Ficha 9

Subproblemas y Funciones

1.] Introducción al concepto de *subproblema*.

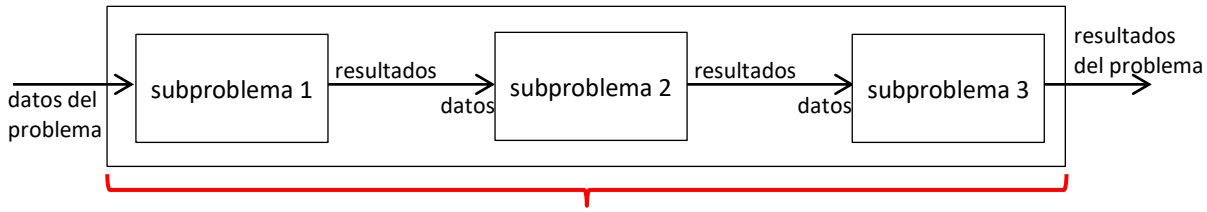
La mayoría de los problemas que normalmente se enfrentan en la práctica son de *estructura compuesta*, es decir, son problemas que pueden ser divididos en subproblemas cada vez menores en complejidad, hasta llegar a subproblemas que no necesiten nuevas subdivisiones (y que por eso se designan como *problemas 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*[1], 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*.



