no se ataquen entre ellas en ninguna fila y ninguna columna: está claro que no debería haber dos o más casilleros del arreglo *rc* con el mismo número, ya que eso implicaría que dos o más reinas en dos o más columnas distintas estarían ubicadas en la misma fila (como en el ejemplo siguiente, en el cual la tercera reina (columna 2) y la sexta (columna 5) están ubicadas en la misma fila (la 7)):

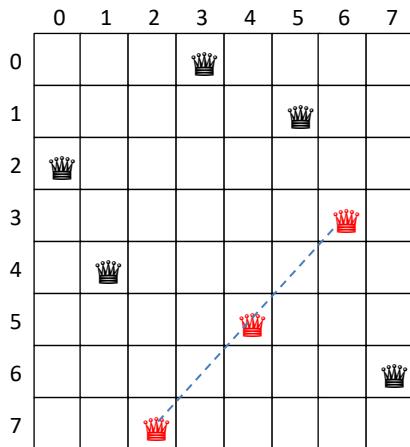
```
rc = [2, 4, 7, 0, 5, 7, 3, 6]
```

<sup>1</sup> El tema de las Ocho Reinas nos lleva inexorablemente a la excelente película argentina *Nueve Reinas* (del año 2000), dirigida por *Fabián Bielinsky* y protagonizada por *Ricardo Darín* y *Gastón Pauls*. En ella se muestra la historia de dos estafadores que en un momento dado se quedan con una plancha de nueve sellos postales, muy valiosa, conocida como las "Nueve Reinas". Claro... la plancha que ellos tenían era una falsificación y los dos bandidos intentan a toda costa venderla por un alto precio a un millonario español. Uno de los estafadores busca por todos los medios sacar al otro del negocio, para quedarse con toda la ganancia... pero sobre el final se produce un giro tan inesperado como deseado por todos los espectadores...

<sup>2</sup> De todos los posibles enfoques algorítmicos que pueden sugerirse, nos hemos basado especialmente en el esquema indicado por *Niklaus Wirth* en su libro "*Algoritmos + Estructuras de Datos = Programas*", capítulo 3, sección 3.5.

Queda todavía el tema no menor del control de las diagonales. La notación *row on column* que expresamos en el arreglo *rc* permite con sencillez controlar las filas y las columnas, pero nada controla respecto de posibles ataques en diagonal. Si hacemos un gráfico más detallado de las posiciones indicadas por el primer arreglo *rc* = [2, 4, 7, 0, 5, 1, 3, 6], vemos que el tablero representado para las ocho reinas sería el siguiente:

**Figura 1: Una ubicación incorrecta para las Ocho Reinas.**



Podemos ver de inmediato que la disposición prevista en el arreglo *rc* no es correcta, ya que las reinas (dibujadas en *rojo*) en las columnas 2, 4 y 6 comparten una de las diagonales del tablero, y por lo tanto se están atacando. El punto es que esperaríamos poder hacer el control en las diagonales en forma simple, emulando lo que se hizo con el control de las filas y las columnas. Si el tablero se representa como una matriz cuadrada ese control no será sencillo, ya que tendremos que realizar una gran cantidad de cálculos y validaciones para saber si una reina comparte o no una diagonal con otra.

