Ficha 11

Módulos y Paquetes

1.] Tratamiento especial de parámetros en Python.

En la *Ficha 9* hemos analizado la forma utilizar parámetros en una función en Python, y hemos visto luego que la parametrización es uno de los mecanismos que favorecen el desarrollo de funciones genéricas y reutilizables. Pero los conceptos y mecanismos que vimos en la Ficha 9 no agotan el tema: una función en Python puede plantearse de forma de aceptar y manejar un *número variable de parámetros*, en forma similar a lo que permiten hacer otros lenguajes como C y C++, o a lo que ya hace Python con algunas de sus funciones predefinidas como *max()* y *min():*

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

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

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

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

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

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

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

En lugar de definir dos funciones (una para cada tipo de ordenamiento) se puede desarrollar sólo una, que tome tres parámetros: los dos números a ordenar y un flag o bandera de tipo

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

```
__author__ = 'Catedra de AED'


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

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

    return first, second


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
__author__ = 'Catedra de AED'


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

```
    print('Tiene trabajo?:', trabaja)
    print('Estado civil:', estado)

def test():

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

    # Ok...
    datos('Pedro')

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

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

test()
```

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

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

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

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

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

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

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

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

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

```
Datos recibidos:
Nombre: Luisa
Pais: Uruguay
```

```
Sexo: Mujer
Tiene trabajo?: True
Estado civil: Soltero
```

Pero finalmente, si se ejecuta el ejemplo tal como está, *la última invocación a la función datos() asignará valores incorrectamente en los parámetros formales*: si queremos dejar el parámetro *pais* en su valor *default*, *no podemos 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*:

```
__author__ = 'Catedra de AED'


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


def test():

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

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

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

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

    # error: luego de una palabra clave, no puede seguir explícito ni default...
    # 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 ni default...
# 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

[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.

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

la salida producida es de la forma:

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

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

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

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

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

```
Valor x:--> 10

Valor y: 20
```

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

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

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

La salida producida es:

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

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

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

La salida producida es de la forma:

```
Valor x: 10
```

```
Valor y: 20
```

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

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

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

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

```
__author__ = 'Catedra de AED'


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

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


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

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


# script principal...…
test()
```

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

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

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

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

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

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

2.] Definición y uso de *módulos* en Python.

Un *módulo* es una colección de definiciones (variables, funciones, etc.) contenida en un archivo separado con extensión *.py* que puede ser importado desde un script o desde otros módulos para favorecer el reuso de código y/o el mantenimiento de un gran sistema. Se trata de lo que en otros lenguajes llamaríamos una *librería externa*. Además, un conjunto de módulos en Python puede organizarse en carpetas llamadas *paquetes* (o *packages*) que favorezcan aún más el mantenimiento y la distribución (veremos el uso de *packages* posteriormente).

La idea de usar un *módulo*, es agrupar definiciones de uso común (funciones genéricas, por ejemplo) en un archivo separado, cuyo contenido pueda ser accedido cuando se requiera, sin tener que repetir el código fuente de cada función o declaración en cada programa que se desarrolle. Esto permite que un programador reutilice con más sencillez sus funciones ya desarrolladas, y acorte la longitud de sus programas (entre otras ventajas) [1].

Para crear un módulo sólo debe escribir en un archivo con extensión *.py* las definiciones de funciones y variables que vaya a necesitar. Por ejemplo, mostramos aquí un módulo guardado en el archivo *soporte.py*, con algunas de las funciones que hemos estado usando como ejemplo en esta Ficha o en otras anteriores (vea el proyecto *[F11] Ejemplo* que acompaña a esta Ficha como anexo, tanto para el código fuente del módulo *soporte.py* como para todos los programas que siguen en esta sección):

```
__author__ = 'Cátedra de AED'

# archivo soporte.py
# Este archivo es un "modulo" que contiene funciones varias...

# una funcion para retornar el menor entre dos numeros...
def menor(n1, n2):
    if n1 < n2:
        return n1
    return n2
```

```
# una funcion para calcular el factorial de un numero...
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f


# una función para ordenar dos numeros...
def ordenar(n1, n2, ascendent=True):
    first, second = n2, n1
    if n1 < n2 :
        first, second = n1, n2
    if not ascendent :
        first, second = second, first
    return first, second
```

Para que un script o un programa pueda usar las funciones y demás declaraciones de un módulo externo, debe *importarlo* usando alguna de las variantes de la instrucción ***import*** [1]. Suponga que el siguiente programa está almacenado en un segundo archivo *prueba01.py*, *dentro de la misma carpeta donde está el módulo soporte.py*:

```
__author__ = 'Cátedra de AED'

# Archivo prueba01.py
# Un programa en Python, que incluye al módulo soporte.py

import soporte


def test():
    a = int(input('Cargue el primer  número: '))
    b = int(input('Cargue el segundo número: '))

    # invocación a las funciones del módulo "soporte"
    m = soporte.menor(a,b)
    f = soporte.factorial(m)

    # si una función se va a usar mucho, se la puede referenciar con
    # un identificador local...
    men = soporte.menor
    c = men(3,5)

    print('El menor es:', m)
    print('El factorial del menor es:', f)
    print('El segundo menor es:', c)


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

La instrucción *import soporte* que se muestra al inicio de este programa introduce el *nombre del módulo* en el contexto del programa, pero ***no*** el nombre de las funciones definidas en él. Por eso, para invocar a una de sus funciones debe usarse el operador *punto (.)* en la forma: ***nombre del módulo + punto + función a invocar***:

```
m = soporte.menor(a,b)
f = soporte.fact(m)
```

Note también que es posible asociar una función a un identificador local (que en realidad es una referencia o un puntero a esa función), a modo de sinónimo que simplifique el uso:

```
men = soporte.menor
c = men(3,5)
```

En el siguiente programa (que suponemos almacenado en otro archivo *prueba02.py*), mostramos que se puede usar la variante ***from - import*** de modo que se incluya el nombre de la función específica que se quiere acceder desde un módulo, o de varias de ellas separadas por comas, evitando así tener que usar el *operador punto* cada vez que se quiera invocarla:

```
__author__ = 'Cátedra de AED'

# Archivo prueba02.py
# Un programa en Python, que incluye al módulo soporte.py

from soporte import factorial


def test():
    a = int(input('Cargue un número entero: '))
    f = factorial(a)
    print('El factorial del número es:', f)


