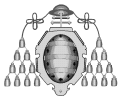

Universidad de Oviedo
Grado en Física

Introducción a la Física Computacional

Rosario Díaz Crespo, Julio Manuel Fernández Díaz

Curso 2015–16

Python como lenguaje algorítmico: Segunda parte

Práctica 5

Python como lenguaje algorítmico: Segunda parte

5.1. Objetivo

Familiarizarse con las estructuras de control de flujo y las funciones de entrada y salida.

5.2. Estructuras de control de flujo

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Por un lado, en los programas no existiría la posibilidad de elegir uno de entre varios caminos en función de ciertas condiciones (sentencias alternativas). Y por el otro, no podrían ejecutar algo repetidas veces, sin tener que escribir el código para cada una (sentencias repetitivas).

Para estos dos problemas tenemos dos soluciones: las sentencias de control alternativas y las repetitivas.

Las sentencias alternativas nos permiten seleccionar uno de entre varios caminos por donde seguirá la ejecución del programa. Este tipo de sentencias se denominan `Condicionales`.

Las sentencias repetitivas se denominan también sentencias iterativas—ya que permiten realizar algo varias veces (repetir, iterar)—, y de manera más corriente `Bucles`.

5.2.1. Condicionales

Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de una condición. Aquí es donde cobran su importancia el tipo booleano y los operadores lógicos y relacionales.

Existen diferentes formas de construir un condicional:

- La forma más simple es un `if` (del inglés "si") seguido de la condición a evaluar, dos puntos (`:`) y en la(s) siguiente(s) línea(s) e **indentado**, el código a ejecutar en caso de que se cumpla dicha condición, su sintaxis es:

```
if condición:
    código a ejecutar
```

- Una segunda forma de la sentencia `if` es la ejecución alternativa, en la que hay dos posibilidades, y la condición determina cuál de ellas se ejecuta. Su sintaxis es:

```
if condición:
    código a ejecutar
else:
    código a ejecutar
```

- La construcción `if ... elif ... elif ... else` se utiliza para condiciones múltiples (`elif` es una contracción de `else if`). Su sintaxis es:

```
if condición:
    código a ejecutar
elif condición:
    código a ejecutar
else:
    código a ejecutar
```

Ejemplo 5.2.1. *Uso de la construcción* `if ... else`.

1. *Cree un fichero con el nombre* `paridad_if.py` *y escriba el siguiente código:*

```
a = 10
if a%2 == 0:
    print (str(a)+' es par')
else:
    print (str(a)+' es impar')
```

2. *Ejecute el programa.*

Ejemplo 5.2.2. *Uso de la construcción* `if ... elif ... elif ... else`.

1. *Cree un fichero con el nombre* `signo.py` *y escriba el siguiente código:*

```
a = 1.5
if a < 0.0:
```

```
    print('El signo de a es negativo')
elif a > 0.0:
    print('El signo de a es positivo')
else:
    print('a es igual a cero')
```

2. *Ejecute el programa.*

Ejercicio 5.2.1. Sea la ecuación de segundo grado $ax^2 + bx + c = 0$. Prepare un código en el que dados los tres parámetros a, b y c nos determine el signo del discriminante $\Delta = b^2 - 4ac$ y según el mismo calcule las raíces reales o complejas conjugadas de la ecuación y las imprima.

5.2.2. Bucles

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición. Las construcciones que nos permiten crear un bucle son las sentencias `while` y `for`.

La sintaxis de la sentencia `while` (en inglés "mientras") es de la forma:

```
while condición:
   código a ejecutar
```

El código se ejecuta mientras la condición sea verdadera. Este proceso continuará hasta que la condición sea falsa.