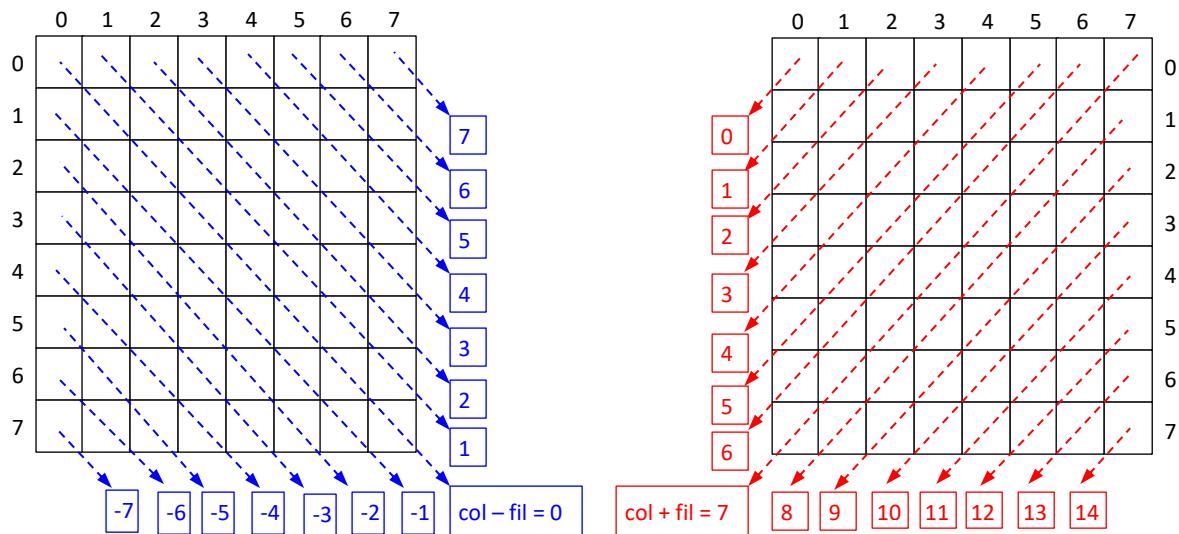
Para intentar simplificar, notemos que en general hay dos tipos de diagonales en una matriz (y esto es especialmente visible si la matriz es cuadrada): Por un lado, las diagonales que se orientan en la misma forma que la *diagonal principal* (y las llamaremos *diagonales normales*) y por el otro, las que se orientan en sentido inverso al de la diagonal principal (las designaremos como *diagonales inversas*). Y una propiedad interesante es que en cada una de las *diagonales normales*, la *diferencia* entre el número de columna y el número de fila de sus elementos es constante, mientras que en las *diagonales inversas* es constante la *suma* entre el número de fila y el de columna de cada elemento. La *Figura 2* (página 564) ilustra lo dicho (los números en *azul* o *rojo* al final de cada diagonal *normal* o *inversa* indican el valor de la *resta* o la *suma* de los índices de columna y fila de cada casillero en esas diagonales).

Como se puede ver en esa gráfica, una matriz cuadrada de 8 \* 8 casilleros tiene 15 diagonales de cada tipo, y cada una de ellas puede asociarse a un *número único*: la resta de los índices de los casilleros en las diagonales normales (15 números diferentes entre -7 y 7) y la suma de esos índices en las diagonales inversas (15 números diferentes entre 0 y 14).

Con estos elementos a la vista, en el planteo de la solución para el *Problema de las Ocho Reinas* incorporaremos otros tres arreglos unidimensionales (además del arreglo *rc*) para controlar lo que ocurre en el tablero. Estos nuevos arreglos almacenarán valores booleanos, y serán los siguientes [6]:

- ***qr*** (de *queens on row*) con 8 casilleros, de forma que  $qr[fil] = \text{True}$  indicará que la fila  $fil$  está libre de reinas (o sea, no hay reinas en la fila indicada por  $fil$ ).
- ***qid*** (de *queens on inverse diagonal*) con 15 casilleros, de forma que  $qid[di] = \text{True}$  indicará que la diagonal inversa  $di$  (con  $0 \leq di \leq 14$ ) está libre de reinas (o sea, no hay reinas en la diagonal inversa indicada por  $di$ ).
- ***qnd*** (de *queens on normal diagonal*) con 15 casilleros, de forma que  $qnd[dn] = \text{True}$  indicará que la diagonal normal  $dn$  (con  $0 \leq dn \leq 14$ ) está libre de reinas (o sea, no hay reinas en la diagonal normal indicada por  $dn$ ).

Figura 2: Diagonales **normales** e **inversas** en una matriz de  $8 * 8$ .