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

En el ejemplo anterior se incluyó la instrucción *from soporte import factorial*, la cual permite acceder directamente a la función *factorial()*, pero no al resto de las funciones del módulo. Si se quisiera acceder (por ejemplo) a las funciones *factorial()* y *menor()* pero no a otras, podría haberse usado la instrucción de esta otra forma:

```
from soporte import factorial, menor
```

En forma similar, se puede usar un *asterisco (*)* en lugar del nombre de una función particular, para lograr así el acceso *a todas las funciones del módulo sin usar el operador punto*. El siguiente programa (que suponemos almacenado en tercer archivo *prueba03.py*) muestra la forma de hacerlo:

```
__author__ = 'Cátedra de AED'


# Archivo prueba03.py
from soporte import *


def test():
    a = int(input('Cargue un número entero: '))
    f = factorial(a)
    print('El factorial del número es:', f)
    r = menor(a,f)
    print('Menor:', r)


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

Todo módulo en Python tiene definida una *tabla de símbolos* propia, en la que figuran los nombres de todas las variables y funciones que se definieron en el módulo. Esa tabla está disponible para el programa desde que el módulo se carga por primera vez en memoria con el primer import que pida acceder a ese módulo.

Existen además, en Python, algunas *variables especiales* que es importante comenzar a conocer, tal como la variable global *__name__* que todo módulo contiene en forma automática, y que siempre está asignada con el *nombre del módulo* en forma de cadena de caracteres [1] [2]. Note que el nombre de la variable (al igual que el de todas las *variables especiales* de Python) lleva ***dos*** *guiones de subrayado de* ***cada*** *lado del nombre*). Si el módulo fue correctamente cargado con una instrucción *import*, se puede acceder a la variable *__name__* con el consabido operador punto para consultar su valor, como se muestra en el siguiente programa (que suponemos guardado en el archivo *prueba04.py*)

```
__author__ = 'Cátedra de AED'

# Archivo prueba04.py
import soporte


def test():
    # invocacion a alguna funcion del modulo...
    print('Factorial de 4:', soporte.factorial(4))

    # mostrar el nombre del modulo...
    print('Nombre del modulo usado:', soporte.__name__)


test()
```

La salida del programa anterior será la siguiente:

```
Factorial de 4:  24
Nombre del modulo usado:  soporte
```

Note que todo archivo fuente de Python lleva la extensión *.py* y que en definitiva, todo archivo con extensión *.py* constituye un *módulo*… Todos los programas que hemos mostrado conteniendo nuestros scripts, funciones de ejemplo y funciones de entrada (como *test()*) *eran módulos*… y sus contenidos podían/pueden ser usados desde *otros módulos* (entiéndase: otros programas) simplemente pidiendo su acceso mediante la instrucción *import*.

Sabemos también que *podemos ejecutar un programa directamente* (ya sea desde el shell de Python, desde un IDE o desde la línea de órdenes del sistema operativo). Es decir, podemos ejecutar un módulo directamente, sin accederlo con import desde otro módulo. Esto es lo que hemos hecho desde el inicio del curso: todos nuestros programas constituían *un único módulo*, que ejecutábamos cuando queríamos desde dentro del IDE *PyCharm*.

¿Qué pasa si se pide ejecutar un módulo que *no tiene un script principal*, y sólo contiene definiciones de funciones u otros elementos (como el caso de nuestro sencillo módulo *soporte.py*)? En un caso así, ninguna función de ese módulo será ejecutada, y en la consola de salida simplemente veremos un *mensaje general indicando que el proceso ha terminado*… sin hacer nada:

```
C:\Python34\python.exe "C:/Ejemplo/soporte.py"
Process finished with exit code 0
```

Técnicamente, es útil saber que si se pide directamente la *ejecución* de un módulo (en lugar de *importar* ese módulo para ser usado desde otro), entonces la variable global *__name__* del módulo que se pide ejecutar se asigna automáticamente con el valor *"__main__"* (una cadena de caracteres). Por lo tanto, siempre podemos saber si alguien ha pedido *ejecutar* un módulo (y no *importar* ese módulo) incluyendo (***al final*** del mismo) un script con la condición [2] [1]:

```
if __name__ == "__main__":
```

En la rama verdadera de esta condición, se puede escribir el script que el programador considere oportuno ejecutar en ese caso. Por ejemplo, se puede incluir allí la invocación a la función que se haya definido como *función de entrada* (si es que había alguna). El siguiente ejemplo muestra la forma de hacer eso en el archivo *prueba05.py*:

```
__author__ = 'Cátedra de AED'

# Archivo prueba05.py
from soporte import *


def test():
    a = int(input('Cargue un número entero: '))
    f = factorial(a)
    print('El factorial del número es:', f)
    r = menor(a,f)
    print('Menor:', r)


# script principal...
# si se pidió ejecutar el modulo, entonces
# lanzar la funcion test()
if __name__ == "__main__" :
    test()
```

Remarcamos: si se utiliza este recurso, la función *test()* será ejecutada *pero sólo si se pidió a su vez ejecutar el módulo prueba05.py que la contiene; y* ***no*** *si este módulo fue* ***importado*** *por otro*. En el siguiente ejemplo, el módulo *prueba05.py* está siendo *importado* desde un script sencillo, pero la función *prueba05.test()* no será ejecutada en forma automática: el programador debe invocarla explícitamente para que se ejecute:

```
# esta línea no provoca la ejecución de la función test()...
import prueba05

# ...pero esta sí...
prueba05.test()
```

Esta forma de controlar si se pidió la ejecución de un módulo o no, finalmente ***muestra la manera más práctica de plantear un programa en Python***. Hemos visto que siempre se puede tener una función designada *ad hoc* como función de entrada, y podemos hacer que un módulo incluya simplemente una invocación a esa función al final del mismo. Pero en ese caso, si el módulo se importa desde otro (en lugar de pedir que se ejecute), la función de entrada se ejecutará y esto puede provocar un caos en el control de flujo de ejecución. La inclusión del control de ejecución indicado más arriba impide que se produzca esta situación dándole al programador el control total del caso y convirtiendo al módulo en un *módulo ejecutable*.

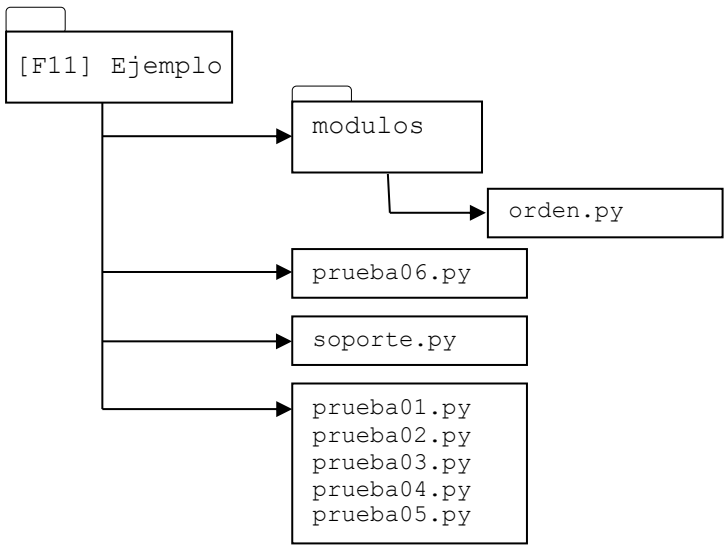
Para terminar esta sección, digamos que en los ejemplos que hemos visto hasta aquí los módulos usados eran o bien nativos (o *internos*) del lenguaje, o bien eran módulos provistos por el programador (y en este último caso, estaban alojados en la misma carpeta que contenía al programa o módulo que lo importaba). Es bueno saber qué reglas sigue el intérprete cuando encuentra una instrucción *import nombre_modulo* para encontrar el módulo requerido [1]:

1. Lo primero que analiza el intérprete, es si existe algún *módulo interno* con ese nombre.
2. Si no lo encuentra así, busca el nombre del módulo en una lista de carpetas dada por la variable *sys.path* (es decir, la variable *path* contenida en el módulo interno *sys*). Esta variable es automáticamente inicializada con los siguientes valores:
    - El nombre del directorio actual (donde está el script o programa que realizó el *import*)
    - El valor de la variable de entorno PYTHONPATH (que se usa en forma similar a la variable PATH del sistema operativo)
    - Y otros valores default que dependen de la instalación Python.

Note que la variable *sys.path* contiene una cadena de caracteres con la información enumerada en el párrafo anterior asignada por default. Sin embargo, el programador puede modificar esa variable usando funciones directas de manejo de cadenas, como la función ***append()*** que permite añadir una cadena a otra.

A modo de ejemplo, analicemos el mismo proyecto *[F11] Ejemplo* que viene anexo a esta Ficha. La estructura de carpetas y archivos de este proyecto es la siguiente:

Figura 1: Estructura de carpetas y subcarpetas del proyecto *[F11] Ejemplo*.



En esta carpeta de proyecto hay una subcarpeta llamada *módulos*, la cual contiene un único módulo muy sencillo llamado *orden.py*, cuyo contenido es el siguiente:

```
__author__ = 'Cátedra de AED'

# archivo orden.py
# Un modulo simple con una función única a modo de ejemplo simple...


# una funcion para retornar el menor entre dos numeros...
def menor(n1, n2):
```

```
        if n1 < n2:
            return n1
        return n2
```

Además, en la carpeta del proyecto *[F11] Ejemplo* se encuentran todos los programas que hemos analizado en esta sección (*prueba01.py*, *prueba02.py*, …) junto con el módulo *soporte.py* que hemos analizado hasta aquí y el programa *prueba06.py* que se muestra más abajo:

```
__author__ = 'Cátedra de AED'

# Archivo: prueba06.py

# Modificacion de la variable sys.path para incluir una subcarpeta de modulos
import sys
sys.path.append('.\\modulos')

# Importacion del modulo "orden" que está en la carpeta "modulos"...
import orden


# función de entrada a modo de prueba...
def test():
    a = int(input('Ingrese un numero: '))
    b = int(input('Ingrese otro: '))
    men = orden.menor(a, b)
    print('El menor es:', men)