Ejemplo 5.2.3. *Uso de la construcción while.*

1. *Cree un archivo con el nombre* `crea_lista_w.py` *y escriba el siguiente código:*

```
nMax = 5
n = 1
a = [ ] # crea una lista vacía
while n < nMax:
    a.append(1.0/n) # añade un elemento a la lista
    n = n + 1
print(a)
```

2. *Ejecute el programa escribiendo:*

```
python crea_lista_w.py
```

3. *Observe la respuesta.*

Ejemplo 5.2.4. *Uso de la construcción* `while` *junto con* `if`.

1. *Cree un fichero con el* `nombre escribe_pares.py` *y escriba el siguiente código:*

```
nMax = 10
n = 1
a = [ ] # crea una lista vacía
while n < nMax:
    if n%2 == 0: # si n es par
       a.append(n) # añade un elemento a la lista
    n = n + 1
print(a)
```

2. *Ejecute el programa.*

Al igual que con la construccion `if`, con la sentencia `while` también podemos utilizar más de dos ramas usando el condicional encadenado, cuya sitaxis es:

```
while condición:
   código a ejecutar
else:
   código a ejecutar
```

Ejercicio 5.2.2. Genere la sucesión de números entre 1 y n (que es un dato) que sean a la vez múltiplos de 11 y 13, imprimiendo término a término.

Ejercicio 5.2.3. Modifique el programa anterior para que guarde los múltiplos de 11 y 13 en una lista, e imprima la lista al final.

La construcción `for ... in` (en inglés "para ... en") también nos permite realizar bucles y su sintaxis es:

```
for elemento in secuencia:
   código a ejecutar
```

Para cada elemento de la secuencia ejecuta las líneas de código. En Python `for` se utiliza para recorrer secuencias (listas, tuplas o cadenas), por lo que vamos a utilizar un tipo secuencia, como es la lista, para nuestro ejemplo.

Ejemplo 5.2.5. *Uso de la construcción* `for ... in`.

1. *Cree un archivo con el nombre* `crea_lista_f.py` *y escriba el siguiente código:*

```
nMax = 5
```

```
a = [ ]
for n in range(1, nMax):
    a.append(1.0/n)
print(a)
```

Aquí la secuencia es la lista `[1, 2, ..., nMax-1]`*, que hemos creado llamando a la función* `range`*.*

2. *Ejecute el programa y observe la respuesta.*

Otra manera de hacerlo es con comprensión de listas, tal y como se ilustra en el siguiente ejemplo.

Ejemplo 5.2.6. *Comprensión de listas*

1. *Cree un archivo con el nombre* `crea_lista_fc.py` *y escriba el siguiente código:*

```
nMax = 5
a = [1.0/n for n in range(1, nMax)]
print(a)
```

2. *Ejecute el programa y compruebe el resultado.*

Ejercicio 5.2.4. Modifique el último ejercicio, generando la lista de números múltiplos de 11 y 13, mediante comprensión de listas.

Se puede iterar en una lista mediante el *índice* y el *elemento* usando la función de lista `enumerate`.

Ejemplo 5.2.7. *Uso de la función* `enumerate`*.*

1. *Invoque el modo interactivo de Python.*
2. *Escriba:*

```
>>> z = ['a', 5, (1.0, 2.0)]
>>> for i, e in enumerate(z):
...       print(i, e)
...
```

3. *Presione <Intro> y observe el resultado.*
4. *Ahora escriba:*

```
>>> for i, e in enumerate(z):
...       print(e)
...
```

5. *Presione <Intro> y compare la respuesta con la anterior.*

Ejercicio 5.2.5. Cree una lista con los números enteros de 1 hasta n (que es un dato) y luego modifique los términos múltiplos de 3 cambiándolos de signo. Finalmente imprima la lista final para comprobar que el programa funciona bien.

