Ficha 12

Programación Modular

1.] Tratamiento especial de parámetros en Python.

En una ficha anterior hemos analizado la forma utilizar parámetros en una función en Python, y hemos visto luego que la parametrización es uno de los mecanismos que favorecen el desarrollo de funciones que puedan volver a emplearse en distintos lugares de un programa, o incluso de otros programas. Pero los conceptos y mecanismos que vimos no agotan el tema: una función en Python puede plantearse de forma de aceptar y manejar un *número variable de parámetros*, en forma similar a lo que permiten hacer otros lenguajes como C y C++, o a lo que ya hace Python con algunas de sus funciones predefinidas como *max()* y *min():*

```
m1 = max(3, 5, 6)
m2 = max(2, 5, 6, 7, 2, 1)

m3 = min(5, 6, 3, 8)
m4 = min(4, 9, 2, 8, 1, 7, 0)
```

Estas dos funciones de Python pueden ser invocadas enviándoles diferentes cantidades de parámetros actuales cada vez, y ambas procesarán esos parámetros y retornarán el valor mayor o el valor menor, respectivamente.

Puede verse que de forma, una función resulta aun más general, ya que puede ser aplicada en muchos contextos que requieran ese servicio, sin importar la cantidad de parámetros que deban pasarse a modo de datos. No es necesario contar con funciones diferentes que hagan la misma tarea sobre un número distinto de parámetros: *la misma y única función puede aceptar la cantidad de parámetros que se requiera procesar, y los procesa*.

Obviamente, el programador puede desarrollar sus propias funciones con número variable de parámetros. Existen básicamente *tres mecanismos* para lograr esto en Python, y esos tres mecanismos pueden a su vez combinarse. Veamos brevemente cada uno de ellos [1]:

a.) ***Parámetros con valores por defecto***: Es común y muy útil asignar a los *parámetros formales* de una función un *valor por defecto*, de forma de poder invocar a la función enviando menos parámetros actuales. Se dice que un *parámetro formal tiene un valor por defecto*, cuando el mismo es asignado en forma explícita con un valor en el momento en que se define en la cabecera de la función. Veamos un ejemplo:

Supongamos que se quiere definir una función que tome como parámetros a dos números *n1* y *n2*, y se desea que la función retorne una tupla con esos mismos números, pero ordenados. Típicamente, el ordenamiento esperado es de menor a mayor pero en ocasiones podría requerirse que sea de mayor a menor.

En lugar de definir dos funciones (una para cada tipo de ordenamiento) se puede desarrollar sólo una, que tome tres parámetros: los dos números a ordenar y un flag o bandera de tipo booleano para indicar si queremos un ordenamiento ascendente o descendente, como se ve en el siguiente modelo: En la función *ordenar()* los parámetros formales *n1* y *n2* son los números que se deben comparar. Estos dos parámetros no tienen valor por defecto, ***por lo que al invocar a la función deben obligatoriamente ser enviados los parámetros actuales correspondientes.*** El tercer parámetro (*ascendent*) es un valor booleano que indica si la función debe retornar los valores en orden ascendente (*ascendent = True*) o descendente (*ascendent = False*) *El detalle es que este tercer parámetro tiene por defecto el valor True*, por lo cual si el programador desea resultados en orden ascendente, *puede invocar a la función enviando sólo los dos números a comparar y la función asumirá que el valor del tercer parámetro es True*:

```
def ordenar(n1, n2, ascendent=True):
    # se asume ascendent = True...
    first, second  = n2, n1
    if n1 < n2:
        first, second = n1, n2

    # ... pero si ascendent = False, invertir los valores...
    if not ascendent:
        first, second = second, first

    return first, second


def test():
    a = int(input('Ingrese el primer valor:  '))
    b = int(input('Ingrese el segundo valor: '))

    # orden ascendente…
    men, may = ordenar(a, b)

    print('Menor:', men)
    print('Mayor:', may)

    c = int(input('Ingrese el primer valor: '))
    d = int(input('Ingrese el segundo valor: '))

    # orden descendente…
    may, men = ordenar(c, d, False)

    print('Menor:', men)
    print('Mayor:', may)

# script principal...
test()
```

Como se ve, *si se quiere que el ordenamiento sea descendente, la función ordenar() debe invocarse con tres parámetros actuales*, como el caso de *ordenar(c, d, False)*. El tercer parámetro actual puede obviarse al invocar a la función (como en *ordenar(a, b)*) y su valor se asumirá *True*, pero también puede enviarse si el programador así lo desea: una invocación de la forma *ordenar(a, b)* es equivalente a otra de la forma *ordenar(a, b, True)*: en ambas, el tercer parámetro formal se tomará como igual a *True*, y la función ordenará de menor a mayor.

Los parámetros formales que no tienen valor por defecto en la función, se designan como *parámetros posicionales*; y note lo que ya hemos indicado: si una función tiene *parámetros posicionales*, entonces al invocar a la función ***es obligatorio*** enviar los parámetros actuales que correspondan a esos posicionales, *pues de lo contrario se lanzará un error de intérprete*. Además, al declarar una función, los *parámetros posicionales* deben definirse siempre ***antes*** que los parámetros con valor default. La siguiente cabecera para la función *ordenar()* es incorrecta, ya que el parámetro formal posicional *n2* se está definiendo después de un parámetro con valor default:

```
# incorrecto...
def ordenar(n1, ascendent=True, n2):
```

En nuestro ejemplo original, la función *ordenar()* original tiene *dos parámetros formales posicionales* (*n1* y *n2*) y un tercero con *valor por defecto* (*ascendent*). Por lo tanto, al invocarla es siempre obligatorio enviarle al menos dos valores actuales para los posicionales *n1* y *n2*:

```
may, men = ordenar(a, b, False)    # correcto…
men, may = ordenar(c, d)           # correcto…
men, may = ordenar(c, d, True)     # correcto…

may, men = ordenar()               # incorrecto…
men, may = ordenar(a)              # incorrecto…

men, may = ordenar(a, False)        # atención aquí...
```

Los tres ejemplos *resaltados en color verde en el modelo anterior son correctos*. *Los dos resaltados en rojo producirán un error de intérprete*: en el primero de ellos, los dos parámetros posicionales *n1* y *n2* se quedan sin valor actual, y en el segundo, el primer parámetro *n1* se asigna con el valor actual de *a*, pero el segundo se queda sin asignar.

El *ejemplo resaltado en color violeta* no produce un error de intérprete, ya que efectivamente se han enviado dos parámetros actuales al invocar a la función… Sin embargo, hay un problema: los parámetros actuales que se envían son asignados a los parámetros formales posicionales en orden de aparición, de acuerdo a su posición en la cabecera de la función (de allí el nombre de *posicionales*) por lo que el valor de *a* será asignado en *n1* y el valor *False* será asignado en *n2*. En la función, el parámetro *ascendent* tomará entonces su valor por defecto (*True*), y la función procesará datos incorrectos (comparará un *int* con un *boolean* y asumirá un ordenamiento ascendente cuando el programador lo quería descendente)…

b.) ***Parámetros con palabra clave***: Si una función tiene parámetros asignados con valores por defecto, podría haber problemas de ambigüedad para invocarla correctamente, ya que los valores de los parámetros actuales se toman en orden de aparición para ser asignados en los formales [1] [2]. Considere el siguiente ejemplo simple:

```
def datos(nombre, pais='Argentina', sexo='Varon', trabaja=True, estado='Soltero'):
    print('Datos recibidos: ')
    print('Nombre:', nombre)
    print('Pais:', pais)
    print('Sexo:', sexo)
    print('Tiene trabajo?:', trabaja)
    print('Estado civil:', estado)
```

```
def test():

    # error (interprete): falta el parametro 1 (obligatorio)...
    # datos()

    # Ok...
    datos('Pedro')

    # Ok...
    datos('Luisa', 'Uruguay', 'Mujer')

    # error (ambiguedad): toma pais = "Mujer", sexo = "Casada"...
    datos('Maria', 'Mujer', 'Casada')

test()
```

La función *datos()* del modelo, tiene cinco parámetros: el primero (*nombre*) es posicional y los otros cuatro (*pais*, *sexo*, *trabaja*, *estado*) tienen valores por defecto (o por *default*). En forma correcta, el parámetro posicional *nombre* está definido *antes* que los cuatro parámetros con valor default. Como la función tiene al menos un parámetro posicional, entonces obligatoriamente debe enviarse al menos un parámetro actual al invocarla. *La siguiente invocación produciría un error de intérprete* (y por eso está *comentarizada* en el ejemplo anterior):

```
# error (interprete): falta el parametro 1 (obligatorio)...
datos()
```

La siguiente invocación es *correcta*: el parámetro posicional *nombre* queda asignado con el valor 'Pedro', y los otros cuatro parámetros toman sus valores default:

```
# Ok...
datos('Pedro')
```

La salida en pantalla producida al invocar a la función en estas condiciones, sería la siguiente:

```
Datos recibidos:
Nombre: Pedro
Pais: Argentina
Sexo: Varon
Tiene trabajo?: True
Estado civil: Soltero
```

Como los cuatro últimos parámetros formales tienen valores default, podemos ignorar algunos de ellos al invocar a la función. La siguiente invocación también es correcta:

```
# Ok...
datos('Luisa', 'Uruguay', 'Mujer')
```

El parámetro *nombre* se asignará con la cadena 'Luisa', y los dos parámetros que siguen (*pais* y *sexo*) serán asignados con los valores 'Uruguay' y 'Mujer' respectivamente. Los dos últimos parámetros (*trabaja* y *estado*) quedarán con sus valores *default*. La salida sería la siguiente:

```
Datos recibidos:
Nombre: Luisa
Pais: Uruguay
Sexo: Mujer
Tiene trabajo?: True
Estado civil: Soltero
```

Pero finalmente, si se ejecuta el ejemplo tal como está, *la última invocación a la función datos() asignará valores incorrectamente en los parámetros formales*: si queremos dejar el parámetro *pais* en su valor *default*, *no podemos saltearlo* y luego seguir enviando valores explícitos porque eso provocaría ambigüedad. La salida producida por la última invocación sería la siguiente:

```
Datos recibidos:
Nombre:  Maria
Pais:  Mujer
Sexo:  Casada
Tiene trabajo?:  True
Estado civil:  Soltero
```

Conclusión: los parámetros con valores default deben declararse después que los parámetros posicionales, *y además no pueden saltearse en forma directa cuando la función es invocada*. Todos los parámetros actuales que se envíen a la función, serán tomados y asignados a los parámetros formales *en estricto orden de aparición* de izquierda a derecha, pudiendo provocar ambigüedades si se intenta saltear un parámetro.

Para evitar este tipo de problemas y poder *seleccionar* qué parámetros se deben dejar con su valor *default* y qué parámetros se desea asignar en forma explícita, en Python *se puede seleccionar cada parámetro por su nombre (usando el mecanismo de* ***selección de parámetro por palabra clave****) cuando se invoca a la función*:

```
def datos(nombre, pais='Argentina', sexo='Varon', trabaja=True, estado='Soltero'):
    print('Datos recibidos: ')
    print('Nombre:', nombre)
    print('Pais:', pais)
    print('Sexo:', sexo)
    print('Tiene trabajo?:', trabaja)
    print('Estado civil:', estado)


def test():

    # ok...
    datos('Luigi', pais='Italia')

    # ok... el parámetro "nombre" tambien puede accederse asi…
    datos(nombre='Luigi', pais='Italia')

    # ok... uno sin palabra clave, otro con palabra clave, y el resto default...
    datos('Camila', 'Argentina', sexo='Mujer')

    # ok... el orden de palabras clave no importa...
    datos('Bruno', sexo='Varon', pais='Italia')

    # error: luego de una palabra clave, no puede seguir explícito...
    # datos('Mary', pais='Inglaterra', 'Mujer')

    # error: no se puede asignar dos veces el mismo parametro...
    # datos('Federico', pais='Argentina', pais='Italia')

    # error: no puede usar un parametro que no existe...
    # datos('Conrado', colegio='Lasalle')


test()
```

Como puede verse, la idea es en principio simple: al ***invocar*** a la función, se escribe el nombre del parámetro formal cuyo valor se quiere asignar en forma explícita y se asigna a ese parámetro el valor que se requiere. Sin embargo, hay algunas reglas que deben respetarse, y que se deducen del modelo anterior:

- Cualquier parámetro puede seleccionarse usando la notación de palabra clave, *incluso los parámetros posicionales* (que no tienen un valor default asociado):

```
# ok... el parámetro "nombre" tambien puede accederse asi…
datos(nombre='Luigi', pais='Italia')
```

- Una vez que se accedió a un parámetro por palabra clave, *los que siguen a él en la lista de parámetros formales* ***deben*** ser accedidos por palabra clave cuando se invoca a la función:

```
# error: luego de una palabra clave, no puede seguir explícito...
# datos('Mary', pais='Inglaterra', 'Mujer')
```

- No se puede asignar *más de un valor al mismo parámetro*:

```
# error: no se puede asignar dos veces el mismo parametro...
# datos('Federico', pais='Argentina', pais='Italia')
```

- No se puede usar un *parámetro que no existe*:

```
# error: no puede usar un parametro que no existe...
# datos('Conrado', colegio='Lasalle')
```

- Mientras se respeten las reglas anteriores, *no hay problema en cambiar el orden de acceso a los parámetros usando sus palabras clave*:

```
# ok... el orden no importa... si se respeta todo lo demás...
datos('Bruno', sexo='Varon', pais='Italia')
```

Un detalle interesante, es que obviamente Python usa masivamente tanto el mecanismo de parámetros con valores default como este mecanismo de selección de parámetros por palabra clave[1], en sus funciones predefinidas de la librería estándar. Una de las funciones en las que se puede ver claramente este hecho, es la conocidísima función *print()* que usamos para visualizar resultados y mensajes en la consola estándar [1].

