My sample book

The Jupyter Book Community

Dec 05, 2020

CONTENTS

This is a small sample book to give you a feel for how book content is structured.

Check out the content pages bundled with this sample book to get started.

CHAPTER

ONE

CONTENT IN JUPYTER BOOK

There are many ways to write content in Jupyter Book. This short section covers a few tips for how to do so.

1.1 Markdown Files

Whether you write your book's content in Jupyter Notebooks (`.ipynb`) or in regular markdown files (`.md`), you'll write in the same flavor of markdown called **MyST Markdown**.

1.1.1 What is MyST?

MyST stands for "Markedly Structured Text". It is a slight variation on a flavor of markdown called "CommonMark" markdown, with small syntax extensions to allow you to write **roles** and **directives** in the Sphinx ecosystem.

1.1.2 What are roles and directives?

Roles and directives are two of the most powerful tools in Jupyter Book. They are kind of like functions, but written in a markup language. They both serve a similar purpose, but **roles are written in one line**, whereas **directives span many lines**. They both accept different kinds of inputs, and what they do with those inputs depends on the specific role or directive that is being called.

Using a directive

At its simplest, you can insert a directive into your book's content like so:

```
```{mydirectivename}
My directive content
```
```

This will only work if a directive with name `mydirectivename` already exists (which it doesn't). There are many pre-defined directives associated with Jupyter Book. For example, to insert a note box into your content, you can use the following directive:

```
```{note}
Here is a note
```
```

This results in:

Note: Here is a note

In your built book.

For more information on writing directives, see the MyST documentation.

Using a role

Roles are very similar to directives, but they are less-complex and written entirely on one line. You can insert a role into your book's content with this pattern:

```
Some content {rolename}`and here is my role's content!`
```

Again, roles will only work if `rolename` is a valid role's name. For example, the `doc` role can be used to refer to another page in your book. You can refer directly to another page by its relative path. For example, the role syntax `{doc}`intro`` will result in: *Welcome to your Jupyter Book*.

For more information on writing roles, see the MyST documentation.

Adding a citation

You can also cite references that are stored in a `bibtex` file. For example, the following syntax: `{cite}`holdgraf_evidence_2014`` will render like this: [HdHPK14].

Moreoever, you can insert a bibliography into your page with this syntax: The `{bibliography}` directive must be used for all the `{cite}` roles to render properly. For example, if the references for your book are stored in `references.bib`, then the bibliography is inserted with:

````
```{bibliography} references.bib
```
````

Resulting in a rendered bibliography that looks like:

Executing code in your markdown files

If you'd like to include computational content inside these markdown files, you can use MyST Markdown to define cells that will be executed when your book is built. Jupyter Book uses *jupytext* to do this.

First, add Jupytext metadata to the file. For example, to add Jupytext metadata to this markdown page, run this command:

```
jupyter-book myst init markdown.md
```

Once a markdown file has Jupytext metadata in it, you can add the following directive to run the code at build time:

````
```{code-cell}
print("Here is some code to execute")
```
````

When your book is built, the contents of any `{code-cell}` blocks will be executed with your default Jupyter kernel, and their outputs will be displayed in-line with the rest of your content.

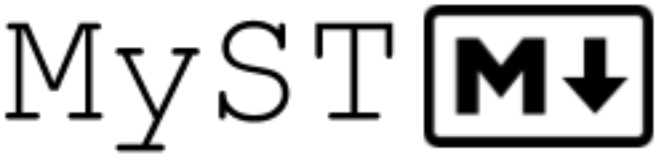
For more information about executing computational content with Jupyter Book, see The MyST-NB documentation.

1.2 Content with notebooks

You can also create content with Jupyter Notebooks. This means that you can include code blocks and their outputs in your book.

1.2.1 Markdown + notebooks

As it is markdown, you can embed images, HTML, etc into your posts!



You an also add_{math} and

$$math^{blocks}$$

or

$$\text{mean} la_{tex}$$

$$mathblocks$$

But make sure you \$Escape \$your \$dollar signs \$you want to keep!