[1] 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.*

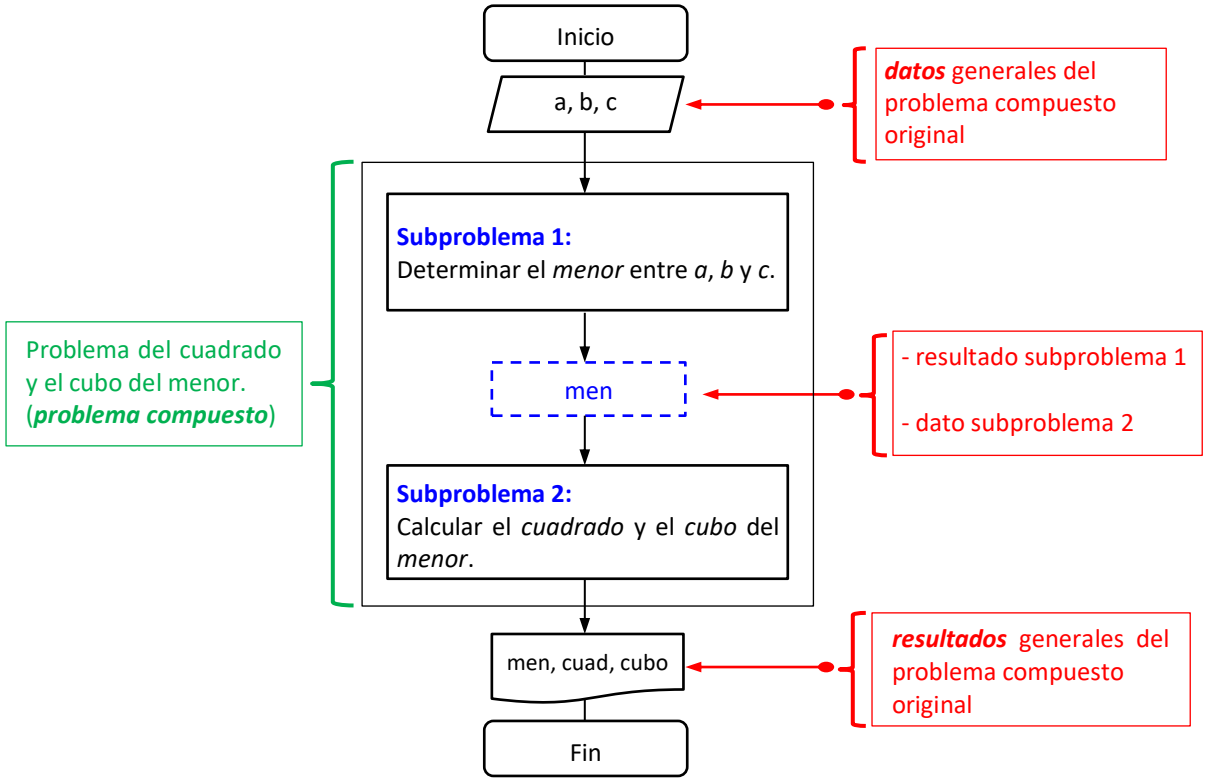
- **Resultados:** El cuadrado y el cubo del menor. (*cuad*, *cubo*: int)
- **Datos:** El menor de los valores de entrada. (*men*: int)
- **Procesos:** En este subproblema, todo el trabajo consiste en calcular el cuadrado y el cubo del menor:

```
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.

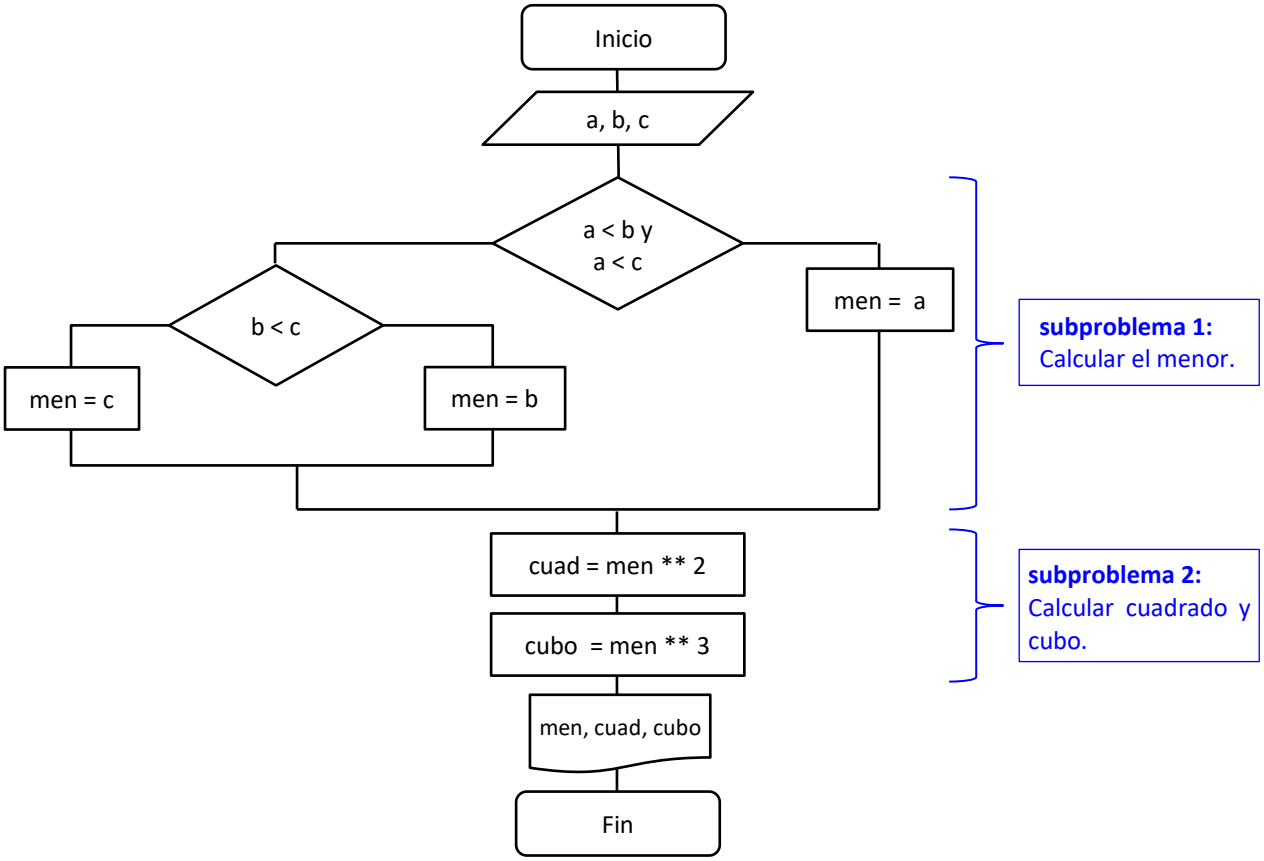
Es fácil ver que con estos dos subproblemas detectados y analizados, la estructura gráfica completa del problema original puede representarse a través del modelo que se ve en la *Figura 2* :

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)
```