Esta función (entre otros) dispone de dos parámetros formales con valores default que son típicamente seleccionados por palabra clave: *sep* y *end*. El primero se usa para indicar a la función qué caracter o cadena de caracteres debe usar para separar las cadenas que se quieren mostrar, y *su valor default es un espacio en blanco (' ')*. El segundo se usa para indicar con qué caracter o cadena de caracteres debe terminar la visualización, y *su valor default es un salto de línea ('\n')*. Por ese motivo, en la siguiente secuencia:

```
x, y = 10, 20
print('Valor x:', x)
```

[1] El uso de palabras clave para seleccionar un elemento o para restringir el acceso (o permitir el paso) a determinados lugares o aplicaciones es obviamente común en programación… pero también en muchos otros ámbitos reales o imaginarios. ¿Quién no recuerda la increíble trilogía de películas de *El Señor de los Anillos* (o *The Lord of the Rings*) dirigida por *Peter Jackson* y protagonizada por *Ian McKellen* y *Elijah Wood*, y concretamente la primera de la saga (*The Fellowship of the Ring* o *La Comunidad del Anillo* del año 2001) en la que *Gandalf*, *Frodo* y los demás deben abrir las *Puertas de Durin* de las *Minas de Moria*? Esas puertas sólo se abrían si se decía la frase o palabra correcta… que resultó ser la palabra "*mellon*" ("*amigo*" en el idioma de los elfos…) Las películas de Jackson están basadas en la aún más extraordinaria obra literaria en tres volúmenes de *J. R. R. Tolkien*: *The Lord of the Rings* de 1954.

```
print('Valor y:', y )
```

la salida producida es de la forma:

```
Valor x: 10
Valor y: 20
```

Como el parámetro *sep* tiene un espacio en blanco como valor default, la función pone un espacio en blanco luego de las cadenas 'Valor x:' y 'Valor y:', haciendo que los números 10 y 20 aparezcan a un blanco de distancia de los dos puntos en cada caso. Y como el valor default del parámetro *end* es un salto de línea, la función muestra su salida y salta al renglón siguiente, provocando que ambas salidas aparezcan a en dos renglones separados.

Se pueden cambiar esos caracteres llamando a la función y seleccionando esos parámetros por su palabra clave:

```
x, y = 10, 20
print('Valor x:', x, sep='--> ', end='\n\n')
print('Valor y:', y)
```

La salida producida por el script anterior, será de la forma:

```
Valor x:--> 10

Valor y: 20
```

Como puede verse, en la primera invocación a la función el parámetro *sep* fue asignado con la cadena '--> ', lo que hace que la ejecutarse la función se agregue esa cadena después del título 'Valor x:'. Y como el parámetro *end* fue asignado con *dos saltos de línea* en lugar de uno ('\n\n'), entonces ambas líneas de salida aparecen a dos renglones de distancia.

En el ejemplo siguiente, el valor de *end* se reemplaza por un *espacio en blanco*, lo que hace que ambas invocaciones a la función muestren sus salidas en la misma línea de la pantalla:

```
x, y = 10, 20
print('Valor x:', x, end=' ')
print('Valor y:', y)
```

La salida producida es:

```
Valor x: 10 Valor y: 20
```

Note que si la función *print() es invocada sin ningún parámetro* su efecto será simplemente mostrar el valor de *end*, provocando sólo un salto de línea:

```
x, y = 10, 20
print('Valor x:', x)
print()
print('Valor y:', y)
```

La salida producida es de la forma:

```
Valor x: 10

Valor y: 20
```

ya que la primera invocación provoca un salto de línea (*end* tiene en ella su valor default '\n') y la segunda invocación a *print() sin parámetros* provoca un segundo salto.

c.) Listas de parámetros de longitud arbitraria: La tercer forma de tratamiento especial de parámetros en Python, consiste en permitir que una función acepte un *número arbitrario* de parámetros. Los parámetros enviados de acuerdo a esta modalidad entrarán a la función empaquetados en una *tupla*: esto es, entrarán a la función como una lista de valores separados por comas y accesibles por sus índices [1] [2].

La forma de indicar que una función tiene una lista variable de parámetros, consiste (en general) en *colocar un asterisco delante del nombre del último **parámetro posicional** que se prevea para la función*. Ese último parámetro, definido de esta forma, representa la secuencia o tupla de valores de longitud variable.

En el ejemplo siguiente, se muestra una función *procesar_notas()* que toma dos parámetros posicionales y obligatorios: el *nombre* y la *nota* final de un alumno en un curso. Pero además la función define una *lista de parámetros de longitud variable mediante el tercer parámetro args*. Se supone que los valores adicionales que la función recibirá, son las notas parciales que el alumno haya obtenido (si se decide enviarlas). La función simplemente muestra todos los datos en consola estándar. Observe una posible forma de procesar la secuencia de valores en la tupla *args*, usando un *for* que itera sobre esa tupla, y la forma de chequear si efectivamente hay parámetros adicionales preguntando si la longitud de la tupla es diferente de cero mediante la función *len()* de la librería estándar:

```
def procesar_notas(nombre, nota, *args):
    # procesamiento de los parámetros normales...
    print('Notas del alumno:', nombre)
    print('Nota Final:', nota)

    # procesar la lista adicional de parámetros, si los hay...
    if len(args) != 0 :
        print('Otras notas ingresadas:')
        for d in args:
            print('\tNota Parcial:', d)


def test():
    # una invocacion "normal", sin parámetros adicionales...
    procesar_notas('Carlos', 9)
    print()

    # una invocacion con tres notas adicionales...
    procesar_notas('Juan', 8, 10, 6, 7)


# script principal...…
test()
```

La salida producida por el programa anterior, sería la siguiente:

```
Notas del alumno: Carlos
Nota Final: 9

Notas del alumno: Juan
Nota Final: 8
Otras notas ingresadas:
    Nota Parcial: 10
    Nota Parcial: 6
    Nota Parcial: 7
```

También es interesante notar que Python aplica esta técnica en numerosas funciones de su librería estándar. Conocemos un par de esas funciones: *min()* y *max()*, que determinan y retornan el menor/mayor de una secuencia de valores de longitud arbitraria:

```
mn = min(2, 4, 5, 7, 2, 3)
my = max(2, 6, 3, 7)
```

En ambos casos, los parámetros son ingresados a la función como tuplas en la forma descripta en esta sección, y ambas funciones procesan luego esas tuplas.

2.] Programación Modular: Introducción.

Existen básicamente dos motivos por los cuales se utilizan funciones (subrutinas) en un programa en cualquier lenguaje de programación [3]:

- ✓ El primer motivo es lograr *ahorro de líneas de código*, desarrollando como función a todo bloque de instrucciones que en un mismo programa se use en muchas ocasiones. Así, si en un programa se debe realizar varias veces el ordenamiento de tres o cuatro variables, pero cambiando cada vez las variables que se ordenan, se define *una sola función* (con los *parámetros* que correspondan), y *no* se escribe dos o tres veces el código para el ordenamiento. Este es el motivo por el cual originalmente se definió el concepto de función en un lenguaje de programación.
- ✓ El segundo motivo es el de permitir que un programador pueda *modularizar adecuadamente sus programas*, dividiéndolos en subproblemas que le resulten más fáciles de manejar y controlar. En este sentido, el uso de funciones no se limita sólo a evitar posibles redundancias de código, sino que también apunta a la mejor estructuración de un programa. De hecho (y como vimos), se designa como *programación estructurada* al paradigma de programación que, entre otros principios, aplica la *programación modular*, que consiste en dividir un problema en subproblemas, y estos a su vez en otros, de forma que finalmente se plantee un programa en base a funciones que resuelvan los subproblemas.

Notemos, a modo de comentario, que de acuerdo con esto último un lenguaje de programación será tanto más apto para la programación estructurada mientras más fácilmente permita que un programador la lleve a cabo. En ese sentido, puede decirse que ***todo*** lenguaje facilita la programación estructurada, pues todo lenguaje permite definir funciones. Sin embargo, algunos lenguajes van más allá: en el lenguaje *Python* o en el lenguaje *Pascal*, por ejemplo, se pueden definir funciones y *también se puede definir una función dentro de otra*. Esto último no es válido en la mayoría de los lenguajes: en *C / C++* una función puede llamar a otra si es necesario (como en todos los lenguajes), *pero no puede contener la declaración de otra dentro de sí*. Los lenguajes *Python* y *Pascal*, entonces, son más aptos para aplicarse en situaciones de problemas que se dividen en otros, y estos a su vez en otros, y por lo tanto favorecen mejor la programación estructurada que el lenguaje *C / C++*. Por supuesto, como se dijo, en *C / C++* (como en cualquier lenguaje, incluido *Python*) se pueden definir tantas funciones como se quiera, y luego hacer que estas se invoquen entre sí.

Hemos indicado que un *paradigma de programación* es un conjunto de reglas, convenciones y prácticas para desenvolverse con éxito en la tarea de la programación de computadoras. Hemos dicho también que uno de esos paradigmas es el que se conoce como *programación estructurada*, dentro del cual nos mantendremos a lo largo de este curso. La programación estructurada se basa en una serie de principios y buenas prácticas que apuntan primero a

que el programador pueda resolver un problema descomponiéndolo en partes más simples (que hemos designado como *subproblemas*) y segundo a que el programador pueda plantear programas conceptualmente claros, más simples de comprender y eventualmente también más simples de corregir o modificar.

Dentro de la *programación estructurada*, la estrategia de dividir un problema en subproblemas más simples, y luego escribir subrutinas en un lenguaje de programación que se correspondan con la solución a esos subproblemas, se conoce como *programación modular*. Este nombre se debe a que en general, las subrutinas o conjuntos de subrutinas que se plantean para resolver a los subproblemas detectados, suelen designarse también como *módulos de programación*.

En este punto, debe quedar claro que la *programación modular* es ***uno*** de los principios básicos o estrategias de la *programación estructurada*, pero no es el único: el paradigma estructurado dispone de otras convenciones de trabajo (que iremos descubriendo y analizando a medida que avancemos en el curso), tales como, por ejemplo, evitar los llamados *saltos incondicionales* dentro un bloque de instrucciones (que en muchos lenguajes pueden hacerse con órdenes de la forma "*go to*").

A modo de ejmplo, analizaremos más abajo un problema relativamente simple del campo de la gestión administrativa y plantearemos para ese problema una *solución modular*: intentaremos descubrir los subproblemas más relevantes, dar un algoritmo que permita resolverlos, y finalmente escribir un programa en Python que se base en el planteo modular propuesto. Si durante el desarrollo aparece algún elemento que requiera la aplicación de nuevos elementos del lenguaje y/o de alguna variante interesante de temas o elementos ya conocidos, entonces se hará una explicación puntual dentro del análisis del mismo problema. Esperamos con esto seguir contribuyendo a que los estudiantes dispongan de un pequeño "catálogo" de problemas resueltos que siempre es útil cuando se enfrentan nuevos problemas, ya que muchas veces la solución a un nuevo problema suele inspirarse en soluciones conocidas para problemas ya analizados.