Ejemplo 5.2.8. *Un bucle puede finalizarse con la sentencia* `break`*. En el ejemplo siguiente, para obtener lo que el usuario escriba en pantalla utilizamos la función* `raw_input` *(que luego veremos).*

1. *Cree un archivo con el nombre* `lista_nombres.py` *y escriba el siguiente código:*

```
lista = ['Marta', 'Juan', 'Pedro', 'Carmen']
nombre = raw_input('Escribe un nombre: ')
encontrado = False
for i in range(len(lista)):
    if lista[i] == nombre:
       print(nombre+' es el número '+str(i+1)+' de la lista')
       encontrado = True
       break
if not encontrado: print(nombre+' no está en la lista')
```

2. *Ejecute el programa y observe la respuesta.*

La sentencia `continue` salta directamente a la siguiente iteración del bucle.

Ejemplo 5.2.9. *Uso de la sentencia* `continue`*.*

1. *Crea un archivo con el nombre* `lista_div7.py` *y escriba el siguiente código:*

```
x = [ ]
for i in range(1,100):
    if i%7 != 0: continue
    x.append(i)
print(x)
```

2. *Ejecute el programa y observa la respuesta.*

Ejercicio 5.2.6. Cree un fichero con el nombre `TemperaturasFC.py` que calcule las temperaturas en la escala Fahrenheit equivalentes el rango de temperaturas desde -20°C hasta 40°C con saltos de 5 °C. Para ello utilice un bucle `while` para crear dos listas `Fdeg` y `Cdeg` que almacenen respectivamente los valores de las temperaturas en grados Fahrenheit y en grados Celsius.

Ejercicio 5.2.7. Repita el ejercicio anterior utilizando un bucle `for` en lugar de `while`.

5.2.3. Excepciones

La sentencia `try` (en inglés "intentar") permite controlar errores de ejecución sin que el programa aborte. Su sintaxis es la siguiente:

```
try:
    sentencias que se prueban
except:
    sentencias que se ejecutan en caso de error
else: # opcional
    sentencias que se ejecutan si no hay error
finally: # opcional
    sentencias que se ejecutan siempre al final
```

Ejemplo 5.2.10. *Uso de excepciones.*

1. *Cree un archivo con el nombre* `excep.py` *y escriba el siguiente código:*

   ```
   x = input('x ? ')
   try:
       y = 1/x
   except:
       print('se divide por cero')
   else:
       print('no se divide por cero')
   finally:
       print('esto se ejecuta siempre al final')
   ```

2. *Ejecute el programa y vea la salida usando diferentes entradas enteras (incluyendo* `0`*).*

Para comprobar una afirmación que si es falsa nos muestra un mensaje de error y aborta el programa, y si es verdadera se continua la ejecución, se usa `assert` (podríamos traducirlo por "comprobar").

Ejemplo 5.2.11. *Uso del comando* `assert`*:*

1. *En el programa del ejemplo anterior añadiremos una aserción de manera que si introducimos un número negativo aborte el programa. Para ello introduzca el siguiente código en la segunda línea:*

   ```
   assert x > 0, 'x debe ser no negativa'
   ```

2. *Ejecute nuevamente el programa introduciendo números positivos y negativos para comprobar que funcione.*

Ejercicio 5.2.8. Modifique el programa anterior para que muestre el resultado de calcular el inverso de x.

5.3. Entrada y salida de datos

5.3.1. Entrada estándar

La forma más sencilla de obtener información por parte del usuario es mediante la función `raw_input`. Esta función toma como parámetro una cadena a usar como prompt (texto a mostrar al usuario pidiendo la entrada) y devuelve una cadena con los caracteres introducidos por el usuario.

```
nombre = raw_input("¿Como te llamas?")

print("Encantado, " + nombre)
```

Si necesitamos un valor numérico (incluso una expresión matemática) como entrada en lugar de una cadena utilizaremos la función `input`.

La forma de introducir una variable numérica es mediante la sentencia:

```
a = input(prompt)
```

Ejemplo 5.3.1. *Uso de las función* `input` .