# control de solicitud de ejecucion del modulo...
if __name__ == '__main__':
    test()
```

En el ejemplo que se acaba de mostrar, el archivo *prueba06.py* (un módulo ejecutable…) realiza un *import* para acceder al módulo ***orden.py***, pero como ese módulo se encuentra a su vez grabado en la subcarpeta *modulos* (como subcarpeta del proyecto *F[11] Ejemplo* que incluye a *prueba06.py*) entonces se cambia el valor de la variable *sys.path* para añadir a ella la ruta de la subcarpeta *".\\modulos"*. Note que el propio módulo interno *sys* debe ser importado para acceder a la variable *sys.path*.

Otro detalle técnico interesante es que cuando un módulo es importado por primera vez en la ejecución de un programa o script, se crea un archivo cuyo nombre incluye al nombre del módulo como parte (en el caso del módulo soporte.py, el archivo creado se llama *soporte.cpython-34*), pero con extensión *.pyc* (por *Compiled Python File*).

Este archivo contiene una versión "precompilada" del módulo (al estilo de los archivos *.class* de Java). Esto se hace para agilizar el proceso de carga de un módulo, lo cual es muy práctico si el programa usa muchos módulos ya que cada módulo no debe ser compilado nuevamente cada vez que se lo cargue: se compila la primera vez que es importado, y luego se toma el archivo *.pyc* cuando se vuelve a importar.

El lugar donde se ubican estos archivos depende del IDE que esté usando: si está trabajando con *PyCharm*, entonces en la misma carpeta del proyecto se creará una subcarpeta *__pycache__* conteniendo a ese archivo. Obviamente, esos archivos precompilados *no son editables* (el programador no puede simplemente abrir este tipo de archivos con un editor de texto), pero son independientes del sistemas operativo (se dice que son *independientes de la plataforma*), por lo cual un archivo de módulos Python precompilado puede ser compartido entre diversas arquitecturas y sistemas operativos sin tener que volver a compilarlo.

3.] La librería estándar de Python

Como es de esperar, Python provee una amplísima librería de módulos estándar listos para usar. Ya hemos visto, a modo de ejemplo, una mención al módulo *sys* que provee acceso a muchas variables usadas o mantenidas por el intérprete y/o por funciones que interactúan fuertemente con el intérprete. Solo a modo informativo (y confiando en la sana curiosidad de cada estudiante…) mostramos aquí una tabla sencilla, a modo de referencia y descripción muy simple de los distintos *módulos estándar de Python* (la lista no es exhaustiva… solo es una contribución de consulta inmediata):

Módulo estándar	Contenido general
os	Interfaz de acceso a funciones del sistema operativo.
glob	Provee un par de funciones para crear listas de archivos a partir de búsquedas realizadas con caracteres "comodines" (wildcards)
sys	Variables de entorno del shell, y acceso a parámetros de línea de órdenes. También provee elementos para redireccionar la entrada estándar, la salida estándar y la salida de errores estándar.
re	Herramientas para el reconocimiento de patrones (expresiones regulares) en el procesamiento de strings.
math	Funciones matemáticas típicas (logarítmicas, trigonométricas, etc.)
random	Funciones para el trabajo con números aleatorios y decisiones aleatorias.
urllib.request	Funciones para recuperación de datos desde un url.
smtplib, poplib	Funciones para envío y recepción de mails.
email	Gestión integral de mensajes de email, permitiendo decodificación de estructuras complejas de mensajes (incluyendo attachments).
datetime	Clases para manipulación de fechas y horas.
zlib, gzip, bz2, zipfile, tarfile	Los cinco módulos citados contienen funciones para realizar compresión/descompresión de datos, en diversos formatos.
timeit, profile, psstats	Proveen funciones para realizar medición de rendimiento de un programa, sistema, bloque de código o instrucción.
doctest	Herramienta para realizar validaciones de tests que hayan sido incluidos en strings de documentación.
unittest	Similar a doctest, aunque no tan sencillo. Permite realizar tests más descriptivos, con la capacidad de poder mantenerlos en archivos separados.
xmlrpc.client, xmlrpc.server	Permiten realizar llamadas a procedimientos remotos, como si fuesen tareas triviales.
xml.dom, xml.sax	Soporte para el parsing de documentos XML.
csv	Lectura y escritura en formato común de "comma separated values" (CSV).
gettext, locale, codecs	Soporte de "*internationalization*".

El acceso a todos los módulos nombrados (y otros que podrían no figurar en la tabla) requiere el uso de la instrucción *import* o alguna de sus variantes.

Además, y también en forma esperable, Python provee un amplio conjunto de funciones internas, que están siempre disponibles sin necesidad de importar ningún módulo. La lista completa de estas funciones puede verse en la documentación de Python que se instala junto con el lenguaje, y de todos modos en la *Ficha 3*, *página 61*, hemos mostrado oportunamente una lista de las más comunes de esas funciones.

4.] Definición de *paquetes* en Python.

Un *paquete* (o *package*) en Python es una forma de organizar y estructurar *carpetas de módulos* (y los espacios de nombres que esos módulos representan), de forma que luego se

pueda acceder a cada módulo mediante el conocido recurso del operador *punto* para especificar la ruta de acceso y el nombre de cada módulo. Así, por ejemplo, si se tiene un paquete (o sea, una *carpeta de módulos*) llamado *modulos*, y dentro de él se incluye un módulo llamado *funciones*, entonces el nombre completo del módulo sería ***modulos.funciones*** y con ese nombre deberá ser accedido desde una instrucción *import*.

Además de facilitar una mejor organización y mantenimiento de los módulos disponibles en un gran proyecto o sistema, el uso de *paquetes* permite evitar posibles ambigüedades o conflictos de nombres que podrían producirse si diversos programadores o equipos aportan módulos diferentes que incluyan funciones llamadas igual, y otros tipos similares de conflictos. El nombre completo del *módulo* incluida la ruta de acceso dentro del *paquete* y el *operador punto* para acceder a la función, evita el conflicto.

Si bien la creación de un *paquete de módulos* no conlleva un proceso complicado en Python, tampoco es tan directo como simplemente armar una estructura de carpetas y subcarpetas y luego distribuir en ellas los módulos disponibles. Para que el intérprete Python reconozca la designación de un *paquete* mediante el *operador punto* y la traduzca desde una estructura de carpetas física, esas carpetas deben incluir (*cada una de ellas*) un archivo fuente Python llamado *__init__.py*, el cual puede ser tan sencillo como un archivo vacío, o puede incluir scripts de inicialización para el paquete, o asignar valores iniciales a ciertas variables globales que luego describiremos.

Llegados a este punto, conviene aclarar un hecho práctico: no todos los IDEs para Python proveen la funcionalidad de crear y administrar paquetes de módulos en forma automática a partir de opciones de menú. Pero nuestro IDE *PyCharm* es uno de los que sí lo permiten.

Si está trabajando con *PyCharm*, y quiere crear un *paquete de módulos* dentro de un proyecto, simplemente pida crear un nuevo proyecto (en lugar de un archivo fuente) y dentro de ese proyecto incluya un *paquete*: apunte al proyecto con el mouse, haga click derecho, seleccione la opción *New* en el menú emergente que aparece, y luego seleccione el ítem "*Python Package*".

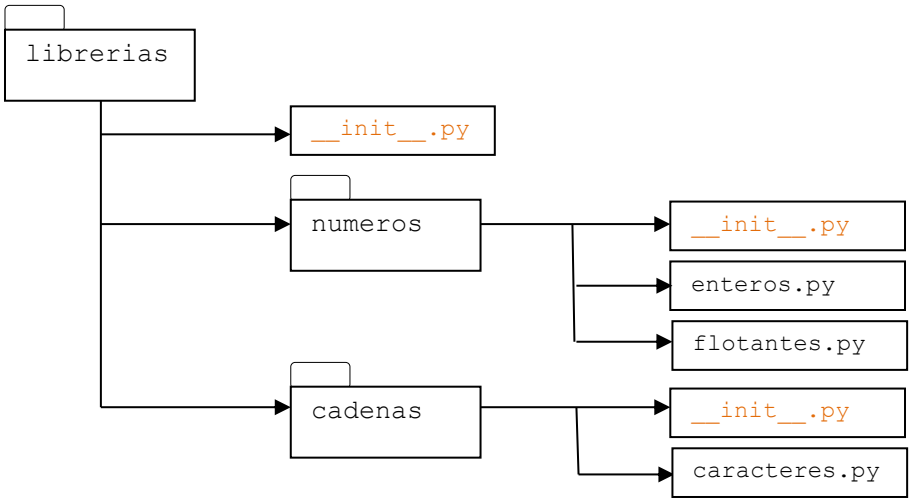
Dentro del proyecto se creará una carpeta con el nombre que usted haya elegido para el paquete, y esa carpeta ya contendrá su correspondiente archivo *__init__.py* (posiblemente vacío o solo conteniendo alguna inicialización de alguna variable global, tal como la variable *__author__* asignada con el nombre del usuario o creador del *paquete*). Luego de creado el *paquete*, puede crear *subpaquetes* dentro de él, repitiendo el procedimiento marcado más arriba, pero ahora apuntando con el mouse al *paquete* dentro del cual desee crear un *subpaquete*.

A modo de ejemplo, junto con esta Ficha se incluye otra carpeta *[F11] [Paquetes]*, que no es otra cosa que un proyecto creado con *PyCharm*. Dentro de ese proyecto se ha incluido un paquete llamado *librerias*, que contiene a su vez dos subpaquetes llamados *cadenas* y *numeros* respectivamente. Los archivos *__init__.py* de estas tres carpetas solo contienen la inicialización de la variable *__author__* (puede abrir estos archivos con el mismo editor de *PyCharm* si quiere ver y/o editar su contenido).

A su vez, el paquete *librerias.numeros* contiene dos módulos de funciones: uno se llama *enteros.py*, el otro se llama *flotantes.py*, y contienen algunas funciones sencillas para manejar números. El paquete *librerias.cadenas* contiene solo un módulo llamado

caracteres.py que incluye alguna función simple para manejo de cadenas y caracteres. El siguiente esquema gráfico muestra la estructura del paquete completo:

Figura 2: Estructura de carpetas del paquete librerias.



Note que cada carpeta o subcarpeta que forma parte de la estructura del paquete se considera ella misma como un paquete o un subpaquete, pero debe tener para ello su propio archivo *__init__.py* (vacío, con scripts de inicialización, o con asignación en alguna variable de control).

El módulo *números.enteros* tiene el siguiente contenido:

```
__author__ = 'Cátedra de AED'


# una funcion para retornar el menor entre dos numeros...
def menor(n1, n2):
    if n1 < n2:
        return n1
    return n2


# una funcion para calcular el factorial de un numero...
def factorial(n):
        f = 1
        for i in range(2,n+1):
            f *= i
        return f


# una función para ordenar dos numeros...
def ordenar(n1, n2, ascendent=True):
    first, second = n2, n1
    if n1 < n2:
        first, second = n1, n2

    if not ascendent :
        first, second = second, first

    return first, second
```

El módulo *numeros.flotantes* tiene a su vez el siguiente contenido:

```
__author__ = 'Cátedra de AED'


# una funcion para obtener el promedio entre los parametros
def promedio (x1, x2, *x):
    suma = x1 + x2
    conteo = 2

    n = len(x)
    if n != 0:
        for i in range(n):
            suma += x[i]
        conteo += n

    return suma / conteo
```

Finalmente, el módulo *cadenas.caracteres* se compone así:

```
__author__ = 'Cátedra de AED'


# una funcion que determina si la cadena tomada como parametro esta
# formada por un unico caracter que se repite
def caracter_unico(cadena):
    n = len(cadena)

    # si la cadena no tiene caracteres, retornar falso
    if n == 0:
        return False

    c0 = cadena[0]
    for i in range(1, n):
        if c0 != cadena[i]:
            return False
    return True
```

Si se desea acceder a módulos que han sido almacenados en paquetes o subpaquetes, se puede usar la ya conocida instrucción *import*, pero ahora escribiendo la ruta completa de paquetes y subpaquetes delante del nombre del módulo que se quiere acceder, separando los nombres de las carpetas con el operador punto. Desde el proyecto *F[11] Paquetes* mostramos el archivo *test01.py* con un ejemplo de uso:

```
__author__ = 'Cátedra de AED'

# Estos dos import acceden a los modulos "flotantes" y "caracteres"
import librerias.numeros.flotantes
import librerias.cadenas.caracteres


def test():
    p = librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    if librerias.cadenas.caracteres.caracter_unico('abcde'):
        print('La cadena tiene repeticiones de un unico caracter...')
    else:
        print('La cadena tiene varios caracteres distintos...')
```

```
if __name__ == '__main__':
    test()
```

Como de costumbre, la instrucción *import* introduce el nombre del módulo como un nombre válido para el programa, y si se quiere invocar a una función de ese módulo se debe seguir usando el nombre completo de la misma. La siguiente línea invoca a la función *promedio()* que se encuentra dentro del módulo *flotantes* del paquete *librerías.numeros*:

```
librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
```

También puede usar la también conocida variante *from – import* si desea ser más específico en cuanto a los elementos que quiere acceder desde un módulo (ver ahora el archivo fuente *test02.py* del mismo proyecto *[F11] Paquetes*):

```
__author__ = 'Cátedra de AED'

# el from - import que sigue importa los modulos "flotantes" y "enteros"
from librerias.numeros import flotantes, enteros

# el from - import que sigue importa la funcion caracter_unico()
from librerias.cadenas.caracteres import caracter_unico


def test():
    p = flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    m = enteros.menor(4,6)
    print('El menor es:', 4)

    if caracter_unico('aaaaa'):
        print('La cadena tiene solo repeticiones de un unico caracter...')
    else:
        print('La cadena tiene varios caracteres distintos...')

if __name__ == '__main__':
    test()
```

La primera instrucción *from – import* de este ejemplo da acceso al nombre de los módulos *flotantes* y *enteros*, de forma que ya no es necesario luego incluir toda la ruta de paquetes y subpaquetes para nombrarlos:

```
p = flotantes.promedio(2.34, 4, 5.34)
m = enteros.menor(4,6)
```

El segundo *from – import* del ejemplo, da acceso específicamente a la función caracter_unico() del módulo *caracteres*, evitando por completo el tener que poner la ruta de paquetes:

```
if caracter_unico('aaaaa'):
```

Es siempre posible usar *from – import* e incluir un asterisco (*) para garantizar que el nombre de ese paquete sea tomado en forma implícita en todos sus elementos. Una instrucción como:

```
from librerias.numeros import *
```

permite acceder a los elementos incluidos en el paquete *librerias.numeros* sin tener que repetir el prefijo *librerias.numeros*. Sin embargo, considere que un paquete podría contener un sinnúmero de subpaquetes y módulos, y eventualmente el creador del paquete principal podría querer limitar lo que se accede mediante un *from – import ** (por ejemplo, algún módulo que sólo fue creado a los efectos de permitir pruebas (o *testing*). Para lograr esa restricción, se puede recurrir al archivo *__init__.py* del paquete cuyo contenido se quiere controlar, y asignar en la variable *__all__* una lista de cadenas de caracteres con los nombres de los módulos que serán importados con un *from – import **.

En la carpeta de fuentes que acompaña a esta Ficha, se incluye otro proyecto *F[11] Import*, que a su vez contiene el mismo paquete *librerias* del modelo anterior, pero con un pequeño agregado: el subpaquete *librerias.numeros* agrega un módulo más, llamado *testing.py* con una sola función sencilla llamada *mensaje()*. El archivo *__init__.py* del paquete *librerias.numeros* contiene las siguientes instrucciones Python [1]:

```
# el autor del módulo...
__author__ = 'Cátedra de AED'

# los módulos que serán incluidos en un from numeros import *
__all__ = ['enteros', 'flotantes']
```

Como dijimos, en el proyecto se encuentra el archivo *test01.py*, que contiene lo siguiente:

```
__author__ = 'Cátedra de AED'

# el from - import que sigue importa SOLO los modulos "flotantes"
# y "enteros"... ver archivo __init__.py de este paquete...
from librerias.numeros import *


def test():
    p = flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    m = enteros.menor(4,6)
    print('El menor es:', 4)

    # esta funcion esta en el modulo librerias.numeros.testing...
    # pero ese modulo no fue incluido en el from - import *...
    # testing.mensaje('Solo para probar...')  # error…

if __name__ == '__main__':
    test()
```

Puede verse que este programa contiene una instrucción *from librerias.numeros import **, pero esta instrucción NO importará el nombre del módulo *testing.py*, ya que el mismo no había sido incluido en la lista de nombres de la variable *__all__* del paquete. De hecho, la invocación:

```
testing.mensaje("Solo para probar...")
```

está comentarizada ya que de otro modo provocará un error de interpretación.

Anexo: Temas Avanzados

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

a.) Cadenas de documentación (docstrings) en Python.

Un elemento importante a considerar en la fase de desarrollo de un sistema informático es la documentación técnica del mismo. Sabemos que esa tarea es harto tediosa y larga, y pocos profesionales de la informática gustan de hacerla. Sin embargo, todos comprendemos la importancia de dejar claramente expresado para qué sirve cada programa o módulo que hayamos desarrollado, cuál es el objetivo de cada uno, qué funciones contiene, para qué sirve cada una, qué retorna cada una, o qué significa cada uno de sus parámetros formales. En otras palabras, es importante dejar a otros miembros de un proyecto o a otros eventuales programadores los manuales técnicos de nuestros desarrollos.

Por otra parte, esa misma documentación técnica es utilizada en forma automática por algunos IDEs para ofrecer ayuda de contexto al programador cuando este tiene alguna duda respecto de como usar una función (por ejemplo). En *PyCharm*, puede apuntar con el mouse a una función cualquiera y pulsar la combinación de teclas *<Ctrl> Q* para ver una pequeña ventana emergente conteniendo la documentación técnica de esa función (si es que esa documentación existe…)

En forma similar a lo que se puede hacer en el lenguaje *Java* mediante los llamados comentarios *javadoc*, *Python* permite introducir elementos llamados *cadenas de documentación* o *docstrings* en el código fuente, de tal forma que el contenido de los bloques *docstring* pueda luego ser usado para generar archivos de documentación general sobre nuestros programas y módulos [1].

Una cadena de documentación es un string literal, encerrado entre comillas triples ("""), que puede desplegarse *en una única línea* o *en un bloque de varias líneas*:

```
"""Ejemplo de docstring de unica linea"""


"""Ejemplo de docstring de multiples lineas
Aquí continua…
Y en esta linea termina
"""
```

Los *docstrings* pueden usarse para especificar detalles de contenido y/o funcionamiento de una función, una clase, un método, un módulo o incluso un paquete. Luego, la información contenida en los *docstrings* de cualquiera de estos elementos puede ser obtenida por herramientas analizadoras de código que generen texto de ayuda (designadas en general como *parsers*), como ejemplo, la función *help()* del shell de Python, o la herramienta *pydoc* que viene incluida en el módulo *pydoc.py* del SDK de Python.

Hay algunas reglas y convenciones que se espera que se respeten al incluir información *docstring* en un elemento [1]:

- Una cadena *docstring debe aparecer como la primera línea* dentro del elemento que se está describiendo.
- Normalmente, la primera línea dentro de la cadena debe ser una descripción corta y somera del elemento que se está documentando (típicamente, no más de una línea de texto). Esta línea debería comenzar con letra mayúscula y terminar con un punto, y no debería contener información que describa el nombre del elemento (o sus parámetros o su tipo de retorno) ya que estos ítems normalmente son identificados automáticamente por introspección.
- Si el bloque *docstring* va a contener varias líneas, la siguiente línea debería quedar en blanco.
- A partir de la tercera línea debería realizarse un resumen un poco más detallado acerca de la forma de usar el objeto descripto, sus parámetros y valor retornado (si los hubiese), excepciones y situaciones a controlar, etc.
- La información *docstring* de un *módulo* debería generalmente listar las clases, excepciones, funciones y cualquier otro elemento que el módulo contenga, mediante una línea de texto breve para cada uno. Esta lista breve debería brindar menos detalles que el resumen general que cada objeto tendrá a su vez en sus docstring.
- La información *docstring* de un *package* se escribe dentro del archivo *__init__.py* que lo describe, y también debería incluir una lista breve de los módulos y subpaquetes exportados por ese paquete.
- Finalmente, el *docstring* de una *función* debería resumir su comportamiento y documentar sus parámetros, tipo de retorno, efectos secundarios, excepciones lanzadas y restricciones de uso (si son aplicables) Los parámetros opcionales deberían ser indicados, y también sus parámetros de palabra clave.

Para más información y detalles, invitamos a consultar la página sobre convenciones de uso referidas a *docstring*, en el url http://www.python.org/dev/peps/pep-0257/ (incluido entre la *documentación PEP* de consulta del sitio oficial de Python).

El proyecto *[F11] Docstrings* incluido con esta Ficha, contiene el mismo proyecto que ya presentamos como *F[11] Paquetes*, pero de tal forma que ahora los paquetes, módulos y funciones del proyecto han sido brevemente documentados mediante *docstrings*. Así, el archivo *__init__.py* del paquete *librerias* contiene ahora lo siguiente:

```
"""El paquete contiene subpaquetes para operaciones con numeros y con cadenas.