1.2.2 MyST markdown

MyST markdown works in Jupyter Notebooks as well. For more information about MyST markdown, check out the MyST guide in Jupyter Book, or see the MyST markdown documentation.

1.2.3 Code blocks and outputs

Jupyter Book will also embed your code blocks and output in your book. For example, here's some sample Matplotlib code:

```
from matplotlib import rcParams, cycler
import matplotlib.pyplot as plt
import numpy as np
plt.ion()
```

```
# Fixing random state for reproducibility
np.random.seed(19680801)

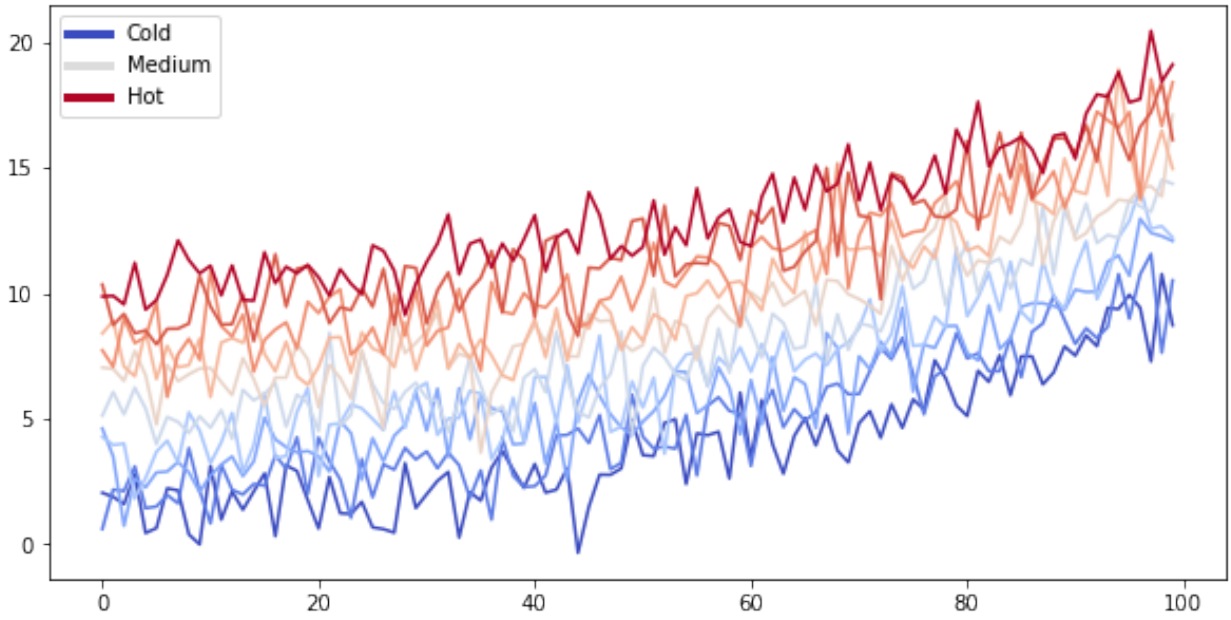
N = 10
data = [np.logspace(0, 1, 100) + np.random.randn(100) + ii for ii in range(N)]
data = np.array(data).T
cmap = plt.cm.coolwarm
rcParams['axes.prop_cycle'] = cycler(color=cmap(np.linspace(0, 1, N)))
```

(continues on next page)

(continued from previous page)

```
from matplotlib.lines import Line2D
custom_lines = [Line2D([0], [0], color=cmap(0.), lw=4),
                Line2D([0], [0], color=cmap(.5), lw=4),
                Line2D([0], [0], color=cmap(1.), lw=4)]

fig, ax = plt.subplots(figsize=(10, 5))
lines = ax.plot(data)
ax.legend(custom_lines, ['Cold', 'Medium', 'Hot']);
```



There is a lot more that you can do with outputs (such as including interactive outputs) with your book. For more information about this, see the Jupyter Book documentation

1.3 La entrada/salida.

Autores: Rogelio Mazaeda Echevarría, Félix Miguel Trespaderne.