Es posible que el estudiante haya notado un pequeño detalle en las descripciones anteriores: según la gráfica de la *Figura 2*, las *diagonales normales* están asociadas a números entre -7 y 7, pero ahora estamos indicando que los 15 componentes del arreglo *qnd* serán usados para marcar si cada diagonal normal está libre o no de reinas. Pero en Python un número negativo usado como índice de un arreglo estaría accediendo a un casillero tomando como referencia que el índice -1 identifica al último componente, el -2 al anteúltimo y así en forma consecutiva. Por lo tanto, si se quiere marcar un casillero del vector *qnd* como libre de reinas, pero ese casillero está asociado a una diagonal con identificador negativo, entonces el programa *podría* estar accediendo a un casillero incorrecto si el programador no hizo sus cálculos en forma detallada.

Para evitar el problema (y evitar también posibles confusiones al mezclar índices positivos con índices negativos), hemos procedido de la siguiente forma *en cuanto a las diagonales normales*: si se quiere saber en qué diagonal normal está el casillero del tablero con índices  $[fil][col]$ , se calcula la resta  $col - fil$ , pero a ese resultado se le suma 7. Con esto, como las restas están en el rango  $[-7, 7]$ , entonces los valores obtenidos al sumar 7 estarán en el rango  $[0, 14]$ . Así, en el arreglo *qnd* el casillero 0 se usará para representar a la diagonal normal con resta igual a -7, el casillero 1 para la diagonal con resta igual a -6, y así sucesivamente. Formalmente, el casillero  $qnd[dn]$  (con  $0 \leq dn \leq 14$ ) representará a la diagonal normal  $dn = (col - fil) + 7$ .

Finalmente, podemos ahora analizar el programa para calcular una posición válida para cada reina en el tablero (ver el modelo *backtracking01.py* en el proyecto que acompaña a esta Ficha). La función de entrada o de arranque del programa es la siguiente:

```
def main():
    global rc, qr, qid, qnd

    # rc[col] = índice de fila de la reina que está en la columna col...
    # ... todas en -1 pues aun no sabemos en qué fila quedará cada reina...
    rc = [-1] * 8

    # qr[fil] = true si la fila fil está libre de reinas...
    # ... inicialmente, todas las filas están libres de reinas...
    qr = [True] * 8

    # qid[k] = true si la diagonal inversa k está libre de reinas...
    # ... 15 diagonales inversas... inicialmente todas libres de reinas...
    qid = [True] * 15

    # qnd[k] = true si la diagonal normal k está libre de reinas...
    # ... 15 diagonales normales... inicialmente todas libres de reinas...
    qnd = [True] * 15

    print('Problema de las Ocho Reinas: Una solución con Backtracking')
    print()

    success = intend(0)
    if success:
        print('Solución posible encontrada:', rc)
    else:
        print('No hay solución posible para esa posición de partida...')
```

La función *main()* comienza definiendo los cuatro arreglos *rc*, *qr*, *qid* y *qnd* como variables globales, para hacer que sea más simple compartirlas con la función *intend()* que es la luego aplicará el proceso de *backtracking*. Los cuatro arreglos son creados de forma de contener tantos casilleros como serán requeridos (8 casilleros con valor inicial -1 para *rc*, otros 8 con valor inicial *True* para *qr*, y 15 con valor inicial *True* para *qid* y *qnd*).

El valor -1 en cada casillero de *rc* indica que por el momento no se sabe en qué fila del tablero será ubicada cada una de las reinas representadas por las columnas de ese arreglo. El valor *True* en cada casillero de los arreglos *qr*, *qid* y *qnd* indica que por ahora todas las filas y todas las diagonales están libres de reinas (ya que de hecho, ninguna columna está ocupada...)

Luego, la función *main()* simplemente invoca a la función *intend(col)* enviando como parámetro el valor 0. El parámetro enviado cuando se invoca a *intend()* la primera vez (o sea, *col = 0*) es el índice de la primera columna del tablero (de hecho, el número de la primera reina). La función *intend()* comienza sus cálculos justamente tratando de determinar en qué fila de esa columna *col = 0* deberá ubicar la primera reina, y luego aplicará *backtracking* para calcular el resto de las ubicaciones.