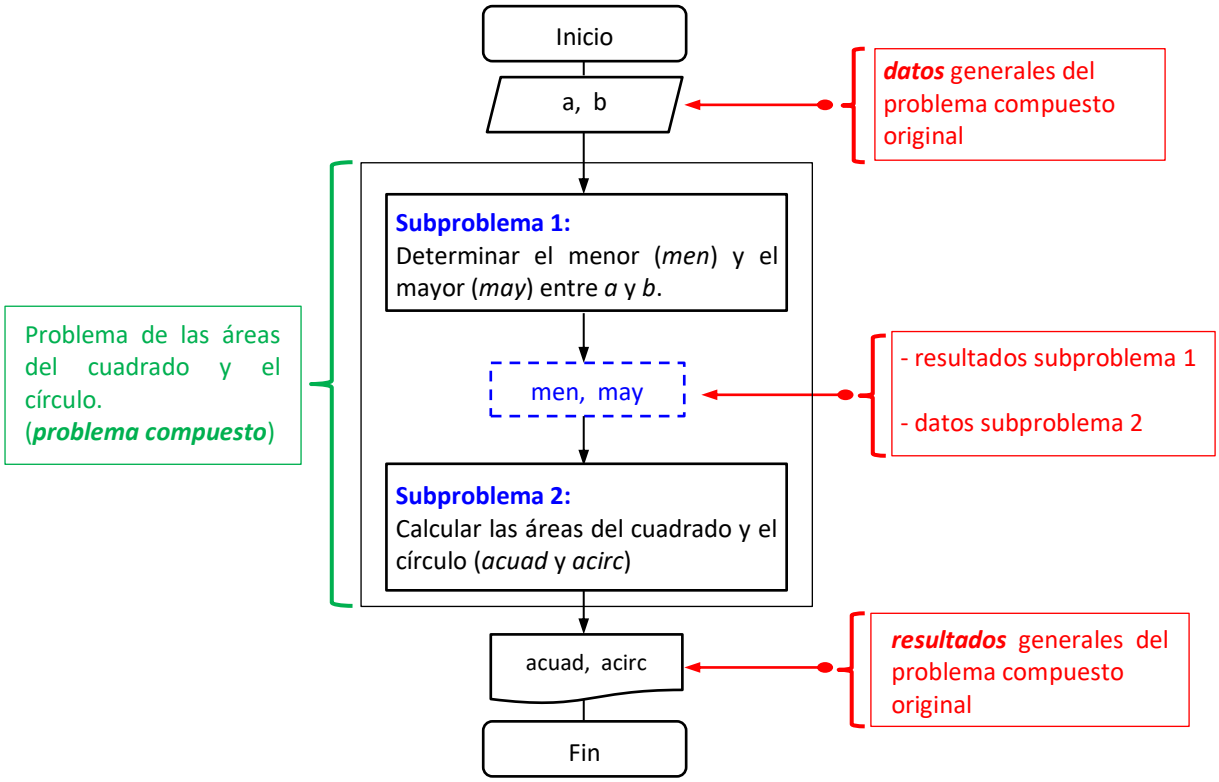
Como ya habrá podido notar, a medida que se estudian problemas de estructura más compleja y se incorporan nuevos elementos (como instrucciones condicionales, instrucciones repetitivas y subproblemas), se hace también cada vez más extenso el análisis del problema a nivel de identificación de datos, resultados 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 brevísima 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.