1.3.1 Contenidos

Introducción La función `print()` La función `input()` Salida con formato

1.3.2 Introducción

El trabajo directo con el intérprete de Python, aunque útil para realizar pruebas puntuales, no resulta la estrategia más apropiada para escribir programas de cierta envergadura. La opción preferida en la práctica es la de escribir, utilizando un editor de texto cualquiera, las sentencias de Python necesarias. Al fichero creado, conocido como **guion** (**script**) se le da un nombre arbitrario con extensión `.py`.

El programa así almacenado podrá ser ejecutado directamente desde una terminal del sistema operativo, con un comando similar al siguiente:

```
python primer_programa.py
```

También se puede editar y ejecutar un **guion** desde **entornos de desarrollo integrado (IDE)** como *Spyder*, utilizando los recursos que los mismos ponen a disposición del programador.

Los programas almacenados en el archivo son ejecutados por el intérprete, sentencia a sentencia, en el orden en que estas aparezcan en el texto del mismo.

El comportamiento de una misma sentencia puede ser ligeramente diferente si se ejecuta directamente por el intérprete o como parte de la ejecución de un programa salvado previamente en un fichero. En particular, en este último caso, la evaluación de una sentencia que contiene, por ejemplo, una expresión, no produce la salida inmediata por pantalla del resultado parcial.

```
# Programa que calcula el área de un cilindro de radio y altura dados

radio = 2
altura = 3
PI = 3.14159

area_seccion = PI*radio**2
area_lado = 2*PI*radio*altura
area_cilindro = 2*area_seccion + area_lado
```

El programa anterior puede ser editado y almacenado en un fichero, para posteriormente ser ejecutado siguiendo alguna de las formas comentadas.

El programa es, desde luego, muy simple y tiene además el defecto de que, tal y como está, su utilidad es muy limitada:

- En primer lugar, para hallar el área exterior de un cilindro de `altura` y/o `radio` diferente, habría que editar el texto del programa para modificar las líneas pertinentes y volverlo a salvar para posteriormente ejecutarlo.
- Otro defecto importante es que no se visualiza por pantalla el resultado del cálculo.

Nuestra experiencia en la utilización de programas de ordenador nos sugiere otra forma de actuar:

- se ejecuta el programa cuantas veces sea necesario
- en cada ocasión, se introducen datos diferentes según las necesidades utilizando el teclado del ordenador
- se visualiza, típicamente por pantalla, el resultado que se desea

O sea, un elemento básico que le falta al programa anterior son las facilidades de **entrada** de datos y **salida** de resultados.

Un programa mucho más útil se muestra en la siguiente celda.

La función `print()`

```
# Programa que calcula el área de un cilindro de radio y altura dados por el usuario.
# Saca por pantalla el resultado
from math import pi

radio = 2
altura = 3

area_seccion = pi*radio**2
area_lado = 2*pi*radio*altura
area_cilindro = 2*area_seccion + area_lado

print("El área del cilindro es:")
print(area_cilindro, "unidades al cuadrado")
```

```
El área del cilindro es:
62.83185307179586 unidades al cuadrado
```

Sin haber introducido aún la entrada de datos, el código anterior es capaz de mostrar por pantalla el contenido de la variable que almacena el resultado del cálculo. Para ello se utiliza la función `print()` nativa del lenguaje Python.

Otra mejora importante se deriva de utilizar una mejor aproximación del número π que se encuentra disponible, como una constante, en el módulo `math`. Al hacer esto, además evitamos cambios accidentales del valor de π.

La función `input()`

Para lograr la introducción vía teclado de los valores de los datos (`radio` y `altura`) se utiliza la función `input()` tal como se muestra a continuación:

```
## Programa que calcula el área de un cilindro con entrada y salida
from math import pi

cadena_radio = input()
radio = float(cadena_radio)

# Las dos líneas anteriores se codifican de forma más compacta e informativa para el␣
↪usuario como:
# radio = float(input('Dame el radio:'))

altura = float(input('Dame la altura:'))

area_seccion = pi*radio**2
area_lado = 2*pi*radio*altura
area_cilindro = 2*area_seccion + area_lado

print("El área del cilindro es:", area_cilindro)
```

```
---------------------------------------------------------------------------
StdinNotImplementedError                  Traceback (most recent call last)
<ipython-input-3-e02e668ae2ef> in <module>
      2 from math import pi
      3
----> 4 cadena_radio = input()
```