El programa está planteado en forma simple, utilizando sólo estructuras condicionales (sin ciclos) debido a que el conjunto de datos a procesar es pequeño y de tamaño fijo, y además para poder centrar la discusión en la forma de dividir en subproblemas y ajustarlos al uso de funciones. Retomaremos además aquí, el uso del esquema de pseudocódigo para aclarar algunas explicaciones. El enunciado es el siguiente:

Problema 31.) *Una empresa de turismo que vende viajes para egresados de colegios secundarios, ofrece a tres cursos distintos la siguiente promoción: El costo del viaje por persona es de $ 1360, pero si el grupo excede de las 40 personas, la empresa realiza un descuento del 5% sobre el costo total del viaje para el curso. Desarrollar un programa, que cargando la cantidad de alumnos de cada uno de los tres cursos, permita determinar:*

1. *El curso más numeroso*
2. *El monto del viaje para cada curso*
3. *El porcentaje que representa el monto del viaje del curso más numeroso sobre el total de la ganancia de la empresa.*

Discusión y solución: En este problema la división en subproblemas resulta imprescindible: si no se hace un planteo bien modularizado y con subprocesos claramente separados, el

resultado podría ser un programa con su lógica muy intrincada, con muchos bloques de código repetidos (y por lo tanto, redundante) [3].

En cuanto a los datos, este problema sólo pide la carga de tres valores (que llamaremos *c1*, *c2* y *c3*) que representan la cantidad de alumnos en tres cursos de una escuela secundaria. El primero de los resultados pedidos es indicar cuál de esos tres cursos es el más numeroso, lo cual puede hacerse con una subrutina que compare los valores de *c1*, *c2* y *c3* y almacene un descriptor de tipo cadena de caracteres en una variable de salida (que podemos llamar *may_cur*). Si llamamos *mayor()* a la subrutina, un primer esbozo en pseudocódigo sería:

```
mayor(c1, c2, c3):
    si c1 > c2 y c2 > c3:
        may_cur = 'Primero'
    sino
        si c2 > c3:
              may_cur = 'Segundo'
        sino:
              may_cur = 'Tercero'

    return may_cur
```

El primer resultado pedido quedaría completamente cubierto con esta subrutina. Note que como ahora sabemos que la subrutina se llamará *mayor()*, entonces el pseudocódigo comienza con el nombre de la misma y no con la palabra *algoritmo* (como hicimos en fichas anteriores).

Cuando seguimos avanzando en la lectura de los resultados a cubrir, vemos que el tercero nos pide calcular el porcentaje que el *monto* del curso más numeroso representa en el *monto total* facturado. Aún no hemos calculados los montos para cada curso, pero por el momento podemos asumir que cuando lo hagamos los tendremos asignados en otras tres variables *m1*, *m2* y *m3*. El monto total será la suma de estos tres valores, y para el porcentaje pedido tenemos que saber cuál de los tres corresponde al curso con más alumnos. La subrutina *mayor()* fue la encargada de determinar el curso más numeroso, pero en el momento en que la planteamos no tuvimos en cuenta la necesidad de conocer los montos. Para no tener que *volver a preguntar cuál de los cursos es el más numeroso*, podemos volver atrás a la subrutina *mayor()*, asumir que para ese momento ya estarán calculados los montos, y replantearla para que no sólo se guarde un descriptor del mayor, sino también el monto de ese curso en una segunda variable de salida (que llamaremos *may*):

```
mayor(c1, m1, c2, m2, c3, m3):
    si c1 > c2 y c2 > c3:
          may_cur = 'Primero'
          may = m1
    sino:
          si c2 > c3:
                may_cur = 'Segundo'
                may = m2
          sino:
                may_cur = 'Tercero'
                may = m3

    return may_cur, may
```

Con esto, la subrutina *mayor()* resuelve el primer objetivo (determinar el curso mayor) y ayuda a resolver el tercero (el porcentaje del monto del mayor sobre el total). Este último

resultado quedará asignado en la variable *porc*, que puede calcularse como se ve en el pseudocódigo de la subrutina que llamaremos *porcentaje()* (note que antes de realizar el cálculo, se verifica el valor de *mtot* para evitar un error de división por cero):

```
porcentaje(m1, m2, m3, may):
    mtot = m1 + m2 + m3
    si mtot != 0:
      porc = may * 100 / mtot
    sino:
      porc = 0

    return porc
```

Con esto, tenemos resueltos el primero y el tercero de los requerimientos del problema. Sólo nos queda (por fin) analizar cómo hacer el cálculo del monto que debe pagar cada curso, e informar esos montos en consola de salida. En principio, cada alumno paga 1360 pesos, por lo cual el monto inicial (*m1*, *m2* y *m3*) de cada curso podría calcularse como se ve en nuestro primer planteo de la subrutina *montos()*:

```
montos(c1, c2, c3):
    m1 = c1 * 1360
    m2 = c2 * 1360
    m3 = c3 * 1360

    return m1, m2, m3
```

Sin embargo, el enunciado aclara que si el número de alumnos en un curso es mayor a 40, entonces la empresa hará un descuento del 5% sobre el total inicial a pagar. Por lo tanto, la subrutina *montos()* podría modificarse de la siguiente forma:

```
montos(c1, c2, c3):
    sea m1 = c1 * 1360
    sea m2 = c2 * 1360
    sea m3 = c3 * 1360

    si c1 > 40:
        m1 = m1 - m1*5/100
    si c2 > 40:
        m2 = m2 - m2*5/100
    si c1 > 40:
        m3 = m3 - m3*5/100

    retornar m1, m2, m3
```

Como sabemos, una instrucción de la forma

```
m1 = m1 - m1*5/100
```

actúa como un acumulador para el valor de la expresión *m1*5/100*, pero restando en lugar de sumar: El miembro derecho de la asignación se ejecuta primero, tomando los valores que en ese momento tengan las variables que aparecen. En este caso, la única es *m1* que al momento de ejecutar la instrucción contiene el *monto inicial* a pagar por el curso. En el miembro derecho se calcula el 5% de ese monto, y el mismo es restado del valor actual de *m1*, con lo cual el resultado es el monto inicial menos el 5% de ese mismo monto. Al asignar ese resultado en la propia variable *m1*, el valor de la misma se actualiza: deja de valer el monto inicial y pasa a valer el monto descontado. Lo mismo vale para las otras dos expresiones con *m2* y *m3*.

Con esto la lógica general del algoritmo para resolver el problema queda cerrada. El código fuente completo en Python se muestra a continuación, aplicando todas las ideas discutidas y recordando que el script principal sólo incluirá una invocación a una función de arranque general (que llamaremos *test()*):

```
def mayor(c1, m1, c2, m2, c3, m3):
    if c1 > c2 and c1 > c3:
        may = m1
        may_cur = 'Primero'
    else:
        if c2 > c3:
            may = m2
            may_cur = 'Segundo'
        else:
            may = m3
            may_cur = 'Tercero'

    return may_cur, may


def montos(c1, c2, c3):
    m1 = c1 * 1360
    m2 = c2 * 1360
    m3 = c3 * 1360

    if c1 > 40:
        m1 = m1 - m1/100*5

    if c2 > 40:
        m2 = m2 - m2/100*5

    if c3 > 40:
        m3 = m3 - m3/100*5

    return m1, m2, m3


def porcentaje(m1, m2, m3, may):
    mtot = m1 + m2 + m3
    if mtot != 0:
        porc = may / mtot * 100
    else:
        porc = 0

    return porc


def test():
    # título general y carga de datos...
    print('Cálculo de los montos de un viaje de estudios...')
    c1 = int(input('Ingrese la cantidad de alumnos del primer curso:  '))
    c2 = int(input('Ingrese la cantidad de alumnos del segundo curso:  '))
    c3 = int(input('Ingrese la cantidad de alumnos del tercer curso:  '))

    # procesos... invocar a las funciones en el orden correcto...
    m1, m2, m3 = montos(c1, c2, c3)
    may_cur, may = mayor(c1, m1, c2, m2, c3, m3)
    porc = porcentaje(m1, m2, m3, may)

    # visualización de resultados
```

```
    print('El curso mas numeroso es el', may_cur)
    print('El monto del viaje del primer curso es:', m1)
    print('El monto del viaje del segundo curso es:', m2)
    print('El monto del viaje del tercer curso es:', m3)
    print('El porcentaje del monto del mas numeroso en el total es:', porc)


# script principal: sólo invocar a test()...
test()
```

La idea de definir una función que contenga todo el montaje inicial completo del programa (como nuestra función *test()*) para después simplemente invocar a esa función como única acción dentro del script principal es la que sugerimos ahora y aplicaremos de aquí en más en todos nuestros modelos, ejemplos y desarrollos. Como se puede observar, esto permite que el programa quede completamente modularizado, sin bloques de código "sueltos" que no podrían ser invocados desde otras funciones si el programador tuviese necesidad de hacerlo.

En el desarrollo de la solución del problema anterior, no hemos desplegado diagramas de flujo pero sí pseudocódigos: recuerde que ambas son herramientas auxiliares que los programadores usan en función de sus necesidades, y en este ejercicio el análisis de la lógica directamente desde el código fuente era simple.

3.] Funciones generalizadas (o reutilizables).

En fichas anteriores se presentaron y se analizaron problemas de todo tipo (del campo de la matemática, la gestión administrativa, los juegos, la física, y otros) pero la idea de la *modularización* es siempre la misma: consiste en plantear un programa a través de *módulos* (en nuestro caso, *funciones*) que representen soluciones a cada uno de los subproblemas que se hayan detectado. A su vez, el uso de parámetros y retorno de valores permite que una función pueda ser interpretada y usada como un *proceso general*, independiente de elementos externos (como variables globales, por ejemplo): los datos que la función necesita son tomados desde sus parámetros formales, y los resultados que ella genera son devueltos al exterior mediante el mecanismo de retorno [3].

A medida que van ganando experiencia los programadores descubren que muchos procesos que fueron pensados originalmente como funciones específicas de un programa en particular, vuelven a ser útiles en otros programas para resolver otros problemas. Ese es el caso, por ejemplo, de la función para calcular el *factorial* de un número (presentada en la *Ficha 7*), que se vuelve a usar en diversos problemas estudiados en otras fichas, o de las funciones que hemos presentado en la *Ficha 9* para *tratamiento de números primos*.

Esta situación es extremadamente común (y además deseable…) y es una de las múltiples formas en que se produce lo se conoce como *reutilización de código* (o *reuso de código*): una misma pieza (o módulo) de software ya diseñada para un sistema, proyecto o programa, se vuelve a utilizar en forma directa y por simple mecanismo de invocación, sin cambios ni ajustes, en otro sistema, proyecto o programa. Note que en ese sentido, todas las funciones predefinidas que un lenguaje de programación provee en sus librerías estándar son reutilizables: el programador puede usarlas una y otra vez en cuanto programa lo requiera, sólo invocándolas (y eventualmente, agregando alguna instrucción adicional como *import*, en Python).

¿Cuáles son los elementos que hacen que una función sea *reutilizable*? En general la reutilización de una función depende de dos factores: por un lado, mientras menos dependiente de elementos *declarados en forma externa* sea una función, más reutilizable será. Y en forma similar, por otra parte, mientras menos dependa una función de *entradas y salidas realizadas a través del teclado y la pantalla*, más reutilizable será también. Y la forma esencial de lograr esa independencia, es recurrir al empleo de parámetros para pasar datos a la función, y al mecanismo de retorno para obtener los resultados que ella entregue.

En este contexto, una función que se ha planteado de forma de ser reutilizable se dice también una *función generalizada*, ya que puede aplicarse en forma general y sin importar las características específicas del programa que la utilice: sólo se requiere saber qué parámetros enviarle, y qué resultados devolverá. Todo lo demás, es local a la función y administrado por su bloque de acciones.

Tomemos por ejemplo nuestra ya conocida función para calcular el *factorial* de un número *n* y analicemos las siguientes variantes para ella:

Figura 1: Variantes para el planteo de la función *factorial()*.

variante a.)