Cuando la función *intend(col)* finalice su ejecución, el arreglo *rc* estará completo y cada casillero contendrá el número de la fila del tablero en que debe colocarse cada reina, representada por cada índice del vector *rc*, por lo cual *main()* sólo debe mostrar ese arreglo. La función *intend(col)* en rigor, retorna un valor booleano que indica si se ha encontrado o no una solución, y *main()* formalmente chequea ese valor retornado antes de mostrar el arreglo. Sin embargo, si la función es invocada enviando como parámetro el valor 0, el

resultado será efectivamente *True* (simplemente, sabemos que al menos una solución existe si se comienza trabajando desde la columna 0...) por lo que el control realizado por *main()* es redundante en este caso.

La función *intend(col)* es la siguiente:

```
def intend(col):
    global rc, qr, qid, qnd

    fil, res = -1, False
    while not res and fil != 7:
        res = False
        fil += 1
        di = col + fil
        dn = (col - fil) + 7
        if qr[fil] and qid[di] and qnd[dn]:
            rc[col] = fil
            qr[fil] = qid[di] = qnd[dn] = False
        if col < 7:
            res = intend(col + 1)
            if not res:
                qr[fil] = qid[di] = qnd[dn] = True
        else:
            res = True
    return res
```

La variable *fil* se usa para contener el índice de cada una de las filas del tablero, que serán exploradas una por una hasta dar con una solución. La variable *res* se usa como bandera para controlar si en algún momento se ha dado con una solución. Antes de iniciar el proceso, la variable *res* se pone en *False*, y luego se lanza un ciclo *while* para controlar cada posible fila. El ciclo continúa mientras queden filas sin explorar y la variable *res* se mantenga en *False*.

En cada vuelta del ciclo, se trabaja con la columna *col* (initialmente, *col* = 0) y se incrementa *fil* en 1 para intentar ubicar la primera reina justamente en esa posición. En la primera vuelta del ciclo, la *posición propuesta* para la *reina 0* es entonces *col* = 0 y *fil* = 0, que en nuestra representación sería *rc[col]* = *fil*.

El paso siguiente es comprobar si esa posición propuesta es *válida* (o *segura*). Está claro que la posición *rc[col]* = *fil* será *válida* si la fila *fil* está libre de reinas (o sea, si *qr[fil]* == *True*), y además si las dos diagonales *di* y *dn* que pasan por esa casilla están a su vez libres de reinas. Los índices de control de las dos diagonales son, respectivamente *di* = *col* + *fil* = 0 + 0 = 0, y *dn* = *col* - *fil* + 7 = 0 + 0 + 7 = 7. Por lo tanto, sólo debemos mirar los arreglos *qid* y *qnd* en esas posiciones para saber si esas diagonales están libres. La condición de control completa sería entonces:

```
if qr[fil] and qid[di] and qnd[dn]:
```

Si esa condición es *True*, entonces la posición *rc[col]* = *fil* es segura, y se puede intentar dejar la reina allí. Por lo tanto, si se entra en la rama verdadera, simplemente *se registra que ahora esa posición está ocupada (bloqueando la fila y las dos diagonales)*:

```
if qr[fil] and qid[di] and qnd[dn]:
    rc[col] = fil
    qr[fil] = qid[di] = qnd[dn] = False
```

Lo que sigue es el *paso recursivo*: habiendo registrado a la reina de la columna *col* = 0 en la posición *rc[col]* = *fil*, debemos ahora *intentar* ubicar a la reina siguiente (la reina de la

columna  $col + 1$ ) en el tablero. Esto sólo será posible (obviamente) si la reina que se acaba de ubicar en el tablero *no es todavía la de la columna 7* (si fuese la 7, sería ya la última y  $col + 1 = 8$  no existiría). Por ese motivo, se pregunta si  $col < 7$ , y en ese caso se activa recursivamente la propia función *intend()*, tomando como parámetro ahora el valor  $col + 1$ .