(continues on next page)

(continued from previous page)

```
      5 radio = float(cadena_radio)
      6

C:\Anaconda3\lib\site-packages\ipykernel\kernelbase.py in raw_input(self, prompt)
    856         if not self._allow_stdin:
    857             raise StdinNotImplementedError(
--> 858                 "raw_input was called, but this frontend does not support␣
↪input requests."
    859             )
    860         return self._input_request(str(prompt),

StdinNotImplementedError: raw_input was called, but this frontend does not support␣
↪input requests.
```

Observe que la función `input()` tiene como argumento opcional una cadena de caracteres que ha de servir para informar al usuario (por pantalla) de lo que se espera introduzca por teclado.

Por otra parte, se debe notar que la función `input()` devuelve un valor que es de tipo `str`. Es cierto que si el usuario ha hecho lo correcto, esa cadena de caracteres contendrá los dígitos y signos que puedan ser interpretados como un número real, pero para poder realizar aritmética con dicho número hay que obtener la representación numérica del mismo como un valor de tipo `float`. Para lograrlo se utiliza la función nativa de Python `float()`, que espera como argumento una cadena y devuelve el tipo de datos real que la representa (en caso de que no haya errores).

La solicitud de la variable `radio` no se hace de una forma **amigable** para el usuario (**user friendly**). Si este no conoce la dinámica del programa le será imposible saber qué se le está solicitando.

La solicitud del valor de la variable `altura` corrige el defecto de la anterior de no contener un indicación clara al usuario. Al mismo tiempo resulta más compacta, puesto que utiliza la capacidad de composición de las funciones para aplicar directamente la función `float()` a la cadena que devuelve la función `input()`.

Otra función útil en este contexto, que hace una tarea similar, es `int(cadena, base)` que convierte la cadena `cadena` a un entero de la `base` dada. Por defecto (en ausencia del argumento base) se presupone la base decimal.

Por otro lado, la función `str()` convierte en el sentido inverso: diferentes valores numéricos a su representación como cadena de caracteres.

```
str(1.2)
```

```
'1.2'
```

1.3.3 Salida con formato

Habrá notado que la función `print()` puede tener un número variable de argumentos:

- cuando estos son cadenas de caracteres, los muestra tal cual en la consola de salida, dispositivo genérico normalmente asociado a la pantalla.
- cuando el argumento de la función no es una cadena de caracteres, implícitamente la función `print()` realiza la conversión requerida desde el tipo de dato original a `str`.

El funcionamiento por defecto de `print()` es tal que, al finalizar la salida por pantalla, escribe un carácter que representa el cambio del cursor de la pantalla hacia una nueva línea, el carácter `\n`. En la mayoría de los casos, este es el comportamiento adecuado.

En algunas ocasiones, sin embargo, se podría requerir que el cursor permaneciera en la misma línea después de ejecutar la función `print()`. Para ello, se puede incluir un argumento invocado mediante el parámetro `end` de forma que contenga el carácter que la función debe utilizar al final de la línea.

Vea el ejemplo siguiente:

```
vol = 3
print("El valor del volumen es:", end=" ")
print(vol)
```

```
El valor del volumen es: 3
```

El funcionamiento ya descrito de `print()` es suficiente para una salida por pantalla básica, útil en muchos casos.

En ocasiones se desea tener un control más detallado de la forma en que los valores van a ser ofrecidos al usuario. Puede desearse, por ejemplo:

- mostrar solo un determinado número de dígitos significativos de determinado valor, o
- reservar un espacio en pantalla específico para sacar datos en forma de tabla, respetando la alineación de las columnas.

Para estos casos, se utiliza preferentemente la opción de especificar formatos en las cadenas de caracteres.