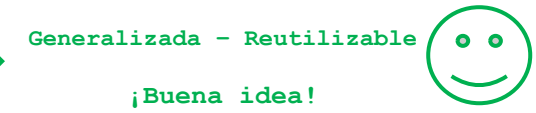
```
def factorial():
    global n, f
    f = 1
    for i in range(2, n+1):
        f *= i
```

variante b.)

```
def factorial():
    n = int(input('Ingrese n: '))
    f = 1
    for i in range(2, n+1):
        f *= i
    print('Su factorial es:', f)
```

variante c.)

```
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
```



En la figura anterior, la *variante a.)* de la función *factorial()* está planteada de forma de usar *variables globales*, lo cual marca que *depende de elementos que están definidos fuera de ella* (como la variable *n*) o que son accesibles desde fuera de ella (como la variable *f*). El problema con esta *dependencia externa*, es que para poder usar esta función en cualquier programa, el mismo debe tener definidas las dos variables *n* y *f* en algún lugar *externo a la función*, pues de otro modo esta no tendrá disponibles los datos que necesita (la variable *n* en este caso) y/o no tendrá forma de comunicar sus resultados (que en este caso dejó

almacenados en la variable *f*). Conclusión: la función así planteada *no es reutilizable* en forma directa (o tiene menos probabilidad de ser reusable). No puede simplemente ser importada a un programa cualquiera e invocarse sin hacer cambios y agregados en el código fuente del programa y/o la propia función.

La *variante b.)* que se muestra en la misma figura no usa variables globales ni depende de elemento alguno definido fuera de la función, por lo que podría parecer una buena solución. Sin embargo, el hecho es que esta segunda versión toma sus datos (el valor de *n*) desde el teclado de la consola estándar, y envía sus resultados (la variable *f*) directamente a la consola de salida estándar. El problema con esta *dependencia de la interfaz de usuario* es más sutil: para invocar a esta función, necesaria y forzosamente se debe aceptar que la misma cargue por teclado el valor de *n* y muestre el valor de *f* por pantalla. Pero si en el programa en el cual se desea reusar la función el valor de *n* estuviese ya definido y asignado como resultado de un cálculo previo, por ejemplo, entonces la carga por teclado exigida por nuestra nueva versión sería completamente inútil y nos obligaría a cambiar el código fuente, para eliminar esa carga… Y algo parecido ocurriría con la visualización del valor de *f* al final de la función. Además, existe otro inconveniente: la función está asumiendo que la interfaz de usuario estará basada en el uso del teclado y la pantalla de la consola estándar, pero si la idea fuese usar esa función en un programa pensado para una interfaz visual (una IGU con ventanas y objetos gráficos controlados por el mouse, por ejemplo), entonces la función no tendría cabida y debería ser completamente rediseñada. El *problema de la dependencia de una función o módulo de software respecto de la interfaz de usuario empleada*, también se conoce como *acoplamiento entre procesos e interfaz de usuario*. Conclusión: otra vez, la función así planteada no es reutilizable en forma directa y/o tiene menos posibilidades de ser reutilizada.

Finalmente, la versión que se muestra en la *variante c* de la *Figura 1* usa *parámetros* para captar sus datos desde el exterior (la *variable formal n* en este caso), con lo cual no importa cuáles sean las *variables actuales* que se envíen al invocarla ni dónde hayan sido definidas, ya que la función *tomará copias* de sus valores y los asignará en sus *parámetros formales* (que a los efectos, son variables locales de la función). En forma similar, la función devuelve con el *mecanismo de retorno* los resultados que haya generado y que tiene almacenados en variables locales propias de la función (la variable *f* en este caso). *No hay dependencia externa*, ya que todas las variables que usa la función son locales a ella, y si es necesario compartir valores con el exterior se hace mediante sus parámetros y sus retornos. Por otra parte, la función *tampoco presenta acoplamiento entre procesos e interfaz* de usuario: sus datos son tomados de sus *parámetros formales* (y no desde el teclado u otro dispositivo) y sus resultados salen al exterior mediante *return* (y no directamente a la pantalla). Conclusión: *la función está diseñada en forma genérica*, sin dependencias, y puede ser reutilizada en forma amplia, sin restricciones, en distintos programas y situaciones.

Obviamente, no siempre el programador necesitará diseñar funciones o módulos genéricos y reusables. En algún momento durante el desarrollo de un programa casi seguro deberá incluir una o mas funciones que interactúen con el usuario mediante la interfaz (en algún momento el usuario deberá introducir los datos y ver los resultados en pantalla…) pero la idea es tratar de delimitar esa responsabilidad a unas pocas funciones especialmente dispuestas para ello. Eso es lo que típicamente hemos hecho hasta ahora con nuestra clásica función *test()* (o como sea que la llame el programador) para *lanzar el programa*: hemos

tratado de incluir en esa función (o en alguna auxiliar de ella) todas las tareas de carga por teclado y visualización por pantalla.

Así, puede notarse que en un programa con estructura relativamente compleja habrá un *primer grupo de funciones cuyo objetivo será la administración de la interfaz de usuario*, y un *segundo grupo de funciones cuyo objetivo será resolver los problemas específicos del enunciado o requerimiento planteado*. Las primeras, necesariamente tendrán algún tipo de acoplamiento entre procesos e interfaz. Pero las segundas, deberían intentar ser diseñadas en forma genérica y al menos estas serían reusables cuando fuese necesario.

Note que una ventaja de esta estrategia, es que si el programa está orientado a una interfaz de consola estándar, y luego se decide hacer un *upgrade* o mejora pasándolo a una interfaz gráfica con ventanas y objetos visuales de alto nivel, entonces el programador sólo deberá cambiar las funciones del *primer grupo*, pero no debería tener que hacer ningún cambio en las del *segundo grupo*.

Proponemos el análisis y resolución del siguiente problema, especialmente pensado para resaltar los conceptos que hemos expuesto hasta aquí:

Problema 32.) *Desarrollar un programa controlado por menú de opciones, que incluya opciones para realizar las siguientes tareas:*

1. *Cargar un valor entero n por teclado, validando que sea mayor que 0, y mostrar todos los números impares y múltiplos de 3 que haya en el intervalo [1, n] (ambos incluidos).*
2. *Cargar dos valores enteros a y b por teclado, validando que 1 < a < b, y determinar si existe algún número primo en el intervalo [a, b]. Si existe alguno, muestre el primero que encuentre. Si no, informe con un mensaje.*
3. *Cargar por teclado una secuencia de números uno a uno, cortando el proceso cuando el número cargado sea el 0. Determinar si todos los números entraron ordenados de menor a mayor.*

Discusión y solución: El planteo de la estructura del *menú de opciones* fue analizado en la *Ficha 8*, por lo que remitimos al estudiante a ese contexto por si necesita revisarlo. El menú estará controlado por la siguiente función *menu()*, que no presenta dificultad alguna, y será esa función la que se invoque desde el script principal para comenzar el programa:

```
def menu():
    # titulo general...
    print('Menu de opciones y funciones generalizadas')

    op = 1
    while op != 4:
        # visualizacion de las opciones...
        print('1. Impares multiplos de 3')
        print('2. Primos en un intervalo')
        print('3. Secuencia ordenada')
        print('4. Salir')
        op = int(input('Ingrese el numero de la opcion elegida: '))

        # chequeo de la opcion elegida...
        if op == 1:
            opcion1()
        elif op == 2:
            opcion2()
        elif op == 3:
```

```
                opcion3()


    # script principal...
    menu()
```

Claramente esta función es una de las que tendrá como objetivo la gestión de la interfaz de usuario, y es de esperar entonces que realice operaciones de visualización por pantalla y carga por teclado.

Nos concentraremos ahora en la forma de resolver cada tarea pedida, de ser posible con funciones generalizadas. El primer requerimiento es mostrar todos los impares múltiplos de 3 en el intervalo [1, *n*], cargando *n* por teclado. La función *opcion1()* será invocada si el usuario elige la opción 1 del menú:

```
def validar_mayor_que(inf):
    n = inf - 1
    while n <= inf:
        n = int(input('Valor (mayor que ' + str(inf) + ' por favor...): '))
        if n <= inf:
            print('Error... se pidió > ' + str(inf), '... cargue de nuevo...')
    return n


def opcion1():
    n = validar_mayor_que(0)
    res = odd_multiples(n)
    print('Intervalo analizado: [ 1', ',', n, ']')
    print('Impares multiplos de 3 en ese intervalo:', res)
```

La carga por teclado del valor *n* se hace a través de la función auxiliar *validar_mayor_que(inf)*, la cual toma como parámetro un numero *inf*, y carga un valor por teclado controlando que ese valor sea mayor que *inf*. Como el valor *inf* viene parametrizado, la misma función puede usarse en cuanta situación requiera cargar por teclado un número controlando que sea mayor que otro dado.

Luego, la función *opcion1()* invoca a la función *odd_multiples(n)*, la cual puede plantearse, en principio, en forma simple y directa en forma similar a la que sigue para encontrar los números impares múltiplos de 3 (*considere que la función que sigue aún no es la versión definitiva*):

```
    # versión preliminar... será modificada!!!
    def odd_multiples(n):
        print('Impares y multiplos de 3 en [ 1', ',', n, ']:')
        for i in range(1, n+1):
            if i % 2 == 1 and i % 3 == 0:
                print(i)
```

En esta primera versión, la función toma como parámetro el valor *n* que indica el límite derecho del intervalo a analizar, y con un ciclo *for* explora sistemáticamente todos los números *i* en [1, *n*], chequeando uno por uno si efectivamente *i* es impar (*i % 2 == 1*) y a la vez múltiplo de (o divisible por) 3 (*i % 3 == 0*). En caso de serlo, el valor *i* se muestra en consola de salida.

La función cumple con lo pedido pero hay varios problemas con ella: el primero es que tiene dependencia con la interfaz de usuario (acoplamiento entre interfaz y procesos). No siempre será sencillo eliminar ese problema, sobre todo en las primeras semanas de un curso introductorio cuando aún no se cuenta con todas las herramientas que permitirían hacerlo

en forma simple, pero en este caso tenemos una vía: en lugar de mostrar cada número dentro de la función, podemos ir armando una *tupla* que contenga a esos números, y retornar esa tupla al final del ciclo. La visualización de la tupla (y de paso, también la visualización del mensaje inicial de la función) se deja para la función que invoque a *odd_multiples(n)* (en nuestro caso, la función *opcion1()* del ciclo del menú):

```
def odd_multiples(n):
    r = ()
    for i in range(1, n+1):
        if i % 2 == 1 and i % 3 == 0:
            r = r + i,
    return r
```

En esta segunda versión no hay acoplamiento de procesos e interfaz (ni dependencia externa). El proceso comienza definiendo una tupla vacía *r*, en la cual se irán agregando los números a medida que se encuentren. El ciclo *for* itera sobre el *range(1, n+1)* y cada valor *i* que sea impar y divisible por 3 se añade a la tupla *r* mediante la instrucción

```
r = r + i,
```

que es equivalente a:

```
r = r + (i,)
```

y también a:

```
r += i,    o bien a:    r += (i,)
```

Cualquiera de las cuatro expresiones hace que *r* pase a referir a una nueva tupla, que contiene lo mismo que ya contenía la original, pero ahora con el valor de *i* añadido al final de la tupla. Observe que el operador + permite *unir tuplas para formar una nueva* (del mismo modo que permite unir cadenas de caracteres), pero cada uno de los operandos debe ser a su vez una tupla [1] [2]. En nuestro caso, la expresión:

```
r = r + i
```

sin la coma a la derecha de la variable *i* estaría intentando unir la tupla *r* con el número entero *i*, y se produciría un error de intérprete:

```
Traceback (most recent call last):
  File "C:/Ejemplo/prueba8.py", line 191, in <module>
    menu()
  File "C:/ prueba8.py", line 14, in odd_multiples
    r = r + i
TypeError: can only concatenate tuple (not "int") to tuple
```

Al finalizar el ciclo *for*, la función *odd_multiples(n)* retorna la tupla completa, y con esto se ha eliminado cualquier acoplamiento procesos/interfaz. Sin embargo, habíamos sugerido que la función original tenía varios problemas, y el hecho es que aún subsiste uno: así como está planteada, la *función hace demasiado trabajo extra*, y si el valor de *n* fuese realmente grande (recuerde el problema de la factorización de números enteros…) la demora en el tiempo de ejecución podría comenzar a notarse. Como dijimos, posiblemente en este problema sencillo esa demora sería insignificante incluso para valores grandes de *n*, pero lo importante es que el programador se vaya formando de manera de saber identificar situaciones en las que podría ahorrar tiempo, y las resuelva tan eficientemente como se pueda en preparación a futuros problemas más exigentes.