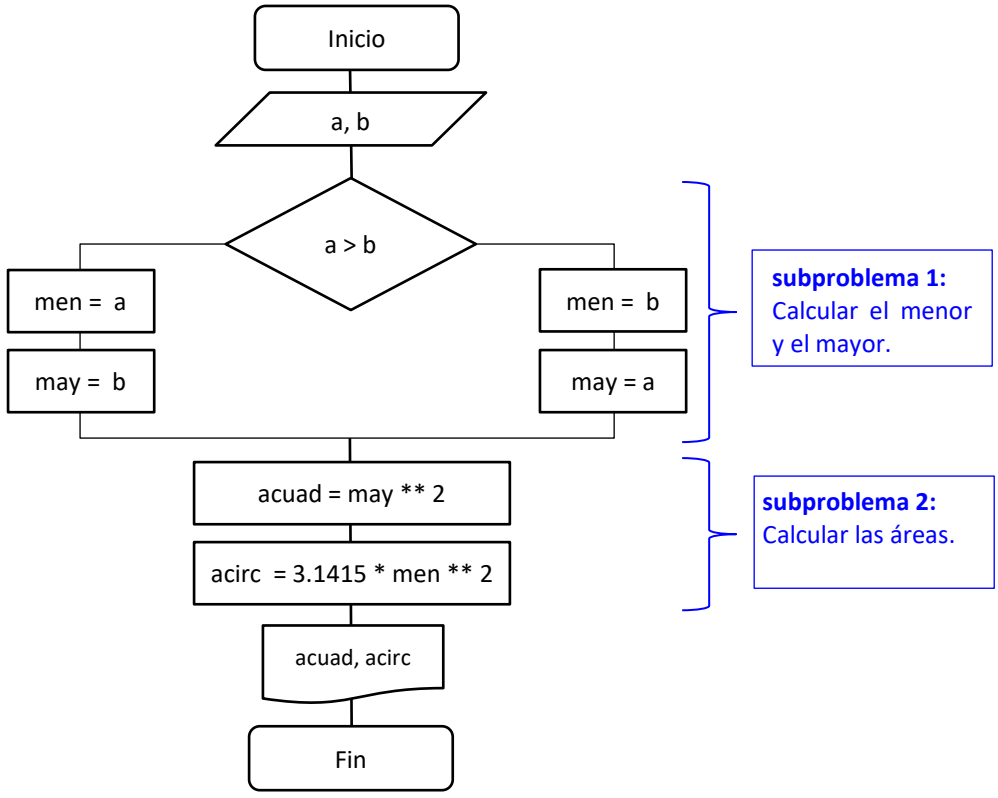
El *subproblema 1* tomará los valores de las variables *a* y *b* cargados por teclado, determinará el menor y el mayor y los copiará a su vez en las *variables temporales men* y *may*. De esta forma, si luego se requiere volver procesar estos valores en diferentes lugares del programa,

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 la áreas...
acuad = may ** 2
acirc = 3.1415 * men ** 2

# visualización de resultados...
print('Area del cuadrado:', acuad)
print('Area 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].