1. *Invoque el modo interactivo de Python y escriba:*

   `>>> a = input('Introduce a: ')`

   al pulsar <Intro> la respuesta será:

   `Introduce a :`

2. *Introduzca un número (por ejemplo el 2)*

3. *Pulse <Intro> y escriba:*

   `print(a, type(a))`

   la respuesta será:

   `(2, <type 'int'>)`

En Python v3 `raw_input` no existe. Hay que usar directamente `input` y para introducir una variable numérica es mediante la sentencia `eval(input(prompt))`

5.3.2. Salida estándar

La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la función `print` (en inglés "imprimir"). En Python v2 print es una instrucción y se puede usar también sin paréntesis. Es preferible usar paréntesis que funciona en Python v3.

En su forma más básica a la función `print` le sigue una cadena, que se mostrará en la salida estándar al ejecutarse la sentencia.

```
>>> print("Hola mundo")

Hola mundo
```

La función `print(objeto1, objeto2, ...)` convierte cada objeto en una cadena y los muestra por pantalla en la misma línea separados por espacios. Es posible usar los caracteres especiales para cambiar la forma de mostrar los parámetros por pantalla.

Ejemplo 5.3.2. *Uso de la función* `print`.

1. *Invoque el modo interactivo de Python y escriba:*

   ```
   >>> a = 1234.56789
   >>> b = [2, 4, 6, 8]
   >>> print(a, b)
   ```

   la respuesta es:

   ```
   (1234.56789 [2, 4, 6, 8])
   ```

2. *Escriba (la función* `str` *se usa para convertir a cadena):*

   ```
   >>> print('a ='+str(a)+'\nb ='+str(b))
   ```

   la respuesta es:

   ```
   a = 1234.56789
   b = [2, 4, 6, 8]
   ```

   Nota: se salta línea debido a `\n`.

Para especificar el formato de salida de los valores es necesario usar el operador de formato `%`. Cuando se aplica a enteros, `%` es el operador de módulo, pero cuando el primer operando es una cadena, `%` es el operador de formato.

El operador `%` puede utilizarse para construir una tupla. La sentencia de conversión es:

```
'%format1%format2 ...'% tupla
```

Donde `format1`, `format2`, etc., son las especificaciones de formato de cada objeto dentro de la tupla. El resultado es una cadena que contiene los valores de las expresiones, formateados de acuerdo a la cadena de formato.

Las especificaciones para los formatos más utilizados son:

ws	Cadena
wd	Entero
w.nf	Coma flotante
w.ne	Notación exponencial

donde w es la anchura del campo y n es el número de dígitos después del punto decimal. Los caracteres s, d, f y e se escriben tal cual.

Ejemplo 5.3.3. *Uso del operador de formato.*

1. *Invoque el modo interactivo de Phyton y escriba:*

   ```
   >>> a = 1234.56789
   >>> n = 9876
   >>> print('%7.2f'% a)
   ```

   la respuesta será:

   ```
   1234.57
   ```

2. *Escriba:*

   ```
   >>> print('n =%6d'% n)
   ```

   la respuesta será:

   ```
   n = 9876
   ```

3. *Escriba:* `>>> print('n =%06d'% n)`

   la respuesta será:

   ```
   n = 009876
   ```

4. *Escriba (hacen falta los paréntesis para indicar la tupla):*

   ```
   >>> print('%12.4e%6d'% (a, n))
   ```

   la respuesta será:

   ```
   1.2346e+003
   9876
   ```

5. *Escriba:*

   ```
   >>> motos = 52
   >>> '%d'% motos
   ```

   la respuesta será:

   ```
   '52'
   ```

   El resultado es la cadena `'52'`*, que no debe confundirse con el valor entero* `52`*.*

   Una secuencia de formato puede aparecer en cualquier lugar de la cadena de formato, de modo que podemos incrustar un valor en una frase.