En este caso, el ciclo *for* está tomando todo valor posible dentro del intervalo [1, *n*], aún sabiendo de antemano que sólo nos interesan los impares. Eso podría ajustarse en forma simple, haciendo que el *range* generado sólo contenga a los impares del intervalo citado:

```
def odd_multiples(n):
    r = ()
    for i in range(1, n+1, 2):
        if i % 3 == 0:
            r = r + i,
    return r
```

En la forma que acabamos de mostrar, el rango generado es de la forma (1, 3, 5, 7, …) y contiene solamente los números impares pedidos. El ciclo, por lo tanto, reduce la cantidad de repeticiones de *n* a *n*/2 (ya que tiene ahora la mitad de los números originales). Y como ahora sabemos que cada valor *i* tomado por el ciclo es efectivamente impar, ya no es necesario preguntar si *i* es impar en cada vuelta… sólo se chequea si *i* es múltiplo de 3.

Podría parecer que eso es todo. Pero todavía nos quedan algunas posibilidades de reducción de tiempo y/o código fuente. Por lo pronto, si sólo nos interesan los impares múltiplos de 3, no hay razón para que el *range* comience desde 1: *podría comenzar directamente desde el propio 3* y seguir desde allí:

```
for i in range(3, n+1, 2):
```

Y otra vez… nos interesan sólo los impares *múltiplos de 3*… Nos preguntamos si habrá forma de predecir dónde estarán esos números dentro de *[1, n]*, sabiendo que partimos desde el 3 (que es el primero que cumple la condición) y sólo iteramos sobre los impares. Un rápido análisis del rango iterado nos muestra lo siguiente:

```
(3, 5, 7, 9, 11, 13, 15, 17, 19, 21,…)
```

Empezando desde el 3, el siguiente impar múltiplo de 3 es el 9 (o sea, 3 + 6). Y el que sigue es 15 (que es 9 + 6). El próximo es 21 (que es 15 + 6)… y todo parece apuntar a que la regla es partir desde el 3 y sumar 6 en forma progresiva. El siguiente ciclo *for*:

```
for i in range(3, n+1, 6):
```

hace exactamente eso, generando el range (3, 9, 15, 21, 27,…) que sólo contiene impares múltiplos de 3. Por lo tanto, el ciclo hará aún menos repeticiones si recorre ese rango (aproximadamente un tercio de la mitad de *n*) y ni siquiera tendrá que seguir preguntando si *i* es múltiplo de 3:

```
def odd_multiples(n):
    r = ()
    for i in range(3, n+1, 6):
        r += i,
    return r
```

La regla de *comenzar desde 3 y sumar 6 en forma progresiva* puede formalizarse: todo número par es de la forma *2k* con *k* entero (el 2 es necesariamente un factor de su descomposición prima). Por lo tanto, un número impar es de la forma *2k + 1* (si un número de esta forma se divide por 2, necesariamente deja un resto de 1). Y todo impar *i* que a la vez sea múltiplo de 3, será entonces de la forma *i = 3(2k + 1)* (claramente, si un número de esta forma se divide por 3 deja un resto de 0) lo que equivale a *i = 6k + 3*. Con *k = 0, 1, 2,…* la sucesión que se obtiene es de la forma *(3, 9, 15, 21,…)* que es la que estábamos buscando.

Por razones de claridad, la anterior será nuestra versión final para la función *odd_multiples(n)*. Sin embargo, note que la tupla *r* finalmente generada y retornada, puede ser directamente creada a partir del *range* que está siendo iterado por el ciclo, usando la función *tuple()* que citamos más arriba como una de las formas de asignar una tupla [1]:

```
r = tuple(range(3, n+1, 6))
```

En este caso, la función *tuple()* toma cada uno de los valores del *range(3, n+1, 6)* y los agrega a la tupla en el mismo orden en que figuran en el *range()*. Por lo tanto, la función *odd_multiples(n)* en realidad ni siquiera necesitaría el *for* para iterar el *range()*: bastaría con retornar directamente la tupla generada:

```
def odd_multiples(n):
    r = tuple(range(3, n+1, 6))
    return r
```

En rigor, el tiempo que llevaría ejecutar esta función estaría en el mismo orden aproximado que el que llevaría ejecutar la versión anterior, con el *for* iterando el *range()* y armando la tupla un número a la vez, ya que la función *tuple()* hace internamente ese mismo trabajo.

La segunda tarea pedida en el enunciado general del problema era determinar si en el intervalo *[a, b]* existe al menos un número primo, y mostrar en ese caso el primero que se encuentre. Está claro que para resolver este problema, podemos *reutilizar* las funciones *is_prime(n)* y *next_prime(n)* que ya hemos analizado y presentado en la *Ficha 9*, y esa reutilización será perfectamente posible ya que ambas funciones se plantearon originalmente en forma genérica (sin dependencias externas y sin acoplamiento procesos/interfaz, gracias al uso correcto de parámetros y retornos).

La función *opcion2()* será la que se invoque cuando el usuario elija la opción 2 del menú, y será la encargada de leer desde el teclado los valores *a* y *b*, e invocar luego a la función que diseñemos para buscar el primer primo en *[a, b]*:

```
def opcion2():
    a = validar_mayor_que(1)
    b = validar_mayor_que(a)
    primo = search_prime(a, b)
    print('Intervalo analizado: [', a, ',', b, ']')
    if primo is not None:
        print('Hay al menos un numero primo en ese intervalo:', primo)
    else:
        print('No hay numeros primos en ese intervalo')
```

La función *opcion2()* carga el valor de *a*, usando la misma función *validar_mayor_que()* que se usó en la carga de *n* para el problema anterior, pero ahora controlando que *a* sea mayor a 1. De forma similar, se invoca otra vez a *validar_mayor_que()* para cargar *b*, controlando ahora que *b* sea mayor que *a*…

La función *search_prime(a, b)* será la encargada de cumplir el requerimiento de buscar el primer primo en *[a, b]*, que en realidad es muy sencillo si se cuenta con la función *next_prime(n)* ya programada (cosa que hicimos en la *Ficha 9*):

```
def search_prime(a, b):
    p = next_prime(a-1)
    if p <= b:
        return p

    return None
```

Como el propio valor *a* podría ser primo, invocamos a *next_prime()* pasando como parámetro el valor *a-1*. Con esto, la función retornará el primer número primo que encuentre que sea *mayor que a-1* (y si *a* es primo, será justamente *a* el número retornado). Sea cual fuese el primo *p* retornado, queremos saber si está en el intervalo *[a, b]*. Ya sabemos que *p >= a* por lo dicho anteriormente. Por lo tanto, sólo nos queda comprobar si a la vez *p* es menor o igual que *b* (cosa que se hace con la instrucción condicional dentro de la función). Si efectivamente es *p <= b*, se retorna *p* y se cumple así con el requerimiento: el valor *p* es el primer primo dentro de *[a, b]* (podría haber otros primos mayores que *p* dentro del mismo intervalo, pero sólo se pidió el primero). Pero si *p* fuese mayor que *b*, entonces ese primer primo encontrado estaría fuera de *[a, b]* y la función en ese caso retorna *None*.

La función *opcion2()* invoca a *search_prime(a, b)* asignando el valor retornado en la variable *primo*:

```
primo = search_prime(a, b)
```

y luego ***controla el valor almacenado en esa variable***, para saber si existía o no un primo en el intervalo [2] [1]:

```
if primo is not None:
    print('Hay al menos un numero primo en ese intervalo:', primo)
else:
    print('No hay numeros primos en ese intervalo')
```

El valor *None* puede ser usado a modo de *flag* o *bandera* (como aquí) para indicar un resultado exitoso o fallido, pero para chequear si una variable vale *None* o vale un valor distinto de *None*, debe usarse el operador ***is*** o la combinación ***is not***:

```
if x is None:
    # x vale None en esta rama...

if t is not None:
    # t no vale None en esta rama...
```

Finalmente, el tercer requerimiento del enunciado del problema era cargar una secuencia de números, cortando cuando llegue el 0, y simplemente determinar si esa secuencia entró ordenada de menor a mayor o no. La función *opcion3()* se activará cuando el usuario elija la opción 3 del menú, invocará a su vez a la función *check_order()* y mostrará un mensaje avisando el resultado del proceso:

```
def opcion3():
    ok = check_order()
    if ok:
        print('La secuencia cargada estaba ordenada de menor a mayor')
    else:
        print('La secuencia cargada NO estaba ordenada de menor a mayor')
```

La función *check_order()* usa un ciclo de carga por doble lectura para ingresar la secuencia a razón de un número por vuelta en la variable *num* [3]. Se usa una segunda variable *anterior* para almacenar en ella el valor de la vuelta anterior del ciclo. Si en cualquier vuelta de ese ciclo, el número cargado en esa misma vuelta en *num* fuese menor que el que se había cargado en la vuelta anterior (y que tenemos en la citada variable *anterior*), entonces sabremos que la secuencia cargada no está ordenada (al menos dos de sus números estaban en orden invertido). La función usa un *flag* llamado *ok* para indicar en todo momento lo que se sabe de la secuencia: el valor *True* se usa para indicar que hasta ese momento la secuencia está ordenada, y el valor *False* se usa para indicar que se ha detectado al menos un par de valores invertidos:

```
def check_order():
    # asumimos que la secuencia está ordenada...
    # ... ya que todavia no hemos cargado nada...
    # ... y la secuencia vacia de hecho está ordenada...
    ok = True

    # cargamos el primer numero...
    num = int(input('Cargue el primer valor (con 0 corta): '))

    # tomamos el primero como el "anterior" para la primera vuelta...
    anterior = num

    # ciclo de carga...
    while num != 0:

        # si el que tenemos es menor que el anterior,
        # entonces la secuencia está desordenada...
        # ... cambiar el flag ok a false
        if num < anterior:
            ok = False

        # almacenar el actual (num) como "anterior"
        # para la vuelta que sigue...
        anterior = num

        # cargamos el siguiente numero para continuar el ciclo...
        num = int(input('Cargue el siguiente valor (con 0 corta): '))

    # retornar el flag con el estado final...
    # True: estaba ordenada -  False: estaba desordenada
    return ok
```

Note un detalle: la función *check_order()* que hemos sugerido aquí está planteada con un evidente acoplamiento entre procesos e interfaz de usuario. En principio, la idea era que ese acoplamiento se permitiese sólo en aquellas funciones que estaban destinadas a la interfaz con el usuario (como la función *menu()*, y las funciones *opcion1()*, *opcion2()*, *opcion3()* y *validar_mayor_que()*), pero se eliminase en las funciones cuyo objetivo fuese el de resolver el problema o requerimiento central (como las funciones *odd_multiples()* y *search_prime()* o sus auxiliares *is_prime()* y *next_prime()*).

El hecho es que esa idea es conceptualmente correcta, y el programador debería hacer el mayor de sus esfuerzos por respetarla; pero en este caso el proceso está profundamente mezclado con la carga desde el teclado y hacer la separación implica la aplicación de herramientas del lenguaje que aún no hemos presentado en el curso (por ejemplo, el uso generalizado de estructuras de datos mutables y de tamaño adaptativo, como los *arreglos dinámicos* (llamados *listas* en Python)). Por este motivo, aceptaremos como versión final en este análisis para la función *check_order()* a la que acabamos de discutir, y oportunamente se estudiarán técnicas que permitirán desacoplar los procesos y la interfaz de usuario también en casos como este.