Por ejemplo:

```
area_base = 10.6666
area_lado = 20.3891
area_total = 2*area_base + area_lado

print("El área de la base es {:.2f}, el del lado {:.2f} y el área total es {:.3f}".
      format(area_base,area_lado,area_total))
print("Cambiando el orden total: {2:.2f} es 2*{0:.2f} + {1:.2f}".
      format(area_base,area_lado,area_total))
```

```
El área de la base es 10.67, el del lado 20.39 y el área total es 41.722
Cambiando el orden total: 41.72 es 2*10.67 + 20.39
```

Observe el use de las llaves `{}` para introducir dentro de la cadena de caracteres *referencias* a valores que serán proporcionados mediante el **método** `.format()`. Estas referencias tienen el **formato** `{[n]:[tam][.precisión]formato}`. Se debe entender que los corchetes no se mostrarán por pantalla, sino que, como es costumbre a la hora de describir la sintaxis de algunas sentencias, significa que su contenido puede ser omitido.

- `n` se utiliza para especificar el número de orden del valor que debe ser embebido tal y como aparece en los argumentos de `.format()`, comenzando en cero. Si se omite esta especificación, se asume el mismo orden en que los argumentos aparecen en `.format()`.
- `tam` establece el tamaño del campo en pantalla; de no existir, se tomará el espacio necesario, cualquiera que este sea.
- `.precisión` especifica el número de lugares decimales al que se redondeará el valor. Esto es aplicable en el caso de que se trate de un valor real.
- `formato` es una letra que identifica el tipo del valor: `f` para `float`, `d` para `int` y `s` para `str` son los más habituales.

Véase el ejemplo siguiente. Haced modificaciones y razonad el resultado.

```
print("{:5d}{:15.3f}".format(123,1.234343))
print("{1:5f}{0:15d}".format(1,1180.2))
```

```
  123          1.234
1180.200000              1
```

1.4 Introducción

Los **saltos incondicionales** son sentencias que permiten **interrumpir** la secuencia natural de ejecución.

C++ dispone de 4 tipos de saltos incondicionales:

- `break`: Sale de forma automática del último bucle abierto (`for`, `while`, `do while`) o de una sentencia `switch`.
- `continue`: Da un salto hasta el final del último bucle abierto. (Apenas se usa y no lo utilizaremos en el curso)
- `goto`: Salta a una línea **etiquetada** del programa. (No es recomendable usarlo y no lo utilizaremos en el curso)
- `return`: Salida de una función, pudiendo devolver un valor (Se verá más adelante cuando estudiemos funciones)

Nos centraremos en este tema en el salto incondicional `break`.

1.5 Salto incondicional `break`

Supongamos un programa cuya introducción de datos debe interrumpirse cuando se cumpla una de las dos condiciones preestablecidas siguientes:

- El número de datos introducido supera un valor máximo.
- Se cumple una determinada condición, en cuyo caso abandonaremos la introducción de datos.

Veamos dos alternativas, una sin usar `break` y otra utilizándolo.

Ejemplo con `while` sin usar `break`

```
// Suma de los números positivos introducidos: sin break

#include <iostream>
using namespace std;

int main()
{
  int num_datos;
  cout << "Número máximo de datos a leer: ";
  cin >> num_datos;
  int i{}, dato{}, suma{}; // A cero con inicialización uniforme
  while (i < num_datos && dato >= 0)
  {
    cout << "Introduzca un dato (negativo para finalizar): ";
    cin >> dato;
    if (dato >= 0)
    {
      suma += dato;
      ++i;
    }
  }

  if (i > 0)
  {
    cout << "Se han introducido " << i << " datos válidos.\n";
    cout << "La suma de los datos introducidos es " << suma << ".\n";
  }
  else
    cout << "No se han introducido datos válidos.\n";
}
```

Edita, compila y ejecuta el código

Al final del tema anterior sugeríamos que antes de usar un determinado tipo de bucle debíamos preguntarnos sobre si conocíamos el número de iteraciones a realizar de antemano.

En este ejemplo, la respuesta podría ser **NO**, si prevemos que el usuario no completará todos los posibles valores a introducir, es decir, el bucle `while` terminará al evaluarse la expresión `dato >= 0` a `false`.

Sin embargo, podríamos adoptar una postura más *optimista* y pensar que el usuario introducirá todos los posibles valores y, por tanto, la salida del bucle `while` será debida a la evaluación de la expresión `i < num_datos` a `false`. Y en este supuesto es donde el bucle `for` parece más oportuno.