Lista de subpaquetes incluidos:
:cadenas: Contiene un modulo con funciones para manejo de caracteres
:numeros: Contiene dos modulos con funciones para numeros enteros y en coma
flotante
"""
__author__ = 'Cátedra de AED'
```

Note que el bloque de documentación *docstring* comienza desde la primera línea del archivo *__init__.py*, y que sólo después de este bloque aparece la consabida asignación en la variable *__author__* del archivo. En forma similar se ha procedido con los archivos *__init__.py* de los subpaquetes *numeros* y *cadenas*.

El módulo *enteros.py* luce ahora en la forma siguiente (y el resto de los módulos del proyecto en forma similar):

```
    """Funciones generales para manejo de numeros enteros.

    Lista de funciones incluidas:
```

```
    :menor(n1, n2): Retorna el menor entre dos numeros
    :factorial(n): Retorna el factorial de un numero entero
    :ordenar(n1, n2, ascendent = True): Ordena dos numeros
    """
    __author__ = 'Cátedra de AED'


    def menor(n1, n2):
        """Retorna el menor entre dos numeros.

        :param n1: El primer numero a comparar
        :param n2: El segundo numero a comparar
        :return: El menor entre n1 y n2
        """
        if n1 < n2:
            return n1
        return n2


    def factorial(n):
        """Retorna el factorial de un numero.

        :param n: El numero al cual se le calculara el factorial
        :return: El factorial de n, si n>=0. Si n<0, retorna None.
        """
        f = 1
        for i in range(2,n+1):
            f *= i
        return f


    def ordenar(n1, n2, ascendent = True):
        """Ordena dos numeros.

        :param n1: El primero de los numeros a ordenar
        :param n2: El segundo de los numeros a ordenar
        :param ascendent: True ordena de menor a mayor - False en caso contrario
        :return: Los dos numeros, ordenados segun ascendent
        """
        first, second  = n2, n1
        if n1 < n2 :
            first, second = n1, n2
        if not ascendent :
            first, second = second, first
        return first, second
```

Observe que el *docstring* de cada función aparece dentro del bloque de la función, pero inmediatamente luego de la cabecera de la misma (como dijimos: el *docstring* debe ir en la primera línea del objeto descripto). Dependiendo del IDE que esté usando, el programador podrá disponer de mayor o menor ayuda de contexto para generar esta documentación (el IDE *PyCharm*, por ejemplo, incorpora abundantes elementos de ayuda en el editor de textos o en sus menúes de opciones).

Las cadenas literales que conforman los *docstrings* de un objeto cualquiera (módulo, función, etc.) se asignan en la variable global *__doc__* que forma parte de los atributos o elementos contenidos en ese objeto. En ese sentido, el archivo *test01.py* del mismo proyecto que estamos analizando a modo de ejemplo, incluye una simple invocación a la función *print()* para mostrar el valor de *__doc__* en forma directa [1]:

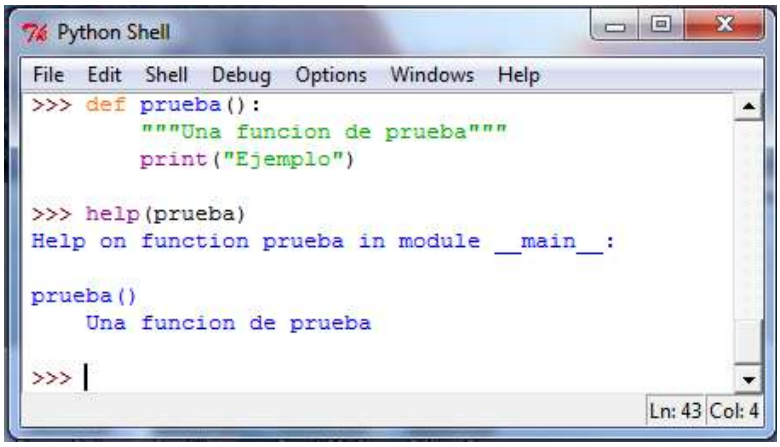
```
__author__ = 'Cátedra de AED'

import librerias.numeros.flotantes
import librerias.cadenas.caracteres
```

```
def test():
    p = librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
    print('Promedio:', p)

    if librerias.cadenas.caracteres.caracter_unico('abcde'):
        print('La cadena tiene una o varias repeticiones de un unico caracter...')
    else:
        print('La cadena tiene varios caracteres distintos...')

    print('Contenidos docstring del modulo:')
    print(librerias.cadenas.__doc__)

if __name__ == '__main__':
    test()
```

La salida producida por esas dos instrucciones *print()* es la siguiente:

```
Contenidos docstring del modulo:
El paquete contiene un modulo con funciones para operar con caracteres y
cadenas.