El programa completo en su versión final, sería entonces el siguiente:

```
__author__ = 'Catedra de AED'


# funciones "grupo 2": genéricas, para resolver cada problema...
def is_prime(n):
    # numeros negativos no admitidos...
    if n < 0:
```

```
        return None

    # por definicion, el 1 no es primo...
    if n == 1:
        return False

    # si n = 2, es primo y el unico primo par...
    if n == 2:
        return True

    # si no es n = 2 pero n es par, no es primo...
    if n % 2 == 0:
        return False

    # si llegamos aca, n es impar... por lo tanto no hace falta
    # probar si es divisible por numeros pares...
    # ... y alcanza con probar divisores hasta el valor pow(n, 0.5))...
    raiz = int(pow(n, 0.5))
    for div in range(3, raiz + 1, 2):
        if n % div == 0:
            return False

    return True


def next_prime(n):
    # si n es menor a 2, el siguiente primo es 2...
    # ... no nos preocupa si n es negativo...
    # ... buscamos primos naturales...
    if n < 2:
        return 2

    # Si n es par (puede ser n = 2, pero no nos afecta)
    # entonces el SIGUIENTE posible primo p no es 2...
    # ...y es impar...
    p = n + 1
    if p % 2 == 0:
        p += 1

    # ahora p es impar... comenzar con el propio p
    # y buscar el siguiente impar que sea primo...
    while not is_prime(p):
        p += 2

    # ... y retornarlo
    return p


def odd_multiples(n):
    # asignar en r una tupla vacia...
    r = ()

    # iterar el rango de valores impares entre 3 y n...
    # ...y tomar solo los multiplos de 3...
    # ...que comienzan en 3 y siguen uno cada 6...
    for i in range(3, n+1, 6):
        # agregar i a la tupla r...
        r += i,

    # retornar la tupla con los números encontrados
    return r


def search_prime(a, b):
    # tomar el siguiente primo "p" desde a-1...
    # que puede ser el propio "a"...
    # ... u otro mayor que "a"...
    p = next_prime(a-1)
```

```
    # ... si p es menor o igual que b, retornar p...
    # ... ya que entonces a <= p <= b...
    if p <= b:
        return p

    # ... y si p > b, pues retornar None...
    return None


def check_order():
    # asumimos que la secuencia esta ordenada...
    # ... ya que todavia no hemos cargado nada...
    # ... y la secuencia vacia de hecho esta ordenada...
    ok = True

    # cargamos el primer numero...
    num = int(input('Cargue el primer valor (con 0 corta): '))

    # tomamos el primero como el "anterior" para la primera vuelta...
    anterior = num

    # ciclo de carga...
    while num != 0:

        # si el que tenemos es menor que el anterior,
        # entonces la secuencia esta desordenada...
        # ... cambiar el flag ok a false
        if num < anterior:
            ok = False

        # almacenar el actual (num) como "anterior"
        # para la vuelta que sigue...
        anterior = num

        # cargamos el siguiente numero para continuar el ciclo...
        num = int(input('Cargue el siguiente valor (con 0 corta): '))

    # retornar el flag con el estado final...
    # True: estaba ordenada -  False: estaba desordenada
    return ok


# funciones "grupo 1": gestionan la interfaz de usuario...
def validar_mayor_que(inf):
    n = inf - 1
    while n <= inf:
        n = int(input('Valor (>' + str(inf) + ' por favor...): '))
        if n <= inf:
            print('Error... se pidió >' + str(inf), '... cargue de nuevo...')
    return n


def opcion1():
    n = validar_mayor_que(0)
    res = odd_multiples(n)
    print('Intervalo analizado: [ 1', ',', n, ']')
    print('Impares multiplos de 3 en ese intervalo:', res)


def opcion2():
    a = validar_mayor_que(1)
    b = validar_mayor_que(a)
    primo = search_prime(a, b)
    print('Intervalo analizado: [', a, ',', b, ']')
    if primo is not None:
        print('Hay al menos un numero primo en ese intervalo:', primo)
    else:
```

```
        print('No hay numeros primos en ese intervalo')


def opcion3():
    ok = check_order()
    if ok:
        print('La secuencia cargada estaba ordenada de menor a mayor')
    else:
        print('La secuencia cargada NO estaba ordenada de menor a mayor')


def menu():
    # titulo general...
    print('Menu de opciones y funciones generalizadas')

    op = 1
    while op != 4:
        # visualizacion de las opciones...
        print('1. Impares multiplos de 3')
        print('2. Primos en un intervalo')
        print('3. Secuencia ordenada')
        print('4. Salir')
        op = int(input('Ingrese el numero de la opcion elegida: '))

        # chequeo de la opcion elegida...
        if op == 1:
            opcion1()
        elif op == 2:
            opcion2()
        elif op == 3:
            opcion3()


# script principal...
menu()
```

4.] Elementos de cálculo combinatorio.

El *cálculo combinatorio* (o también *análisis combinatorio*) es una rama de las matemáticas (y en particular de la matemática discreta) cuyo objeto general es el estudio de las posibilidades en que pueden combinarse los *n* elementos de un conjunto, agrupándolos de maneras diferentes (ya sea en grupos de *m* elementos con *m* < *n*, o en grupos de exactamente *n* elementos, etc.) En algunos problemas importará enumerar o caracterizar cada una de las formas posibles de combinación, y en otros será suficiente sólo con indicar cuántas maneras diferentes existen para hacer esas combinaciones [4].

Un elemento básico del cálculo combinatorio es el llamado *Principio Fundamental de Conteo* (o *Principio Básico de Conteo*) que expresa *cuántas resultados posibles existen de combinar k eventos*: Si un evento cualquiera puede presentarse en *e1* formas diferentes, y luego otro evento puede presentarse en *e2* formas distintas, y otros eventos sucesivos pueden aparecer en *e3*, *e4*, … *ek* formas diferentes, entonces la cantidad total de formas en que pueden combinarse los *k* eventos presentados es igual a:

```
t = e1 * e2 * e3 * … * ek
```

Un ejemplo aclara la idea: suponga que en un comercio de venta de prendas de vestir se tiene que preparar un maniquí en vidriera, y para hacerlo deben colocarle al menos un pantalón, un par de zapatos y una camisa. Si el empleado encargado de esa tarea tiene siete diferentes modelos de pantalones (*e1 = 7*), cuatro diferentes modelos de zapatos (*e2 = 4*) y

ocho modelos distintos de camisas (*e3 = 8*), entonces ese empleado tendrá que elegir entre una cantidad total *t* de variantes igual a:

```
t = e1 * e2 * e3
t = 7 * 4 * 8
t = 224 resultados posibles
```

Otra necesidad común del análisis combinatorio consiste en determinar en cuántas formas diferentes pueden arreglarse o combinarse los *n* elementos de un conjunto, tomados todos a la vez, pero de forma que cada uno aparezca una sola vez en cada arreglo (no valen repeticiones), y de forma que el orden importe: dos variaciones con los mismos elementos pero en diferente orden, se consideran distintas. Esto se conoce como el *número de permutaciones de n elementos, tomados de a n*, y podemos denotarlo como *p(n, n)*.

A modo de ejemplo, suponga que se desea saber cuántos posibles números diferentes pueden formarse con los dígitos 1, 2, 3 y 4, sin repetir ninguno de ellos. Evidentemente, cada arreglo o permutación de estos cuatro números tendrá un total de cuatro dígitos. Para el primer dígito de la izquierda, se puede elegir a cualquiera de los cuatro originales, por lo cual tenemos 4 posibles elecciones de partida.

Pero para el segundo dígito, como no podemos repetir ninguno, sólo podremos elegir alguno que no sea igual al que se eligió como primer dígito, quedando entonces 3 posibilidades adicionales para cada una de las 4 iniciales. Usando el mismo argumento, el tercer dígito sólo tendrá 2 alternativas y el cuarto dispondrá de sólo una. Se deduce que la cantidad de permutaciones p(4, 4) (cuatro dígitos diferentes tomados de a cuatro, sin repetir dígitos) será igual a 4*3*2*1 = 4! = 24 permutaciones.

Esto puede verse con mucha claridad usando un diagrama tipo árbol para exponer las permutaciones [5] (ver *Figura 2*). Entonces es relativamente simple probar que:

```
p(n, n) = n!
```

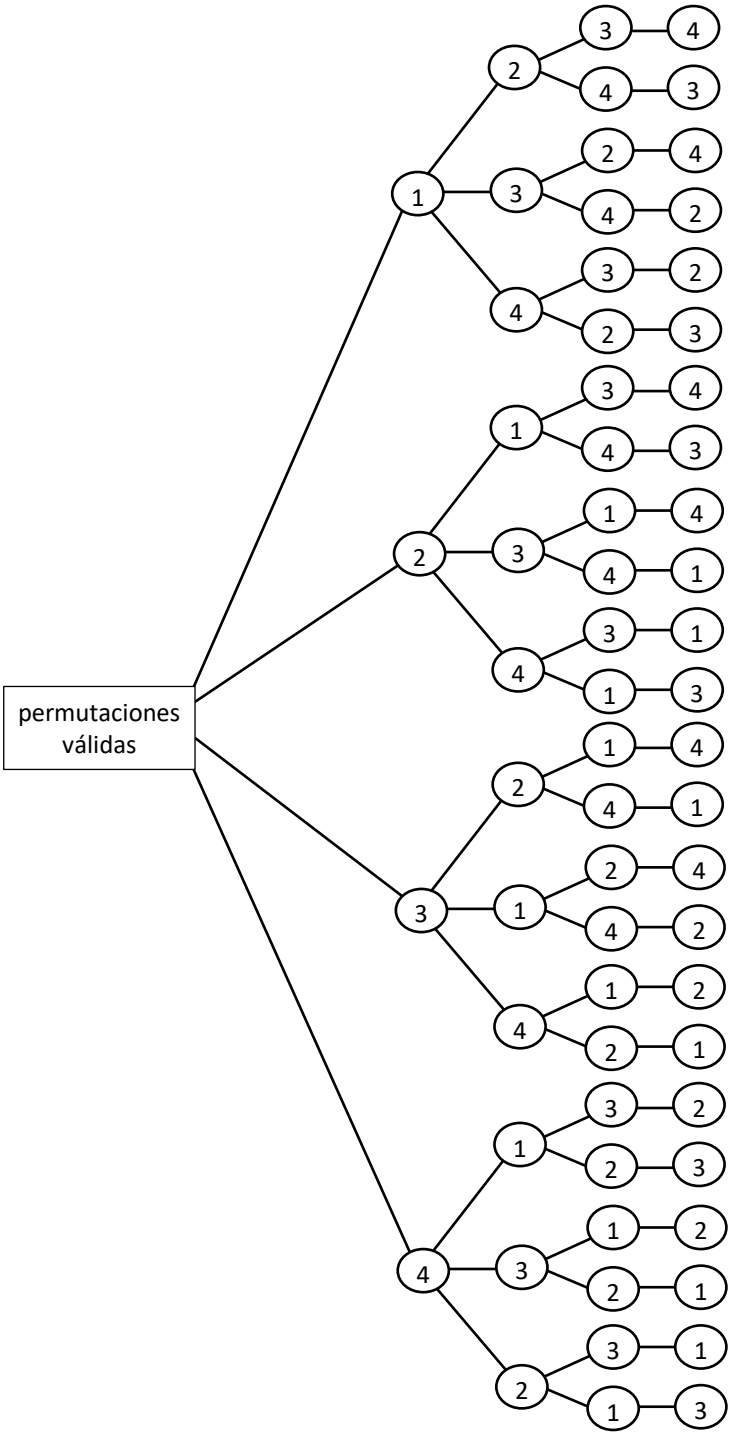
Efectivamente: si en cada arreglo o permutación puede aparecer sólo una vez cada uno de los *n* elementos originales y el orden importa, entonces la posición de partida puede ocuparse de *n* formas distintas. La segunda posición sólo puede ocuparse de *n-1* formas diferentes con el resto de los números, y la progresión será:

```
p(n, n) = n * (n-1) * (n-2) * … * 3 * 2 * 1
p(n, n) = n!
```

En general, la idea de *permutación* implica que *el orden en que se presentan los elementos es relevante*. Así, cada una de las 24 formas de combinar 4 dígitos diferentes es una permutación diferente de las otras (aunque todas tienen los mismos números) ya que en todas los números aparecen en distinto orden.

En muchas situaciones el problema se restringe un poco: se tienen efectivamente *n* elementos en un conjunto, y se quiere determinar la cantidad de permutaciones diferentes que pueden obtenerse *pero tomando los elementos en grupos de m (con m < n), y sin permitir repeticiones.* Como se trata de permutaciones, el *orden es relevante*: se toman como diferentes dos variaciones que tengan los mismos elementos pero en distinto orden. Usaremos la notación *p(n, m) (con m <= n)* para referirnos a permutaciones en general.

Figura 2: Diagrama tipo árbol – Permutaciones de los números 1, 2, 3 y 4 tomados de a 4.