Veámoslo en la siguiente versión del programa.

Ejemplo con `for` usando `break`

```
// Suma de los números positivos introducidos: usando break

#include <iostream>
using namespace std;

int main()
{
  int num_datos;
  cout << "Número máximo de datos a leer: ";
  cin >> num_datos;

  int i{}, suma{}; // A cero con inicialización uniforme

  for (i = 0; i < num_datos; ++i)
  {
    cout << "Introduzca un dato (negativo para finalizar): ";
    int dato;
    cin >> dato;
    if (dato < 0)
      break;
    suma += dato;
  }

  if (i > 0)
  {
    cout << "Se han introducido " << i << " datos válidos.\n";
    cout << "La suma de los datos introducidos es " << suma << ".\n";
  }
  else
    cout << "No se han introducido datos válidos.\n";
}
```

Edita, compila y ejecuta el código

En este ejemplo, la sentencia `break` permite contemplar el caso en el que el usuario no complete la introducción de todos los valores, al introducir un valor negativo.

El salto incondicional `break` ha sido considerado en muchos ámbitos, sobre todo relacionados con la docencia, una mala práctica de programación.

La realidad es que, en muchos casos, el uso de `break` facilita la legibilidad del código y, por ello, es recomendable su uso.

Vamos a ver cómo un bucle `for` es lo suficientemente versátil para permitir una tercera versión del ejemplo, sin usar `break`.

Ejemplo con `for` sin usar `break`

```
// Suma de los números positivos introducidos
// Con for sin usar break

#include <iostream>
using namespace std;

int main()
{
  int num_datos;
  cout << "Número máximo de datos a leer: ";
  cin >> num_datos;

  int i{}, suma{}, dato{}; // A cero con inicialización uniforme
  for (i = 0; i < num_datos && dato >= 0; ++i)
  {
    cout << "Introduzca un dato (negativo para finalizar): ";
    cin >> dato;
    if (dato >= 0)
      suma += dato;
    else
      --i; // Para compensar el ++i de fin de bloque for
  }

  if (i > 0)
  {
    cout << "Se han introducido " << i << " datos válidos.\n";
    cout << "La suma de los datos introducidos es " << suma << ".\n";
  }
  else
    cout << "No se han introducido datos válidos.\n";
}
```

Edita, compila y ejecuta el código

Esta versión es, sin lugar a dudas, *desafortunada*. Los bucles `for` es preferible usarlos con una condición de salida simple. Además, como el incremento del contador `i` se produce al final del bucle, debemos decrementarlo en caso de salida prematura por dejar de cumplirse la condición `dato >= 0`.

1.5.1 Centinelas

El uso de la sentencia `break` suele venir acompañada en muchos problemas del uso de una variable **centinela**, también denominada **testigo** o **bandera**.

Habitualmente la variable **centinela** tiene un tipo booleano y nos permite discriminar cuando la finalización de un bucle se ha debido o no a un salto incondicional.

Un ejemplo clásico es la determinación de si un número es primo.

Ejemplo: determinar si un número es primo (versión 1)

```
// Determina si un número entero es primo. (Versión 1)
#include <iostream>
using namespace std;

int main()
{
  int numero;
  do
  {
    cout << "Deme un entero positivo mayor que 1: ";
    cin >> numero;
    if (numero < 1)
      cout << " El valor introducido no es válido.\n";
  }
  while (numero < 1);

  bool es_primo{true};  // Variable centinela o bandera
  for (int divisor = 2; divisor < numero ; ++divisor)
  {
    if (numero % divisor == 0)
    {
      es_primo = false;
      break;
    }
  }

  cout << "El número " << numero;
  if (es_primo)
    cout << " es primo.\n";
  else
    cout << " no es primo.\n";
}
```

Edita, compila y ejecuta el código

Una advertencia: existen formas más eficientes de realizar la tarea propuesta; el código anterior debe verse como un intento inicial.

La estrategia consiste en determinar, mediante un bucle, todos los posibles divisores *legítimos*, rango `[2, numero-1]`, que harían que se pudiera decidir que el número no es primo.