6. *Escriba:*

```
>>> motos = 52
>>> 'En julio vendimos  %d motos.'% motos
```

La respuesta es:

```
'En julio vendimos 52 motos.'
```

7. *En el ejemplo siguiente la secuencia de formato* `'%f'` *formatea el siguiente elemento de la tupla como un número en coma flotante, y* `'%s'` *formatea el siguiente elemento como una cadena. Compruébelo escribiendo:*

```
>>> 'En  %d días ingresamos  %f millones de  %s.'% (34,6.1,'euros')
```

La respuesta será:

```
'En 34 dias ingresamose 6.100000 millones de euros.'
```

Por defecto, el formato de coma flotante imprime seis decimales.

El número de expresiones en la tupla tiene que coincidir con el número de secuencias de formato de la cadena. Igualmente, los tipos de las expresiones deben coincidir con las secuencias de formato.

8. *Si escribe:*

```
>>> '%d%d%d'% (1,2)
```

la respuesta será:

```
TypeError: not enough arguments for format string
```

9. *y si escribe:*

```
>>> '%d'% 'euros'
```

la respuesta será:

```
TypeError: illegal argument type for built-in operation
```

Para tener más control sobre el formato de los números, podemos detallar el número de dígitos como parte de la secuencia de formato.

10. *Escriba:*

```
>>> '%6d'% 62
'62'
>>> '%12f'% 6.1
'6.100000'
```

El número tras el signo de porcentaje es el número mínimo de espacios que que ocupará el número. Si el valor necesita menos dígitos se añaden espacios en blanco delante del número. Si el número de espacios es negativo, se añaden los espacios tras el número.

11. *Escriba:*

```
>>> '%-6d'% 62
```

la respuesta (hemos sustituido los espacios por una barra baja para que se vean) será:

```
'62___'
```

También podemos especificar el número de decimales para los números en coma flotante.

12. *Escriba:*

```
>>> '%12.2f'% 6.1
```

la respuesta será:

```
'_______6.10'
```

el resultado ocupa doce espacios e incluye dos dígitos tras la coma. Este formato es util para imprimir cantidades con las comas alineadas.

Ejercicio 5.3.1. Modifique el programa `TemperaturasFC.py` de manera que el rango de valores de las temperaturas en °C se solicite a través de la pantalla y sea introducido por el teclado. Guarde el nuevo programa con el nombre `TemperaturasFCv1.py`.

Ejercicio 5.3.2. Modifique el programa `TemperaturasFCv1.py` de manera que los valores de las temperaturas se muestren en una tabla donde la primera columna represente las temperaturas en °C con formato entero y en la segunda columna se muestren los valores correspondientes en °F con una cifra decimal. Guarde el nuevo programa con el nombre `TemperaturasFCv2.py`.

5.4. Ficheros

Mientras un programa se está ejecutando, sus datos están en la memoria. Cuando el programa termina, o se apaga el computador, los datos de la memoria desaparecen. Para almacenar los datos de forma permanente se deben poner en un archivo o fichero. Antes de poder escribir en un fichero o acceder a los datos del mismo, se debe crear un objeto fichero con el comando `open` (en inglés "abrir"):

```
objeto_fichero = open(nombre_fichero, acción)
```

siendo `nombre_fichero` la cadena que especifica el fichero que va a ser abierto, incluido su camino de búsqueda (`path`) si es necesario. La `acción` es una de las siguientes cadenas:

Acción	Definición
`'r'`	Leer de un fichero existente (se usa por defecto si no se especifica la acción)
`'w'`	Escribir en un fichero. Si el fichero no existe lo crea, si existe lo sobreescribe.
`'a'`	Añade al final de un fichero.
`'r+'`	Lee y escribe en un fichero existente.
`'w+'`	Lo mismo que `'r+'`, pero si el fichero no existe, lo crea.
`'a+'`	Lo mismo que `'w+'`, pero los datos son añadidos al final del fichero.