Por ejemplo, suponga que en una universidad se debe seleccionar a los tres mejores estudiantes para distinguirlos como el abanderado y los dos escoltas, y las autoridades deben decidir entre siete estudiantes que fueron propuestos por distintos méritos. Se quiere saber de cuántas formas posibles podría quedar formado el trío de alumnos distinguidos. En este caso, el número de alumnos del conjunto es $n = 7$, y se desea agruparlos de a $m = 3$, pero de forma que, otra vez, será importante el orden en que queden. Así una terna formada por "Juan", "Luis" y "Pedro" será tomada como diferente de otra como "Pedro", "Luis" y "Juan", ya que en ambas las distinciones corresponderán a alumnos diferentes.

Se trata de calcular p*(n, m)* = *p(7, 3)* = cantidad de permutaciones existentes, tomando 7 elementos en grupos de 3 distintos, sin repetirlos, y asumiendo como diferentes a aquellas en que los mismos elementos aparezcan en distinto orden. Podemos ver que cada terna posible de jugadores tendrá 7 posibilidades diferentes en cuanto al primer lugar (el abanderado), y el segundo lugar (primer escolta) será cubierto por alguno de los otros 6, quedando hasta aquí un total de 7 * 6 permutaciones. El tercer lugar (segundo escolta) sólo puede ser cubierto por alguno de los otros 5, quedando un total de 7*6*5. Pero como sólo se pide permutarlos de tres en tres, el proceso finaliza allí… y el total será de 210 posibles ternas [4].

Está claro que en el cálculo de *p(7, 3)* no puede haber más de *m = 3* factores: la cantidad de posibilidades para el primer lugar (*n = 7*) multiplicado por *n-1* (= 6) y luego por *n-2* (= 5):

```
p(7, 3) = n * (n-1) * (n-2) = 7 * 6 * 5 = 210
```

con lo que la multiplicación debe detenerse al llegar al 5. Puede verse entonces que el último factor es de la forma: *n – m + 1 = 7 – 3 + 1 = 5*. Por lo tanto, la cantidad de permutaciones *p(n, m)* (con *m < n*) es:

```
p(n, m) = n * (n-1) * (n-2) * … * (n-m+1)
```

y puede probarse por inducción matemática que lo anterior equivale a su vez a [5]:

$$p(n, m) = \frac{n!}{(n - m)!}$$

Es sencillo comprobar que en el caso particular en que *m = n*, la fórmula anterior lleva a:

```
p(n, m) = p(n, n) = n! / (n - n)! = n! / 0! = n!
```

lo cual ya sabíamos y era esperable: si *m = n*, entonces se está pidiendo lo que ya habíamos calculado: la cantidad de permutaciones tomando *n* elementos en grupos de *n*.

Hasta aquí hemos hablado de permutaciones *sin repetición* de elementos (también llamadas *permutaciones sin reposición*: una vez que se usa un elemento, no se repone al conjunto para volver a usarlo). Pero en algunos casos se requiere saber cuántas permutaciones podrían formarse *permitiendo que los elementos del conjunto se repitan*. Es el típico caso (por ejemplo) de determinar la cantidad de variantes que pueden formarse con los dígitos de una *cerradura de combinación* (que en realidad, debería entonces llamarse *cerradura de permutación*) de un candado o de una caja de seguridad.

Suponga que un candado tiene un mecanismo de seguridad de *m = 4* posiciones, y se quiere determinar cuántas permutaciones posibles existen para ese mecanismo. Cada posición se ocupa con un número entre 0 y 9 (o sea, el conjunto desde el cual se seleccionan los números tiene *n = 10* elementos) y deben permutarse tomados de a *m = 4*. La primera posición de la cerradura puede ocuparse con cualquiera de los *n* dígitos. Y como todos los números siguen siendo válidos para ocupar la segunda posición, entonces esta puede volver a ocuparse con cualquiera de los *n* dígitos teniendo hasta aquí, n * n = 10 * 10 variantes. Y como los *n* dígitos vuelven a ser válidos para la tercera y la cuarta posición, resulta que el total de permutaciones es *n * n * n * n = 10 * 10 * 10 * 10 = 10^4 = 10^m = 10000* permutaciones posibles.

Puede verse entonces que si el conjunto de partida tiene *n* elementos, y se quiere permutarlos tomados de a *m*, pero permitiendo repeticiones (esto es, *permutaciones con reposición*) entonces el total *pr(n, m)* surge de la sencilla fórmula siguiente:

$$pr(n, m) = n^m \quad (m <= n)$$

Las permutaciones tienen en cuenta el orden final de cada variación de elementos. Pero en muchos casos, *ese orden no es relevante y dos variaciones de los mismos elementos se deben tomar como iguales a los efectos del conteo*. Si ese es el caso, se habla de *combinaciones* en lugar de *permutaciones*.

Por ejemplo, suponga que en un evento deportivo para reunir fondos para una obra caritativa se tienen *n = 6* jugadores profesionales y se desea que en cada uno de los dos equipos que participarán del evento haya siempre *m = 3* jugadores profesionales. ¿En cuántas formas diferentes estos jugadores podrían repartirse en estas condiciones?

Puede verse que ahora no es importante el orden: un agrupamiento de tres jugadores como "Diego", "Lionel" y "Gabriel" será exactamente el mismo que otro de la forma "Gabriel", "Diego" y "Lionel", y no deben contarse como dos diferentes. Tampoco es válido en este caso contemplar repeticiones: no es posible que un mismo jugador aparezca más de una vez en un grupo. Por lo tanto, hablamos de cuántas *combinaciones sin repetición* (o *sin reposición*) pueden formarse tomando de a 3 a los 6 jugadores.

La forma general de calcular ese número puede pensarse así: se calculan las permutaciones de 6 tomados de a 3, como si el orden fuese relevante. En este caso, sería 6*5*4 = 120. Pero luego debemos pensar que en esas 120 permutaciones hay muchas que son variantes ordenadas de los mismos nombres, y deben ser eliminadas del cálculo. ¿Cuántas son las que sobran? Es simple: Con *m = 3* nombres se pueden formar *3*2*1 = 3! = m! = 6* permutaciones ordenadas, que en realidad son iguales entre sí si se las toma como combinaciones. Por lo tanto, para eliminar del cálculo general *a todas* las permutaciones repetidas, se divide por 6 al total de 120 anterior, quedando 20 formas de agrupar a estos jugadores.

Entonces, la fórmula general para calcular las *combinaciones* de *n* elementos tomados de a *m* (con *m <= n*) (denotada aquí como *c(n, m)*) es [5]:

$$c(n, m) = \frac{n!}{m! * (n - m)!}$$

Otra vez, note que si *n = m* entonces la fórmula anterior se reduce a *n! / n! = 1* lo cual de nuevo era esperable: hay una sola forma de combinar *n* elementos en grupos de a *n* sin que sea relevante el orden: tomarlos a todos juntos.

Finalmente, nos preguntamos qué pasaría si teniendo un conjunto de *n* elementos, quisiéramos saber el total de combinaciones tomados de a *m*, pero admitiendo repeticiones (o sea, *combinaciones de n elementos en grupos de m, con reposición*, que denotaremos como *cr(n, m)*).

Por ejemplo, suponga que se tienen *n = 5* recipientes, cada uno conteniendo un tipo diferente de fruta (por caso, un recipiente de naranjas, otro de manzanas, y los otros durazno, peras y ciruelas) Suponga que se nos indica que podemos tomar hasta *m = 3* frutas, seleccionándolas como queramos. Está claro que una selección de la forma (naranja, manzana, durazno) será igual a otra de la forma (manzana, durazno, naranja) y por lo tanto,

el orden en que se combinen no es relevante. Y como ahora sería válido elegir dos o tres futas del mismo recipiente, entonces una combinación de la forma (naranja, naranja, manzana) también sería válida. Si se pide determinar cuántas variantes de combinación existen en estas condiciones, se tiene un caso de *combinaciones de n elementos tomados en grupos de a m, con reposición* (*cr(n, m)*).

Para deducir la fórmula a utilizar, supongamos que cada vez que se toma una fruta de un recipiente esa operación se marca como *"Tomar"* (y simbolizamos como *T*), y cada vez que se salta a un recipiente desde otro (sea que tome una fruta o no del anterior) marcamos ese hecho como *"Saltar"* (simbolizado como *S*). Entonces, las siguientes combinaciones podrían representarse como sigue (suponiendo que los recipientes están en orden indicado más arriba):

- ✓ (naranja, naranja, pera) ⇒ (T T) S S S (T) S

  Al comenzar, estamos en el recipiente de naranjas y tomamos 2 (T T). Luego, la primera S nos saca del recipiente de naranjas y nos lleva al de manzanas. Como no queremos manzanas, no tomamos ninguna y la segunda S nos pone en el recipiente de duraznos. Tampoco queremos duraznos, y la tercera S nos deja en el de peras, donde tomamos una (T). Finalmente, (¡para salir del puesto de venta!), saltamos otra vez y la última S nos lleva al recipiente de ciruelas (y ya no volvemos saltar… la S significa "*salte desde un recipiente a otro*")

- ✓ (manzana, pera, ciruela) ⇒ S (T) S S (T) S (T)

  Al comenzar estamos en el de naranjas y sin tomar ninguna saltamos (la primera S) al de manzanas. Allí tomamos una (la primera (T)) y pasamos al de duraznos (la segunda S). No tomamos ninguno y pasamos al de peras (la tercera S) donde tomamos una (la segunda (T)). De allí seguimos al de ciruelas (la cuarta S) y tomamos una (la última (T)).

Con este sistema de codificación, podemos notar que para *n = 5* y *m = 3*, en el cálculo de combinaciones con reposición siempre tenemos *m + (n – 1) = 3 + (5 - 1) = 7* posiciones *en total* (marcadas como S o como T) y de esas 7, siempre hay *m = 3* posiciones marcadas como T. Por lo tanto, el problema puede plantearse como *calcular la cantidad de formas en que pueden combinarse m + (n – 1) tomados de a m (cantidad de posiciones marcadas como T)*, sin importar si hay reposición o no.

Es decir, podemos calcular la cantidad *cr(n, m)* (*n* elementos combinados de a *m*, con reposición) como si se tratase de *c(m+(n-1), m)* (o sea: *m+(n-1)* elementos, combinados de a *m*, sin reposición) y la fórmula sería:

$$cr(n, m) = c(m + (n - 1), m) = \frac{(m + (n - 1))!}{m! * (n - 1)!}$$

Y en el ejemplo de *n = 5* recipientes y *m = 3* frutas en total, el cálculo quedaría:

$$cr(5, 3) = \frac{(3 + (5 - 1))!}{3! * (5 - 1)!} = \frac{7!}{3! * 4!} = \frac{5040}{6 * 24} = 35$$

Tenemos 35 formas de seleccionar 3 frutas de 5 recipientes diferentes, con opción a repetir la fruta que queramos.

Luego de este análisis completo, podemos poner todo en un tabla a modo de resumen de fórmulas y situaciones:

Figura 3: Tabla de fórmulas y situaciones generales del cálculo combinatorio.