Nótese que el bucle tiene una especie de carácter asimétrico:

- para concluir que el número es primo, se debe llevar el bucle hasta su conclusión, investigando todos los posibles divisores, sin hallar ningún divisor exacto.
- para concluir que el número no es primo, basta con encontrar el primer divisor exacto.

La variable **centinela** `es_primo` es la encargada de poner de manifiesto cuál de las dos situaciones se ha producido.

Por supuesto, el lenguaje nos brinda otras opciones para programar el problema anterior sin usar `break`. Como vimos anteriormente, basta incorporar la condición de activación de la salida incondicional a la expresión de la condición del `for`.

Ejemplo: determinar si un número es primo (versión 2)

```
// Determina si un número entero es primo. (Versión 2)
// Sin usar break
#include <iostream>
using namespace std;

int main()
{
  int numero;
  do
  {
    cout << "Deme un entero positivo mayor que 1: ";
    cin >> numero;
    if (numero < 1)
      cout << " El valor introducido no es válido.\n";
  }
  while (numero < 1);

  bool es_primo{true};  // Variable centinela o bandera
  for (int divisor = 2; divisor < numero && es_primo; ++divisor)
    if (numero % divisor == 0)
      es_primo = false;

  cout << "El número " << numero;
  if (es_primo)
    cout << " es primo.\n";
  else
    cout << " no es primo.\n";
}
```

Edita, compila y ejecuta el código

Dejamos al alumno transformar este ejemplo usando `while` en lugar de `for`.

En los ejemplos anteriores hemos utilizado una variable *ad hoc* para el **centinela**. En muchos problemas no es estrictamente necesario utilizarlas. Sin embargo, su uso suele mejorar la legibilidad del código.

Así, el ejemplo de la determinación de si un número es primo permite usar la variable `divisor` como centinela.

Ejemplo: determinar si un número es primo (versión 3)

```
// Determina si un número entero es primo. (Versión 3)
// Sin usar centinela explícito
#include <iostream>
using namespace std;

int main()
{
  int numero;
  do
  {
    cout << "Deme un entero positivo mayor que 1: ";
    cin >> numero;
    if (numero < 1)
      cout << " El valor introducido no es válido.\n";
  }
  while (numero < 1);

  int divisor{2};
  for (; divisor < numero; ++divisor)
    if (numero % divisor == 0)
      break;

  cout << "El número " << numero;
  if (divisor == numero)
    cout << " es primo.\n";
  else
    cout << " no es primo.\n";
}
```

Edita, compila y ejecuta el código

Véase como nuestro centinela `divisor` se define e inicializa fuera del bucle y, por tanto, podemos dejar vacía esa parte del bucle `for`.

Para terminar con esta panoplia de ejemplos, véase una última implementación con `while` muy compacta.

Ejemplo: determinar si un número es primo (versión 4)

```
// Determina si un número entero es primo. (Versión 3)
// Con while sin usar centinela explícito
#include <iostream>
using namespace std;

int main()
{
  int numero;
  do
  {
    cout << "Deme un entero positivo mayor que 1: ";
    cin >> numero;
    if (numero < 1)
      cout << " El valor introducido no es válido.\n";
  }
  while (numero < 1);

  int divisor{2};
  while (numero % divisor)
      ++divisor;

  cout << "El número " << numero;
  if (divisor == numero)
    cout << " es primo.\n";
  else
    cout << " no es primo.\n";
}
```

Edita, compila y ejecuta el código

Nótese que la salida del bucle está garantizada ya que cuando `divisor` alcanza el valor `numero`, la expresión `numero % divisor` se evalúa a `false`.

Decidir cuál de las implementaciones vistas es *superior* es cuestión de debate.

BIBLIOGRAPHY

[HdHPK14] Christopher Ramsay Holdgraf, Wendy de Heer, Brian N. Pasley, and Robert T. Knight. Evidence for Predictive Coding in Human Auditory Cortex. In *International Conference on Cognitive Neuroscience*. Brisbane, Australia, Australia, 2014. Frontiers in Neuroscience.