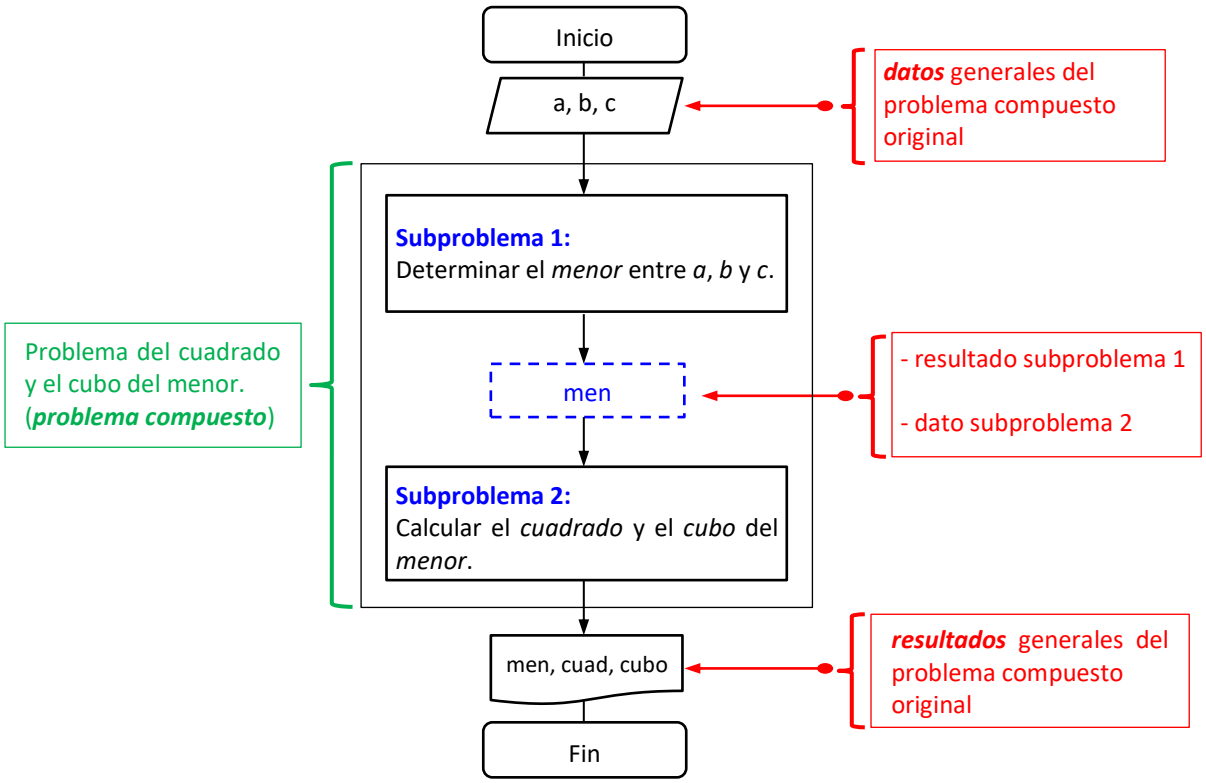
Esencialmente, una *subrutina* (o *subproceso*) es un segmento de un programa que se escribe en forma separada del programa principal. A cada subrutina el programador asocia un nombre o identificador y mediante ese identificador la subrutina puede ser activada todas las veces que sea necesario. Esto 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* (*página 186*),Figura 2: Modelo de subproblemas para el *problema del cuadrado y el cubo del menor*. 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.

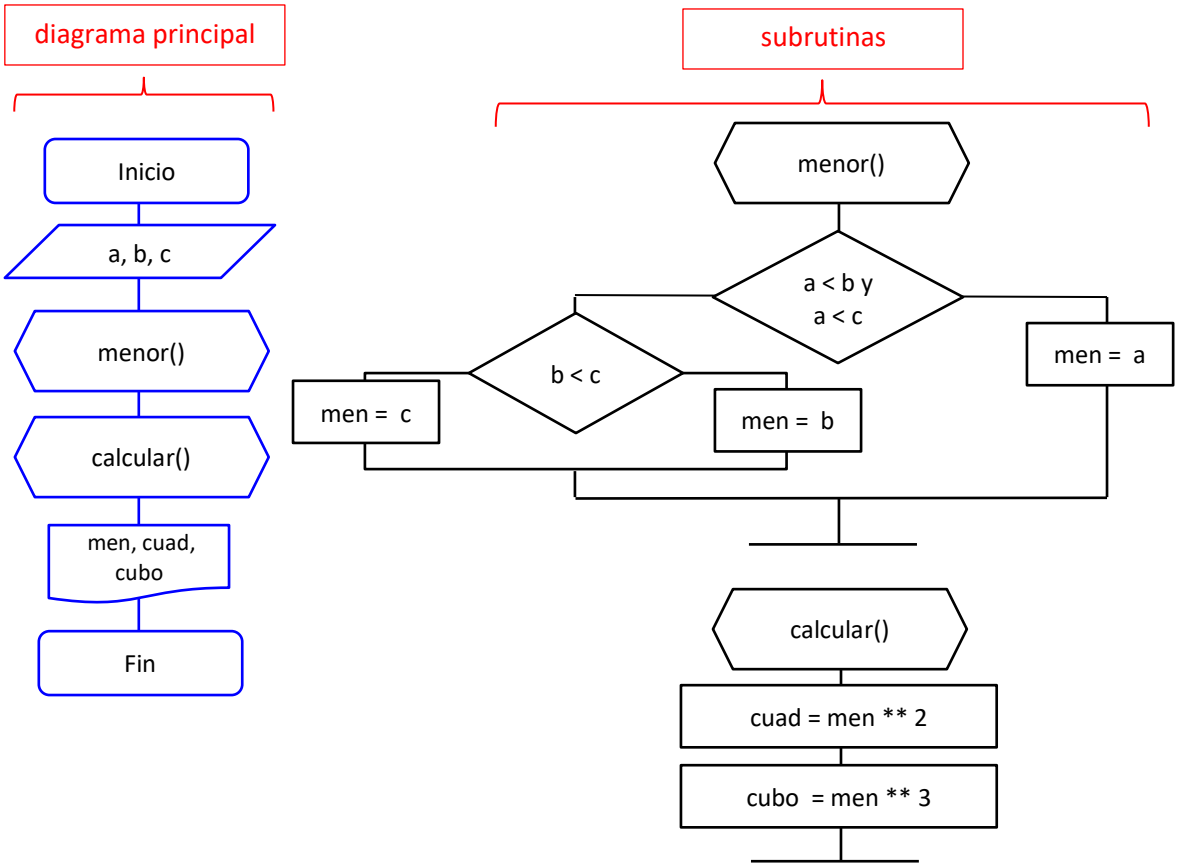
En un diagrama de flujo es especialmente fácil introducir nuevas reglas para denotar la presencia de una subrutina: simplemente se diagrama la misma por separado. El diagrama que se ve en la *Figura 7* (*página 192*) es el mismo que el que anteriormente mostramos en la *Figura 6* para el problema del 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*.