En la nueva instancia recursiva el proceso comienza nuevamente, pero ahora centrado en la columna  $col = 1$  (reina de la columna 1). Los arreglos qr, qid y qnd han modificado sus valores, por lo cual alguna fila, diagonal normal y/o diagonal inversa podrían estar bloqueadas. Esto hace que cada nueva reina que se va incorporando (con cada nueva instancia recursiva) tenga cada vez menos libertad de acción para su posición propuesta, y podría ocurrir que para la reina actual la posición  $rc[col] = fil$  no sea válida. Si tampoco fuese válida ninguna otra fila en esa columna, la instancia actual de la función *intend()* terminaría retornando *False*. Pero ese retorno de *False* sería tomado por la instancia recursiva anterior de *intend()*. La condición inmediatamente debajo de la llamada recursiva a *intend()*

```
res = intend(col + 1)
if not res:
    qr[fil] = qid[di] = qnd[dn] = True
```

controla justamente si la invocación recursiva para ubicar a la *siguiente* reina tuvo éxito o no. Y **si no lo tuvo**, entra en acción el mecanismo de corrección: la posición de la reina actual era segura, pero la reina siguiente no tuvo lugar para ella, por lo cual el camino seguido no conduce a una solución: se debe retroceder, eliminar el registro seguro de la reina actual (es decir, informar que la fila y las dos diagonales no están bloqueadas por esa reina), y probar una nueva ubicación en la siguiente vuelta del ciclo.

Este *mecanismo de corrección* es justamente el proceso de *backtracking* que hemos venido a buscar. La recursión permite ir hacia adelante explorando posibles posiciones de cada reina, y mientras todo vaya bien, no hay motivo para cambiar las posiciones ya registradas. Pero si en algún momento se llega a un callejón sin salida (la reina actual no puede ubicarse en ningún sitio), la instancia recursiva que no logró ubicar a esa reina termina, y vuelve atrás a la instancia anterior retornando *False*, lo cual es tomado como un aviso de *solución incorrecta*. La única forma de cambiar eso, consiste en quitar de su posición a la reina anterior y probar todo de nuevo. Y eso es justamente lo que el *backtracking* permite hacer: ensayar un camino, dejarlo registrado, y si algo falla, retroceder, borrar el registro anterior, y ensayar un nuevo camino.

Los detalles finos del mecanismo de trabajo de esta función se dejan para ser estudiados por el alumno. El programa completo se muestra a continuación:

```
__author__ = 'Cátedra de AED'

def intend(col):
    global rc, qr, qid, qnd

    fil, res = -1, False
    while not res and fil != 7:
        res = False
        fil += 1
        di = col + fil
        dn = (col - fil) + 7
        if qr[fil] and qid[di] and qnd[dn]:
            rc[col] = fil
            qr[fil] = qid[di] = qnd[dn] = False
```

```

    if col < 7:
        res = intend(col + 1)
        if not res:
            qr[fil] = qid[di] = qnd[dn] = True
    else:
        res = True
    return res

def main():
    global rc, qr, qid, qnd

    # rc[col] = índice de fila de la reina que está en la columna col...
    # ... todas en -1 pues aun no sabemos en qué fila quedará cada reina...
    rc = [-1] * 8

    # qr[fil] = true si la fila fil está libre de reinas...
    # ... inicialmente, todas las filas están libres de reinas...
    qr = [True] * 8

    # qid[k] = true si la diagonal inversa k está libre de reinas...
    # ... 15 diagonales inversas... inicialmente todas libres de reinas...
    qid = [True] * 15

    # qnd[k] = true si la diagonal normal k está libre de reinas...
    # ... 15 diagonales normales... inicialmente todas libres de reinas...
    qnd = [True] * 15

    print('Problema de las Ocho Reinas: Una solución con Backtracking')
    print()

    success = intend(0)
    if success:
        print('Solución posible encontrada:', rc)
    else:
        print('No hay solución posible para esa posición de partida...')

if __name__ == '__main__':
    main()