Es una buena práctica de programación cerrar el fichero si no se va a acceder a él durante un tiempo largo dentro del programa. Para ello ejecutamos el comando:

```
objeto_fichero.close()
```

Existen varias maneras de leer datos de un fichero:

- `s = objeto_fichero.read(n)`

  Lee `n` caracteres y los devuelve como una cadena `s`. Si se omite `n`, se leerán todos los caracteres del fichero. Se puede crear una lista con todas las líneas usando:

  `s.splitlines()`

- `objeto_fichero.readline(n)`

  Lee `n` caracteres de una línea y los devuelve como una cadena. Los caracteres los devuelve como cadena que termina con `\n`. Si se omite `n` se lee toda la línea.

- `objeto_fichero.readlines()`

  Devuelve una lista de las líneas desde la primera línea a la última línea del fichero.

También existen diferentes maneras de escribir en un fichero:

- Para escribir una cadena:

  `objeto_fichero.write()`

- Para escribir una lista de cadenas:

  `objeto_fichero.writelines()`

  Ninguno de los métodos añade el carácter `'\n'` al final de la linea.

- Utilizando la función `print` para escribir en un fichero de salida:

  `print(objeto1, objeto2, ..., file=objeto_fichero)`

  La función `print` siempre añade el carácter `'\n'` al final de la linea.

Las funciones `read`, `readline`, `readlines`, `write`, `writelines` y otras que se usan de la manera:

`objeto.función(...)`

se denominan **métodos** asociados al objeto (es una notación de la programación orientada a objetos).

Ejemplo 5.4.1. *Abrir y cerrar ficheros.*

1. *Invoque el modo interactivo de Python y escriba:*

   ```
   >>> f = open('datos.dat', 'w')
   >>> print(f)
   ```

   la respuesta será:

   ```
   <open file 'datos.dat', mode 'w' at 0xb74cb180> (o algo similar)
   ```

   La función `open` *toma dos argumentos. El primero es el nombre del archivo y el segundo es el modo. El modo* `'w'` *significa que lo estamos abriendo para escribir. Si no hay un archivo llamado* `test.dat` *se creará. Si ya hay uno, el archivo que estamos escribiendo lo reemplazará. Al imprimir el objeto archivo, vemos el nombre del archivo, el modo y la localización del objeto.*

2. *Para cerrar el fichero escriba:*

```
>>> f.close()
```

3. *Si se intenta abrir un fichero que no existe, se recibe un mensaje de error. Escriba:*

```
>>> f = open('medidas.dat', 'r')
```

la respuesta será:

```
IOError: [Errno 2] No such file or directory: 'medidas.dat'
```

Ejemplo 5.4.2. *Uso de la orden* `write`.

1. *Cree nuevamente una variable* `f` *que apunte al objeto fichero en modo* `'w'` *con el nombre* `texto.dat`

2. *A continuación escriba:*

```
>>> f.write('Ya es hora')
>>> f.write('de cerrar el archivo')
```

3. *El cierre del fichero le dice al sistema que hemos terminado de escribir y lo deja listo para leer. Para cerrar el fichero escriba:*

```
>>> f.close()
```

Ejemplo 5.4.3. *Lectura del archivo creado en el ejemplo anterior.*

1. *Ya se puede abrir el archivo de nuevo para lectura y poner su contenido en una cadena. Esta vez el argumento de modo es* `'r'` *para lectura. Escriba:*

```
>>> f = open('texto.dat', 'r')
```

2. *Escriba:*

```
>>> lectura = f.read()
>>> print(lectura)
```

la respuesta será:

```
Ya es horade cerrar el archivo
```

No hay un espacio entre 'hora' y 'de' porque no se escribió un espacio entre las cadenas.