El diagrama de los procesos que realiza cada subrutina se desarrolla por separado, pero se comienza cada gráfico de subrutina con *el mismo símbolo hexagonal que se usó para dejarla indicada en el diagrama principal*. Los procesos internos de cada subrutina se grafican de la misma forma y con los mismos símbolos usados en cualquier diagrama de flujo. Y para indicar que el gráfico de la subrutina ha terminado, simplemente se coloca un trazo horizontal al final de su gráfica.

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*.

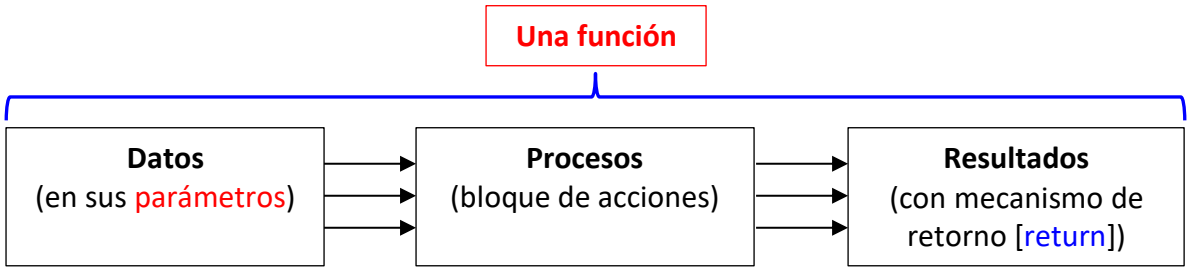
El resto de la función es el ***bloque de acciones*** de la misma. Como ya se dijo, el bloque se escribe a renglón seguido de la cabecera, e indentado hacia la derecha del comienzo de la misma. En ese bloque se escriben las instrucciones que la función debe ejecutar cuando sea invocada o activada desde 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*. Esa 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)[2] que acepta *entradas o datos* (los *parámetros*), procesa esos

[2] 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 196*) 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