```

Específicamente, la función *intend(0)* dejará el arreglo *rc* con estos valores:

Solución posible encontrada: [0, 4, 7, 5, 2, 6, 1, 3]

que corresponden a **una** solución (de muchas posibles), que gráficamente sería equivalente al tablero que sigue:

**Figura 3: El tablero equivalente a la solución calculada por *intend(0)*.**

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

#### 4.] Cálculo de **todas** las soluciones al Problema de las Ocho Reinas.

En el problema 58 hemos estudiado un algoritmo basado en *backtracking* para calcular **una** solución al *Problema de las Ocho Reinas*. Sin embargo, es intuitivamente simple concluir que existen muchas otras soluciones posibles. Un análisis sistemático permite ver que el total de soluciones posibles (para un tablero de  $8 * 8$  y exactamente 8 reinas) es 92, aunque un estudio aún más detallado muestra que las soluciones realmente distintas son en realidad 12, y todas las otras son variaciones simétricas (rotadas de alguna forma) con relación a alguna de las 12 básicas.

Si se quiere obtener un listado de las 92 soluciones posibles (básicas o simétricas...), se puede hacer una modificación simple del programa que ya hemos mostrado para el problema 54: en lugar de un ciclo *while* que explore las filas una por una y se detenga con una bandera de corte cuando encuentre una solución, se puede usar un ciclo *for* que obligatoriamente recorra las 8 filas en cada una de las 8 columnas (comenzando igual que antes desde *col* = 0). Cada vez que se encuentre una solución, contarla y mostrarla en ese mismo momento (mostrar en ese momento el contador y el contenido actual del vector *rc*). Y como esa solución ya fue encontrada y mostrada, inmediatamente luego de mostrarla borrar su registro y proceder con otra vuelta de ciclo para buscar la siguiente.

El modelo *backtracking02.py* del proyecto que acompaña a esta Ficha, hace justamente eso. Al ejecutarlo, se mostrarán 92 líneas con otras tantas variantes del arreglo *rc*, cada una representando un tablero diferente para las *Ocho Reinas*. El programa es el siguiente (y de nuevo, se dejan los detalles para ser analizados por el estudiante):

```
__author__ = 'Cátedra de AED'

def intend(col):
    global rc, qr, qid, qnd, cs
    for fil in range(8):
        di = col + fil
        dn = (col - fil) + 7
        if qr[fil] and qid[di] and qnd[dn]:
            rc[col] = fil
            qr[fil] = qid[di] = qnd[dn] = False
            if col < 7:
                intend(col + 1)
            else:
                cs += 1
                print(rc, ' (', cs, ')', sep='')
                qr[fil] = qid[di] = qnd[dn] = True

def main():
    global rc, qr, qid, qnd, cs
    # rc[col] = índice de fila de la reina que está en la columna col...
    # ... todas en -1 pues aun no sabemos en qué fila quedará cada reina...
    rc = [-1] * 8

    # qr[fil] = true si la fila fil está libre de reinas...
    # ... inicialmente, todas las filas están libres de reinas...
    qr = [True] * 8

    # qid[k] = true si la diagonal inversa k está libre de reinas...
    # ... 15 diagonales inversas... inicialmente todas libres de reinas...
    qid = [True] * 15
```

```
# qnd[k] = true si la diagonal normal k está libre de reinas...
# ...15 diagonales normales... inicialmente todas libres de reinas...
qnd = [True] * 15

# contador de soluciones...
cs = 0
print('Problema de las Ocho Reinas: Todas las soluciones... ')
print()
print('Distintas soluciones encontradas... ')
intend(0)

if __name__ == '__main__':
    main()
```

---

## Bibliografía

---

- [1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.
- [2] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.
- [3] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: <https://docs.python.org/3/>.
- [4] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.
- [5] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.
- [6] N. Wirth, Algoritmos + Estructuras de Datos = Programas, Madrid: Ediciones del Castillo, 1989.