3. *La función* `read` *también puede aceptar un argumento que le indica cuántos caracteres leer, para ello escriba:*

```
>>> f = open('texto.dat', 'r')
>>> print(f.read(7))
```

la respuesta será:

```
Ya es h
```

4. *Si no quedan suficientes caracteres en el archivo,* `read` *devuelve los que hubiera. Escriba:*

   ```
   >>> print(f.read(1000006))
   ```

   la respuesta será:

   ```
   orade cerrar el archivo
   ```

5. *Cuando se llega al final del archivo,* `read` *devuelve una cadena vacía. Escriba:*

   ```
   >>> print(f.read())
   ```

Ejemplo 5.4.4. *Crear un archivo de texto con tres líneas de texto separadas por saltos de línea.*

1. *Escriba:*

   ```
   >>> f = open('lineas.dat', 'w')
   >>> f.write('línea uno\nlínea dos\nlínea tres\n')
   >>> f.close()
   ```

2. *El método* `readline` *lee todos los caracteres hasta e inclusive el siguiente salto de línea. Escriba:*

   ```
   >>> f = open('lineas.dat', 'r')
   >>> print(f.readline())
   ```

   la respuesta será:

   ```
   línea uno
   ```

3. `readlines` *devuelve todas las líneas que queden como una lista de cadenas:*

   ```
   >>> print(f.readlines())
   ['l\xc3\xadnea dos\n', 'l\xc3\xadnea tres\n']
   ```

   En este caso, la salida está en forma de lista, lo que significa que las cadenas aparecen con comillas y el carácter de acentuado aparece como la secuencia `\xc3\xad` *(que es de la codificación de caracteres UTF8).*

4. *Al final del archivo, readline devuelve una cadena vacía y readlines devuelve una lista vacía. Para comprobarlo escriba:*

   ```
   >>> print(f.readline())
   >>> print(f.readlines())
   ```

El argumento de `write` debe ser una cadena, por lo que para poner otros valores en un archivo, hay que convertirlos antes en cadenas. La forma más fácil de hacerlo es con la función `str`. Una alternativa es usar el operador de formato `%` visto anteriormente.

Ejemplo 5.4.5. *Uso de la función* `str`*.*

1. *Cree una variable f que apunte a un objeto fichero que se llame* `num.dat` *que esté preparado para escribir:*

   ```
   >>> f=open('num.dat', 'w')
   ```

2. *Escriba en él el valor de la variable* `x` *de la forma:*

   ```
   >>> x = 52
   >>> f.write(str(x))
   ```

3. *Cierre el fichero y ejecute las órdenes correspondientes para su lectura tal y como mostramos anteriormente. La respuesta observada debe ser:*

   ```
   '52'
   ```

Cuando se crea un archivo nuevo abriéndolo y escribiendo, el nuevo archivo va al directorio en uso (aquél en el que estuviese al ejecutar el programa). Del mismo modo, cuando abre un archivo para leerlo, Python lo busca en el directorio en uso. Si quiere abrir un archivo de cualquier otro sitio, tiene que especificar la ruta del archivo, que es el nombre del directorio (o carpeta) donde se encuentra éste.

Ejemplo 5.4.6. *Para ilustrar el uso de directorios se propone abrir un archivo llamado words que está en un directorio llamado* `dict`*, que está en* `share`*, que está en* `usr`*, que está en el directorio raiz del sistema, llamado* `/`*. Para ello hay que escribir:*

```
>>> f = open('/usr/share/dict/words', 'r')
>>> print(f.readline())
```

Ejercicio 5.4.1. Cree un programa que genere un fichero de datos llamado `temperaturasC.dat` donde se guarden las temperaturas Celsius desde -20 °C a 40 °C con un salto de 5 °C. Guarde el programa en una archivo con el nombre de `generaTempC.py`. Ejecute el programa y compruebe que se genera el fichero `temperaturasC.dat` de forma correcta.

Ejercicio 5.4.2. Cree un programa que lea el archivo de datos `temperaturasC.dat` y que convierta los grados Celsius en grados Fahrenheit y genere un archivo de resultados llamado `TemperaturasFC.dat` donde se guarde la tabla de tempetaturas con el mismo formato que en el programa `TemperaturasFCv2.py`. Guarde este programa con el nombre de `generaTempFC.py`. Ejecute el programa y compruebe que funciona correctamente.