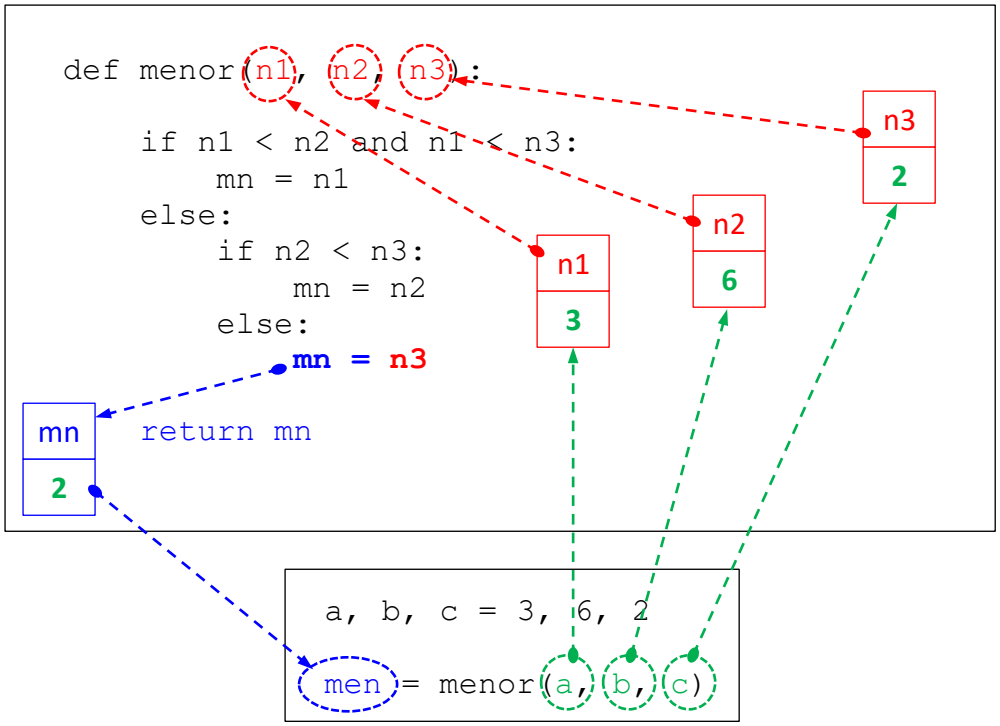
buscar el menor entre otras variables o valores constantes, como se ve en las invocaciones que siguen:

```
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
`y = pow(x, n)`	El valor devuelto se asigna en una variable *y*.
`print(pow(x, n))`	El valor devuelto se muestra directamente en consola.
`y = 2 * pow(x, n)`	El valor devuelto se multiplica por dos y el resultado se asigna en *y*.
`y = pow(pow(x, n), 3)`	Calcula x^n, eleva el resultado al cubo, y asigna el resultado final en *y*.

- *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 subprocesos, no hay inconveniente (aunque tampoco obligación) en plantear cada subproceso 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 retornen 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 retornado 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 siguiente 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 siguiente 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 >= 90%*
- *Nivel Medio: 75% <= Porcentaje < 90%*
- *Nivel Regular: 50% <= Porcentaje < 75%*
- *Fuera de Nivel: 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('Numero:', 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
Numero: 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:

```
__author__ = 'Cátedra de AED'


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('Area del cuadrado:', acuad)
print('Area 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 asignen 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.

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) 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, √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 *√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 √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: $n >= 10^{25}$) 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 *√n*). Por ejemplo, si se quiere comprobar la *primalidad* de un número *n* de 35 dígitos ($n >= 10^{35}$) 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 su factores primos (o factorización de ese número)*. La *factorización* de un número entero positivo *n* es el conjunto de números *{p1, p2, ... pk}* 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 sólo 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
        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 caracter que el programador quiera usar para indicar el final de la línea mostrada por *print()*. Por defecto, el valor de *end* es un caracter "\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*[3], 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* (sección de *Temas Avanzados*). 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 – Temas Avanzados*:

[3] 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.

```
def total_time(expr):
    t1 = time.clock()
    r = eval(expr)
    t2 = time.clock()
    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)

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:

```
__author__ = 'Catedra de AED'


import time


def total_time(expr):
    t1 = time.clock()
    r = eval(expr)
    t2 = time.clock()
    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 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=' ')
            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()
```

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli*, para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, y particularmente los ingenieros *Karina Ligorria*, *Cynthia Corso* y *Gustavo Federico Bett*, que realizaron aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] M. Pilgrim, "Dive Into Python - Python from novice to pro," 2004. [Online]. Available: http://www.diveintopython.net/toc/index.html.

[3] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: https://docs.python.org/3/.

[4] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.