Tipo	¿Importa el orden?	¿Admite repeticiones?	Situación	Fórmula
1. Permutación	sí	no	*n* elementos permutados de a *n*, sin reposición.	$p(n,n) = n!$
2. Permutación	sí	no	*n* elementos permutados de a *m (m <= n)*, sin reposición.	$p(n,m) = \frac{n!}{(n-m)!}$
3. Permutación	sí	sí	n elementos permutados de a *m (con m <= n)*, con reposición.	$pr(n,m) = n^m$
4. Combinación	no	no	*n* elementos combinados de a *m (con m <= n)*, sin reposición.	$c(n,m) = \frac{n!}{m!\,(n-m)!}$
5. Combinación	no	sí	*n* elementos combinados de a *m (con m <= n)*, con reposición.	$cr(n,m) = \frac{(m+(n-1))!}{m!\,(n-1)!}$
6. Conteo de resultados	Total de resultados posibles de combinar un evento con *e1* variantes, con otro de *e2* variantes, y otro con *e3*, y así hasta uno al final con *ek* variantes.			$t = e1 * e2 * e3 * \ldots * ek$

Como conclusión, podemos ver que el uso de factoriales resulta ser mucho más natural de lo que se habría esperado, ya que se requiere como operación fundamental en muchos problemas de conteo. Y podemos trabajarlo en la resolución de un problema de aplicación, para cerrar esta Ficha:

Problema 33.) *Desarrollar un programa controlado por menú de opciones, que incluya opciones para realizar las siguientes tareas:*

1. *Cargar por teclado la cantidad n de colores que pueden tener los automóviles a la venta en una concesionaria, además de la cantidad m de modelos que existen de una marca en particular, y finalmente la cantidad t de opciones que se ofrecen a los clientes en cuanto a tipos de neumáticos a aplicar en cada vehículo. Calcule la cantidad total de combinaciones que tiene un cliente cuando deba elegir un automóvil de esa marca.*
2. *Para el diseño de un juego de tablero basado en combinar n sílabas, se desea saber cuántas formas diferentes existen de combinar esas n sílabas, sin repetirlas. Cargar n por teclado y mostrar esa cantidad.*
3. *Cargar por teclado la cantidad n de agentes de seguridad disponibles en una compañía de servicios de vigilancia. La compañía desea que en todo momento, sus agentes se distribuyan en grupos de m agentes. Calcule la cantidad de grupos diferentes que pueden formarse.*
4. *Cargar por teclado la cantidad n de corredores que participan de una carrera de Fórmula 1. Cargue también por teclado la cantidad de m, con m < n, de posiciones finales que tendrán puntos por llegar en esas posiciones. Calcular y mostrar la cantidad de posibles formas en que pueden cubrirse las m posiciones puntuables, con los n corredores.*
5. *Cargar por teclado la cantidad m de letras que conforman la clave de identificación asignada a los empleados de una empresa. Sabiendo que las posibles letras diferentes del alfabeto español son n = 27, calcular la cantidad de formas que podría tener una clave.*

6. *Cargar por teclado la cantidad n de colores que pueden tener las fichas contenidas en una bolsa, y la cantidad m de fichas que se deben extraer de la bolsa para un ejercicio de probabilidades. Calcular la cantidad de variantes que puede tener cada conjunto posible de m fichas en cuanto a sus posibles colores.*

Discusión y solución: Se trata de un programa para aplicar en forma general las fórmulas presentadas en esta sección.

Para el punto 1 se aplica en forma directa el cálculo del *total de resultados r = n * m * t* que se explicó más arriba (*fórmula 6* de la tabla de la *Figura 3*). La función que sigue hace el trabajo:

```
def total_resultados(n, m, t):
    return n * m * t
```

En el punto 2 se necesita saber la cantidad de *permutaciones* posibles que pueden formarse con *n* sílabas diferentes tomadas de a *n* (esto es, sin repetir sílabas y considerando como diferentes a dos variantes con las mismas sílabas pero en otro orden). Sabemos que en este caso el total de permutaciones es *p(n, n) = n!* (*fórmula 1* de la *Figura 3*) y eso es lo que aplica la siguiente función *permutaciones_sin_reposicion()*, asumiendo que la función *factorial()* está definida y disponible:

```
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

def permutaciones_sin_reposicion(n, m):
    fn = factorial(n)
    fd = factorial(n - m)
    return fn // fd
```

Note que la función *permutaciones_sin_reposicion()* toma dos parámetros *n* y *m* que pueden ser diferentes o también pueden ser iguales. Si *n = m*, la función calculará el valor *n!* y lo retornará. Y si fuese *n > m*, la función calculará las permutaciones que correspondan y también retornará ese valor. La misma función puede usarse para casos en que *n = m* o también en casos en que *n > m* (es decir, aplica tanto para la *fórmula 1* como para la *fórmula 2* de la *Figura 3*)

En el punto 3, se pide la cantidad de *combinaciones* en que pueden agruparse *n* agentes, tomados de a *m*: ahora los grupos con exactamente los mismos agentes son iguales, ya que el orden en que llegan al grupo no es relevante. Y las combinaciones no pueden incluir más de una vez al mismo agente, por lo que se trata de combinaciones sin reposición. La fórmula a aplicar es entonces la *fórmula 4* de la *Figura 3.* La siguiente función hace el cálculo pedido:

```
def combinaciones_sin_reposicion(n, m):
    fn = factorial(n)
    fm = factorial(m)
    fr = factorial(n - m)
    return fn // (fm * fr)
```

El punto 4 es una situación de cálculo de *permutaciones de n tomados de a m, sin reposición*: hay *n* pilotos de carreras y *m* puestos que otorgan puntaje. Como no es lo mismo llegar

primero que segundo o tercero (el puntaje obtenido será diferente), entonces el orden es importante y eso nos define que se trata de permutaciones. Como además, ningún piloto puede llegar en más de una posición (o es primero o es segundo, pero no puede ser primero y segundo al mismo tiempo), entonces sabemos que las *permutaciones son sin repetición* (o *sin reposición*). Por lo tanto, aplica la *fórmula 2* de la *Figura 3*, que ya teníamos programada en la función *permutaciones_sin_reposicion()*:

```
def permutaciones_sin_reposicion(n, m):
    fn = factorial(n)
    fd = factorial(n - m)
    return fn // fd
```

El requerimiento 5 del problema es calcular cuántas formas distintas podría tener una clave formada por *n* letras, sabiendo que cada letra puede ser una de 27 posibles. Está claro que dos claves con las mismas letras pero en orden diferente, valen como distintas, y por lo tanto tenemos un problema de *permutaciones: el orden importa*. El conjunto de letras está formado por 27 letras posibles (*n = 27*), y cada clave debe formarse combinando *m* letras, por lo que tenemos que permutar *n* elementos tomados de a *m*. Además, cada letra podría repetirse en una clave, por lo que finalmente tenemos un problema de permutaciones de *n* tomados de a *m*, con reposición, y aplica la *fórmula 3* de la *Figura 3*. La siguiente función calcula ese valor:

```
def permutaciones_con_reposicion(n, m):
    return pow(n, m)
```

Finalmente, el punto 6 supone fichas de *n* colores guardadas en una bolsa, y se pide calcular en cuántas formas podrían salir combinados esos colores tomando aleatoriamente *m* fichas de la bolsa en un momento dado. Como dos arreglos de fichas de los mismos colores pero en diferente orden serían iguales, entonces estamos ante un problema de cálculo de *combinaciones* (*el orden no es relevante*). Y como podrían aparecer fichas del mismo color cada vez que se extraigan *m* fichas, entonces las combinaciones *admiten reposición*. Se trata de combinaciones de *n* colores, tomados de a *m* por vez, *con reposición*. Aplica la *fórmula 5* de la *Figura 3*, y la función que sigue hace ese trabajo:

```
def combinaciones_con_reposicion(n, m):
    fn = factorial(m + (n - 1))
    fm = factorial(m)
    fr = factorial(n - 1)
    return fn // (fm * fr)
```

El programa completo se muestra a continuación:

```
# funciones "grupo 2": genéricas... resuelven problemas específicos...
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f


def permutaciones_con_reposicion(n, m):
    return pow(n, m)


def permutaciones_sin_reposicion(n, m):
    fn = factorial(n)
```

```
    fd = factorial(n - m)
    return fn // fd


def combinaciones_con_reposicion(n, m):
    fn = factorial(m + (n - 1))
    fm = factorial(m)
    fr = factorial(n - 1)
    return fn // (fm * fr)


def combinaciones_sin_reposicion(n, m):
    fn = factorial(n)
    fm = factorial(m)
    fr = factorial(n - m)
    return fn // (fm * fr)


def total_resultados(n, m, t):
    return n * m * t


# funciones "grupo 1": gestionan la interfaz de usuario...
def validar_mayor_que(inf):
    n = inf - 1
    while n < inf:
        n = int(input('Valor (mayor que ' + str(inf) + ' por favor...): '))
        if n < inf:
            print('Error... se pidió >' + str(inf), '... cargue de nuevo...')
    return n


def validar_menor_que(sup):
    n = sup + 1
    while n > sup:
        n = int(input('Valor (menor que ' + str(sup) + ' por favor...): '))
        if n > sup:
            print('Error... se pidió <' + str(sup), '... cargue de nuevo...')
    return n


def validar_intervalo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor (entre ' + str(inf) + ' y ' + str(sup)+ ': '))
        if n < inf or n > sup:
            print('Se pidió entre', inf, 'y', sup, '... cargue de nuevo...')
    return n


def opcion1():
    print('Cantidad de colores...')
    n = validar_mayor_que(0)

    print('Cantidad de modelos...')
    m = validar_mayor_que(0)

    print('Cantidad de tipos de neumáticos...')
    t = validar_mayor_que(0)

    res = total_resultados(n, m, t)
    print('Cantidad total de combinaciones:', res)


def opcion2():
    print('Cantidad de sílabas diferentes...')
    n = validar_mayor_que(0)
    tp = permutaciones_sin_reposicion(n, n)
```

```
    print('Permutaciones de', n, 'sílabas tomadas de a', n, ':', tp)


def opcion3():
    print('Cantidad de agentes...')
    n = validar_mayor_que(0)

    print('Cantidad de agentes por grupo...')
    m = validar_intervalo(0, n)

    tp = combinaciones_sin_reposicion(n, m)
    print('Combinaciones de', n, 'agentes tomados en grupos de a', m, ':', tp)


def opcion4():
    print('Cantidad de pilotos...')
    n = validar_mayor_que(0)

    print('Cantidad de puestos con puntaje...')
    m = validar_intervalo(0, n)

    tp = permutaciones_sin_reposicion(n, m)
    print('Permutaciones de', n, 'pilotos tomados de a', m, ':', tp)


def opcion5():
    n = 27
    print('La cantidad de letras a combinar es', n)

    print('Cantidad de posiciones en cada clave...')
    k = validar_mayor_que(0)

    tp = permutaciones_con_reposicion(n, k)
    print('Permutaciones de', n, 'letras tomadas de a', k, 'con repeticiones:', tp)


def opcion6():
    print('Cantidad de colores...')
    n = validar_mayor_que(0)

    print('Cantidad de fichas a extraer...')
    m = validar_mayor_que(0)

    tp = combinaciones_con_reposicion(n, m)
    print('Combinaciones de', n, 'colores tomados de a', m, 'con reposicion:', tp)


def menu():
    # titulo general...
    print('Menu de opciones - Cálculo combinatorio')

    op = 1
    while op != 7:
        # visualizacion de las opciones...
        print('1. Total de ofertas de vehículos')
        print('2. Permutaciones de sílabas')
        print('3. Combinaciones de agentes de seguridad')
        print('4. Permutaciones de pilotos de carreras')
        print('5. Permutaciones de letras en una clave')
        print('6. Combinaciones de colores')

        print('7. Salir')
        op = int(input('Ingrese el numero de la opcion elegida: '))

        # chequeo de la opcion elegida...
        if op == 1:
            opcion1()
        elif op == 2:
```

```
            opcion2()
        elif op == 3:
            opcion3()
        elif op == 4:
            opcion4()
        elif op == 5:
            opcion5()
        elif op == 6:
            opcion6()


# script principal...
menu()
```

El programa incluye tres funciones de carga con validación, que seguramente resultarán útiles al estudiante:

- validar_mayor_que(inf): carga y retorna un número entero, validando que ese número sea mayor al valor *inf* tomado como parámetro.
- validar_menor_que(sup): carga y retorna un número entero, validando que ese número sea menor al valor *sup* tomado como parámetro.
- validar_intervalo(inf, sup): carga y retorna un número entero, validando que ese número sea mayor al valor *inf* tomado como parámetro y al mismo tiempo menor que el valor *sup* también tomado como parámetro.

En este caso la segunda no está siendo utilizada en el programa, pero se deja disponible de todos modos por su utilidad práctica.

Las funciones *opcion1()*, *opcion2()* y *opcion3()* son similares a las que ya hemos presentado en otros programas con menú de opciones, y son las encargadas de manejar la interfaz de usuario antes y después de invocar a las funciones genéricas *total_resultados()*, *combinaciones()* y *permutaciones()*. Dejamos su análisis para el alumno.

Bibliografía

[1] Python Software Foundation, "Python Documentation," 2015. [Online]. Available: https://docs.python.org/3/. [Accessed 24 February 2015].

[2] M. Pilgrim, "Dive Into Python - Python from novice to pro," 2004. [Online]. Available: http://www.diveintopython.net/toc/index.html. [Accessed 6 March 2015].

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

[4] S. Lipschutz, Matemáticas para Computación, México: McGraw-Hill / Interamericana, 1993.

[5] E. Gentile, Notas de Álgebra I, Buenos Aires: Eudeba, 1976.