Lista de modulos incluidos:
:caracteres: Contiene funciones simples para manejo de caracteres
```

Una vez que los docstrings se han incluido en el código fuente, se puede generar la documentación de ayuda mediante diversas herramientas más o menos sofisticadas. Si ha trabajado en el shell directamente, una invocación a la función *help()* tomará las cadenas que haya incluido como *docstrings* y las mostrará directamente en la consola de salida (sin guardar nada en ningún archivo externo). Sólo debe enviarle como parámetro el nombre del elemento cuyos docstrings quiere observar:



Obviamente, lo anterior resulta muy limitado cuando se trata de programas o sistemas grandes desarrollados desde un IDE. Python provee para estos casos el *módulo ejecutable pydoc.py* que permite generar la documentación de ayuda en forma similar a la aplicación *javadoc* de Java (aunque *pydoc* es bastante menos flexible).

El módulo *pydoc.py* está almacenado en la carpeta Lib del SDK de Python (típicamente, en Windows la ruta de esa carpeta es algo como *C:\Program Files\Python 3.3.0\Lib*). Lo normal es ejecutar el módulo desde la línea de órdenes del sistema operativo, para lo cual esta ruta debería estar asociada la variable PATH del sistema operativo.

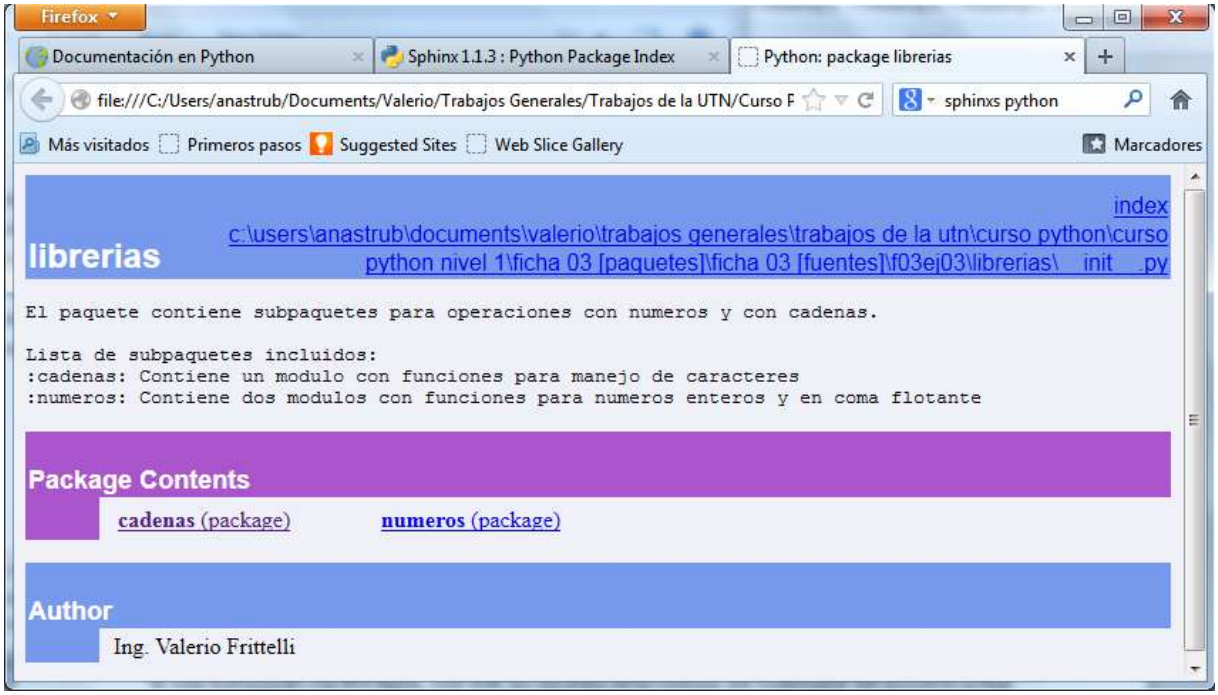
En estas condiciones, se puede acceder a la consola de ejecución por línea de órdenes del sistema operativo, luego cambiarse a la carpeta que contenga al paquete o módulo cuya documentación docstring se quiere analizar, y desde allí ejecutar el módulo pydoc enviándole el nombre del objeto a analizar. Por ejemplo, nuestro proyecto está en la carpeta "*C:\F[11] Docstrings*" y dentro de esa carpeta se encuentra ya el paquete *librerías*, entonces desde la línea de órdenes del sistema la siguiente orden mostrará en pantalla los docstrings de ese paquete:

```
C:\F[11] Docstrings>pydoc.py librerias
```

El módulo *pydoc.py* admite ciertos parámetros de ejecución, como el parámetro **–w** que indica al parser que almacene la documentación en archivos con formato html en lugar de mostrarla por consola:

```
C:\F[11] Docstrings>pydoc.py –w librerias
```

La orden anterior tomará los *docstrings* del paquete *librerias*, creará un documento html con esos elementos, y almacenará el documento en la misma carpeta actual (en este caso, la carpeta del proyecto). Así se ve el archivo html generado de esta forma, cuando se lo despliega en un navegador web:



Puede acceder a subpaquetes y/o submódulos mediante su nombre calificado vía el operador punto. Lo siguiente creará un archivo html con los docstrings del paquete *librerias.numeros* (y note que el archivo quedará almacenado en la misma carpeta del proyecto, con el nombre *librerias.numeros.html*)

```
C:\F[11] Docstrings>pydoc.py –w librerias.numeros
```

Si está trabajando con *PyCharm*, este IDE incorporará en la ventana del explorador del proyecto actual los documentos html que se generen de esta forma, y el programador podrá ver su código fuente (y editarlo si lo desea). También permitirá acceder a esta información

para un elemento en particular, colocando el puntero del mouse sobre él (el nombre de una función, por ejemplo) y pulsando la combinación de teclas *<Crtl> + Q*.

Si bien el módulo *pydoc.py* permite otros parámetros de ejecución, el hecho es que aún así resulta muy poco flexible como herramienta integrada (por caso, no permite modificar el estilo de los documentos html generados agregando etiquetas html especiales). Esa flexibilidad puede lograrse mediante herramientas especiales para generación de documentación (normalmente, proyectos desarrollados por organizaciones que incluso podrían ser externas a Python Software Foundation), que deben ser descargadas e instaladas por separado. Algunas de estas herramientas son muy comunes y muy usadas por la comunidad Python. Indicamos a continuación una breve lista de algunas de ellas, y dejamos para el lector la tarea de investigar formas de uso y aplicaciones:

i. ***Docutils***: es un sistema modular para procesamiento de documentación hacia formatos útiles como HTML, XML y LaTeX. A modo de entrada, la herramienta *Docutils* soporta un estándar de marcas de formato fácil de leer, llamado *reStructuredText*. Se puede descargar desde: https://pypi.python.org/pypi/docutils.

ii. ***EpyDoc***: es una de las herramientas de generación de documentación para Python más utilizadas. Incluye la definición de su propio formato (epytext). Produce salidas de texto plano, pero soporta también *reStructuredText* y sintaxis *Javadoc*. Se puede descargar desde el siguiente url: https://pypi.python.org/pypi/epydoc.

iii. ***Sphinxs***: Originalmente planteada para Python, se puede usar sin problemas con C y C++ y se planea expandirla a otros lenguajes. Soporta HTML, LaTeX, Texinfo y texto plano. Se puede descargar desde el url: https://pypi.python.org/pypi/Sphinx.

Créditos

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

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

Bibliografía

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

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